Design and Implementation of a Ravenscar Extension for Multiprocessors

Fabien Chouteau and José F. Ruiz

AdaCore
46 rue d'Amsterdam, 75009 Paris, France
{chouteau,ruiz}@adacore.com

Abstract. New software architectures demand increasing processing power, and multiprocessor hardware platforms are spreading as the answer to achieve the required performance. Embedded real-time systems are also subject to this trend, but in the case of real-time high-integrity systems, the properties of reliability, predictability and analyzability are also paramount.

The Ada 2005 language defined a subset of its tasking model, the Ravenscar profile, that provides the basis for the implementation of deterministic and time analyzable applications on top of a streamlined run-time system. This Ravenscar tasking profile, originally designed for single processors, has proven remarkably useful for modelling verifiable real-time monoprocessor systems.

The forthcoming Ada 2012 language proposes a simple extension to the Ravenscar profile to support multiprocessor systems using a fully partitioned approach. The implementation of this scheme is simple, and it can be used to develop applications amenable to schedulability analysis.

This paper describes the design and implementation of a restricted run time supporting the Ravenscar tasking model on a bare board multiprocessor machine for safety-critical development.

1 Introduction

The Ravenscar model for single processors defines a deterministic and analysable tasking model which can be supported with a run-time system of reduced size and complexity. It supports accurate analysis of real-time behavior using Rate Monotonic Analysis (RMA) [13] and Response Time Analysis (RTA) [12]. In recent years, research on scheduling theory for multiprocessor systems [8,6] has paved the way to timing analysis in multiprocessor systems.

Major aspects to be dealt with in the design of a multiprocessor environment are priority handling, assignment of tasks to processors (the terms processor and CPU will be used interchangeably in this paper), communication and synchronization mechanisms, time keeping, delays, and handling of external events.

According to the allocation of priorities to tasks, there are either static off-line scheduling algorithms, or dynamic policies, where priorities are calculated at run time. Dynamic-priority scheduling algorithms for multiprocessors, such as Pfair scheduling [7], can achieve better processor utilization than static-priority ones. However, the higher complexity of dynamic algorithms, their much higher run-time overhead, and

A. Romanovsky and T. Vardanega (Eds.): Ada-Europe 2011, LNCS 6652, pp. 31–45, 2011.

their lower predictability and robustness in overload situations make them less attractive for high-integrity systems. The Ravenscar profiles follows this static approach.

In terms of relationship between tasks and processors, the spectrum goes from global scheduling, where any task can be executed on any processor at any time, to partitioned scheduling, where each task is allocated for its whole lifetime to concrete processors. The schedulability of neither approach is strictly better than the other [3] (there are task systems that are feasible using a global partitioning that cannot be scheduled in a partitioned system and vice versa). However, the partitioned approach has some interesting advantages: 1) it can rely on well-known optimal monoprocessor priority-assignment schemes and timing analysis techniques (local RMA on each CPU), and 2) the run-time support is simpler.

This partitioned approach simplifies also development and testing, and eventually certification, by the physical separation between tasks executing on different processors. This concept is the major strength of the Integrated Modular Avionics (IMA) [15] architecture and the ARINC 653 [5] standard, that enables independently-produced applications to execute together on the same hardware. Note that the proposed Ravenscar extension for multiprocessors provides limited temporal partitioning (a task allocated to a given processor cannot use execution cycles from another processor) and no memory partitioning, while ARINC 653 provides a more flexible temporal partitioning (it can provide protection for tasks executing in the same processor) plus memory partitioning. However, the Ravenscar scheduling model could be supplemented with space protection, and a more flexible temporal partitioning using execution-time clocks and timers and timing events [10].

According to the Ravenscar principles of simplicity, reliability, and predictability, fully partitioned scheduling, using static-priority policy, appears as the natural extension of the monoprocessor Ravenscar profile [4]. There are tools and techniques supporting the allocation of tasks to processors, the assignment of task's priorities, and the timing analysis of the resulting systems. The major drawback of such scheduling mechanism is that the maximum worst-case achievable utilization is a third the capacity of the platform [3]. The worst-case achievable utilization is defined as the total utilization that makes any periodic task set below this limit schedulable, while there may be a task set with a total utilization above this limit which is not schedulable.

Note that finding an optimal assignment of tasks to processors is an NP-hard bin-packing problem that needs to be solved off-line in this partitioned scheme (not by the system scheduler), although there exist heuristic algorithms, such as Rate-Monotonic-Next-Fit (RMNF) [9], Rate-Monotonic-First-Fit (RMFF) [9], and Rate-Monotonic-Best-Fit (RMBF) [14].

There exist many different communication and synchronization paradigms for multiprocessor architectures, such as semaphores, monitors, message passing, etc. Restricted protected objects are used in the Ravenscar profile for this, and they will be used the same way over multiprocessors. The underlying run-time support will have to be modified to cope with the new requirement of synchronizing tasks which are potentially operating in parallel, hence demanding extra locking mechanisms.

The provision of a common high resolution time reference and precise and deterministic absolute delays is based on the use of two different hardware timers [21]. There

may be multiprocessor hardware platforms with more timers (there may even be per-processor timers), but requiring just two timers will facilitate portability to different targets.

Finally, handling of external interrupts is supported using protected procedures. Interrupt handlers could be handled by one or more CPUs in the system, but for simplicity of implementation and timing analysis the chosen approach is to allocate each interrupt to a single CPU.

This paper and the described implementation build on the ideas presented at IRTAW 2009 [17] which led to the definition of AI05-0171 [4]. The following sections will describe the specific additions to the existing monoprocessor Ravenscar profile to support multiprocessor systems using a fully partitioned approach, with each task and interrupt allocated statically to a concrete processor. This scheme can be supported by a streamlined run-time system, and applications built following this approach can apply timing analysis techniques on each processor separately (the scheduling problem for partitioned allocation is a combination of bin packing followed by single processor dispatching).

2 Definition of Ravenscar for Multiprocessors

This section provides a high-level description of the model implied by the Ravenscar extension to multiprocessors, based on the monoprocessor definition.

2.1 Task Scheduling

Scheduling is proposed as a simple extension to the monoprocessor fixed-priority preemptive scheduling algorithm supported by the Ravenscar profile, where tasks are statically allocated to processors and task migration among CPUs is not allowed. Each processor implements a preemptive fixed-priority scheduling policy with separate and disjoint ready queues. A task is only in the ready queue of one processor, and the CPU to which a task belongs is defined statically. Whenever a task running on a processor reaches a task dispatching point, it goes back to the ready queue of the same processor.

Tasks are statically allocated to processors using a new pragma (*pragma CPU*). If the pragma is not specified, the task is allocated to a default CPU.

The underlying idea is that each processor executes a statically defined set of tasks, as it would be the case for Ravenscar on a single processor.

There is a single run-time system, where the only per-processor information would be the ready and alarm queues. Some run-time data is common and shared among tasks on different processors (such as the time reference).

When internal data in the run-time system can be modified by tasks executing on different processors, we need to add inter-processor locking mechanisms (such as spin-locks or similar, see section 3.4, “Fair locks”), to guarantee mutual exclusion. The standard monoprocessor solution of disabling interrupts to guarantee that the task is not preempted before the access has been completed is not sufficient for multiprocessors.

Finally, something that must be taken into account is that the execution of a task (or an interrupt handler) in a given processor may modify another processor's ready queues (and may also force the preemption of the running task). These operations on different processors can be implemented triggering a special interrupt in the target processor, which is the one performing the actual changes in the ready queue.

2.2 Task Synchronization

The restricted library-level protected objects defined by the Ravenscar profile are used for inter- and intra-processor communication and synchronization. The same restrictions that exist in the Ravenscar profile for single processors apply to the case of a multiprocessor (a maximum of one protected entry per protected object with a simple boolean barrier using ceiling locking access).

One big advantage of monoprocessor Ravenscar is the simple and very efficient synchronization mechanism required for protected objects, where entering/exiting to/from the protected object can simply be done by just increasing/decreasing task's priorities [16].

In order to simplify timing analysis, and to allow for an efficient implementation when possible, protected objects used only by tasks within the same CPU could use the optimized monoprocessor implementation.

Protected objects for inter-processor communication would require multiprocessor synchronization mechanisms. When a task waiting on an entry queue is awaken by another tasks executing on a different processor than the waiting task, we need to use the inter-processor interrupt facility to modify the ready queues, as described in subsection 2.1, "Task Scheduling".

One possibility would have been to allocate affinities for protected objects to facilitate timing analysis of the application (not part of AI05-0171). Protected objects bound statically to a given processor (local protected objects) would never be affected by interference coming from tasks executing on other processors, as well as no interference being caused on other processors. The design of the application must take into account this fact, and static analysis (and static tools) can also help detecting the use of protected object by tasks living in different processors.

Suspension_Objects are implemented over protected objects. It may seem overkill, but given the restrictions imposed by the Ravenscar profile, protected operations are very efficient. Moreover, this allows for a generic implementation that is not dependent on the underlying support.

For the handling of shared-memory in the target multiprocessor environment (LEON3 based on SPARC V8), the standard memory model called Total Store Ordering (TSO) [19] is used. This memory model guarantees that the stores, flushes, and atomic load-stores of all processors are executed by memory serially in an order that conforms to the order in which the instructions were issued by processors. It means that memory barrier instructions are not required for consistency among different processors. The write-through caches and snooping mechanism in LEON3 guarantee memory coherency.

2.3 Interrupt Handling

The only Ravenscar-compliant mechanism to specify interrupt handlers is to attach a protected procedure. The differences in a multiprocessor system are related to mutual exclusion and assignment of interrupts to CPUs.

The mutual exclusion mechanisms for interrupt handlers will be those of the protected operations, and therefore the same considerations for intra- and inter- processor synchronization (as described in subsection 2.2, "Task Synchronization") apply.

With respect to the processors that may handle the different interrupts, multiprocessor hardware and operating systems typically allow setting and changing the affinity mask for interrupts. It means that the set of processors that may serve a given interrupt can be statically set at initialization time, or it can change dynamically. Additionally, the set can be restricted to a single processor or any number of them.

In order to simplify timing analysis, statically setting the affinity masks for interrupts is the model that fits better the Ravenscar philosophy.

Interrupts can be configured to be handled by any number of processors, and the decision of using one or more processors to handle interrupts depends on several factors. When more than one processor handle a given interrupt, a single interrupt event will be delivered to more than one processor, and hence the interrupt handler will be invoked and executed more than once (the interrupt handler will be executed concurrently on different processors). Mutual exclusion issues are handled by the underlying mechanisms, but the handler needs to take into account these multiple executions. On the one hand, it may decrease the response time, because of the highest probability of having a processor ready to handle the interrupt (not executing a highest priority activity). On the other hand, timing analysis of a system with interrupt events delivered to multiple processors is more complex, and the increased interference reduce the utilization of the system.

Setting the affinity masks of interrupts to a single processor, that may be different for each interrupt, would be the recommended approach. Allocating the interrupts to the CPUs where the tasks use them would remove the associated overhead of inter-processor synchronization.

Affinity for interrupts is not part of AI05-0171. In the current implementation this affinity is set in the startup routine, but it would be interesting to be able to use the same *pragma CPU* defined for tasks. The pragma could be attached to the definition of the protected handler.

2.4 Timing Services

In multiprocessor architectures, hardware support for timing services ranges from just a few shared hardware timers for all processors to several timers per processor.

In the reference implementation used in this paper for the LEON3 multiprocessor board [2], a single common hardware clock and a single shared hardware timer are used for all processors. It provides a common reference for all the tasks in the system, thus avoiding time drifting problems, and limits the amount of hardware resources required from the underlying platform.

Following the same approach as for the partitioned implementation of the ready queue (see subsection 2.1, "Task Scheduling"), each processor implements a separate

and disjoint delay queue. Hence, a task waiting on a delay statement will be placed in the delay queue of the processor where it is allocated.

The interrupt service routines for the two hardware timers are executed in the context of a single given processor. When a timer expires, it has an effect on the ready and delay queues of potentially any processor, not only the one where the timer interrupt is handled. The timer handler may awake tasks waiting on a delay statement, or execute the protected actions associated to either timing events or execution time timers. In any of these cases, the mechanisms to exert the required actions in the different processors are the same as those described in previous subsection 2.1, "Task Scheduling", subsection 2.2, "Task Synchronization", and subsection 2.3, "Interrupt Handling"; if there is a task with an expired delay in a different processor from the one handling the interrupt then an inter-processor interrupt is triggered on the target processor, and the handler for this inter-processor interrupt will traverse the list of local expired events, executing the required actions in the local queues (an operation in a given processor cannot directly modify the queues of another processor).

3 Design and Implementation Details

This section describes the main design decisions and implementation details of a realization of the ideas described in previous sections on a bare board LEON3 multiprocessor board [2]. The LEON3 is a 32-bit processor based on the SPARC V8 architecture with support for multiprocessing configurations. The processor used was synthesized with 2 CPU cores, configured as symmetric multiprocessing (SMP).

3.1 Starting Point

GNAT Pro supports the Ravenscar tasking model on several bare board monoprocessor architectures [16], including LEON3. The idea was to extend the LEON3 monoprocessor run-time system to support the partitioned Ravenscar model defined in this paper, making it easy to choose between the monoprocessor and multiprocessor support according to the hardware platform used.

3.2 Initialization

There is some initial work that needs to be done before jumping to the user application code, which involves both the hardware initialization (registers, devices, etc.) and setting up the run-time system (internal data structures such as the different queues). To avoid race conditions problems, this initialization is performed in a monoprocessor context, by one statically designated CPU (init CPU).

The other CPUs (slave CPUs) wait in a busy loop until initialization is done. At the end of the initialization phase, the slave CPUs are released and start the execution of the first task (highest-priority task) in their ready queue.

As an exception to the partitioned approach, the init CPU has access to all the ready queues during the initialization phase, to assign tasks on the different processors during initialization time.

In multiprocessor mode, the init CPU executes the same initialization as in the previous monoprocessor implementation, the only difference being that there is a ready queue per processor.

3.3 Task Management

Tasks and their related data structures are statically created at compile time, without any use of dynamic memory. Task affinities are specified using the new *pragma CPU* (the default processor when the affinity is not specified is *CPU'First*). The compiler inserts the affinity into the Task Control Block (TCB) so the run-time system knows where to schedule the different tasks.

The ready queues (one per processor) are internally implemented as single-link priority ordered queues, in which each thread points to the next thread to execute, and there is an array of pointers containing the first task for each processor. Task affinities never change, so each task can only be on a given ready queue. Modifications to the ready queue are performed with interrupts disabled only on the affected processor, and the rest of processors can continue their normal operations unaffected (no inter-processor interference).

Tasks are scheduled according to the FIFO within priorities policy [1, D.2] on each processor, as it is the case for a monoprocessor Ravenscar system.

3.4 Synchronization

Mutual exclusion. In the monoprocessor version of the run time, the ceiling priority rules (*Locking_Policy (Ceiling_Locking)*) and the strictly preemptive priority scheduling policy (*Task_Dispatching_Policy (FIFO_Within_Priorities)*) guarantee that protected objects are always available when any task tries to use them [18] (otherwise there would be another task executing at a higher priority), and hence entering/exiting to/from the protected object can simply be done by just increasing/decreasing the task priority. Obviously, this protection no longer works with two or more processors because of the actual parallelism.

Mutual exclusion is now guaranteed by the use of an additional multiprocessor lock (see fair locks below). Access to protected objects is performed in two phases: first, the calling task raises its priority following the ceiling locking rules, and then it tries to get the fair lock. Raising the priority ensures that at most one task per CPU will try to get the lock at a given time. Tasks on different processors trying to access the protected object will wait in a busy loop.

Note that protected objects used only by tasks within the same CPU will always get access to the fair lock, so the multiprocessor overhead in this case is negligible.

Fair locks. In this multiprocessor context, multiple tasks on different CPUs may have to be synchronized, to protect access to shared data or hardware registers for example.

The well-known spin lock algorithm is not suitable for real-time systems, like the Ravenscar run-time, because of starvation risks [20, chapter 2. Existing Solutions].

On a processor with four cores, all of them fighting for the same spin-lock. There can be a situation where lock's ownership switches from core1 to core2 infinitely. In

that case, core3 and core4 are in a state of starvation and will not be able to gain lock. Even if this situation is not likely to last a long period, it would be a major flaw in the predictability of the Ravenscar run-time.

Fair lock [20] is a cooperative mutual exclusion algorithm. When the owner of the lock wants to release, it will search for the next CPU waiting for the lock and transfer the control to it (scanning for waiting CPUs in a round-robin fashion). Therefore, unlike spin locks, the execution time is bounded and thus suitable for real-time systems.

In the case described earlier, the sequence of lock ownership will be:

- core1 → core2
- core2 → core3
- core3 → core4
- core4 → core1
- core1 → core2
- etc. . .

Fair locks are designed to synchronise tasks over two or more CPUs, but they must not be use by two tasks on the same CPU. This constraint is ensured by the properties of the Ravenscar profile and some protected (interrupts disabled) sections of code in the run-time .

Served entries. In the monoprocessor run-time, when a task (the `caller`) tries to call an entry whose barrier is closed, it becomes blocked. Then, another task (the `server`) will release the barrier, execute the `Entry_Body` (proxy model), and then wake up the `caller`.

The proxy model [11] implies that at the end of the execution of any protected procedure, that may change the state of the entry barrier, if there is a task waiting on the protected object's entry, then the barrier is evaluated, and if needed, the entry is executed by the task that opened the barrier on behalf of the queued task. It enhances efficiency, when both the `caller` and the `server` execute on the same processor, by avoiding unnecessary context switches. The self-service model (the `caller` executes always the code associated to the entry it calls) would be interesting for multiprocessors, as the entry could be executed in parallel by another processor, thus increasing the parallelism and reducing the worst-case blocking time. However, the proxy model is used for both the monoprocessor and multiprocessor implementation for maintainability, to reduce the difference between both cases.

Both models (proxy and self-service) require that the `server` wakes up the `caller`. If the `server` runs on a different CPU, it cannot directly wake up the task, because it would imply to modify the ready queue of the `caller` CPU, which is forbidden in the implemented partitioned approach.

In the multiprocessor implementation, if the two tasks are not on the same CPU, the `server` task will open the barrier, and execute the `Entry_Body` (the proxy model is still used). The difference with respect to the monoprocessor model is that it will put the `caller` task in a list (`Served_Entry_Call`, an array containing a list per processor), and then wake up the `caller` CPU using an inter-processor interrupt

(see subsection 3.6, "Interrupt Handling"). The handling of the inter-processor interrupt in the `caller` CPU will finally insert the blocked task in the ready queue as a result of the barrier being open. Of course, if the two tasks run on the same CPU, there is no need for such mechanism, so the run-time will check it and directly wake up the `caller`.

If the `caller` task has a lower priority than the currently executing task on the `caller` CPU, the inter-processor interrupt will not be triggered, and the `caller` task will be inserted in the ready queue at the next scheduling point, without triggering the inter-processor interrupt. It may look like there could be a race condition, if the priority of the `caller` task changes just after it was checked. However, this race condition is not possible because dynamic priority changes are not allowed by the Ravenscar profile restrictions.

To ensure data consistency, the `Served_Entry_Call` lists are protected by fair locks.

3.5 Time-Keeping and Delays

Clock. The implementation of the time-keeping functionality does not differ when migrating to a multiprocessor architecture. The only detail to take into account is to choose the CPU in charge of handling the clock interrupt.

As described in [21], the requirements of having a good resolution and range with 32-bit hardware timers force us to use a hardware periodic timer to store the least significant part of the clock value, while the most significant part of the same value is stored in memory, and incremented by the clock interrupt handler.

Reading the clock (the hardware least significant part plus the most significant part in memory) is not an atomic operation, and there is the possibility of a consistency problem (race condition) if an interrupt occurs between the reading of the hardware and software components of the time. The monoprocessor implementation reads the hardware clock twice to prevent this race condition, and this method can be safely used on multiprocessor systems without introducing any kind of locking mechanism.

Alarms. The implementation of alarms in the Ravenscar run-time relies on a hardware timer. Since most systems have fewer timers than processors, this resource must be shared.

Each CPU manages its own list of alarms but one processor is in charge of the alarm interrupt (Alarm_CPU).

When the alarm interrupt is triggered, the Alarm_CPU will get the time for the next alarm on each CPU. If this time has expired, the Alarm_CPU uses an inter-processor interrupt to wake up the other CPU; otherwise, the next alarm is used to reconfigure the timer.

The CPU that receives the inter-processor interrupt will wake up all the expired alarms in its own list.

Each processor has to configure the timer itself. The CPU checks if the new alarm is before the current one and reconfigures the timer if needed. We use a fair lock to avoid race condition during this operation.

```
for CPU_Id in CPU loop

   if CPU_Id /= Current_CPU then
      Alarm_Time := Get_Next_Timeout (CPU_Id);

      if Alarm_Time <= Now then

         -- Alarm expired, wake up the CPU

         Poke_CPU (CPU_Id);

      else
         -- Check if this is the next non-expired alarm
         -- of the overall system.

         if Alarm_Time < Next_Alarm_Overall then
            Next_Alarm_Overall := Alarm_Time;
         end if;
      end if;
   end if;
end loop;

if Next_Alarm_Overall /= Time'Last then
   Update_Alarm (Next_Alarm_Overall);
end if;
```

3.6 Interrupt Handling

No modifications to the interrupt handling code are required for the multiprocessor support. However, during an interrupt, the context of the running thread is saved and the execution switches to the interrupt context (interrupt stack). If the processor does not have any task to execute, there is no context to save. To handle this case, each slave CPU will create a task with minimum stack to provide an interrupt context.

The implementation currently requires setting the affinity masks of interrupts to a single processor (different interrupts may be handled by different CPUs), which is the approach recommended in subsection 2.3, “Interrupt Handling”.

Each time an event on one CPU involves a rescheduling for another CPU, the former needs a way to wake up or interrupt the latter (inter-processor interrupt). It is done by simulating an external interrupt with the LEON3 IRQ manager. Using specific registers, the run-time can trigger an interrupt on one or more CPU. Therefore, on the chosen target we can trigger an interrupt on any CPU (similar inter-processor interrupt capabilities are typically available on other multiprocessor architectures).

3.7 Sharing Code between Monoprocessors and Multiprocessors

The support for multiprocessor systems implies modifications to the previous implementation of the Ravenscar run time, and shared code between monoprocessor and multiprocessor targets.

Most of the modifications are useless in a monoprocessor context, and would lead to dead object code. For example, any call to the fair locks routines (section 3.4, “Fair locks”) is a non-sense on a monoprocessor system and would reduce the performance.

To avoid this effect, and limit the impact of the multiprocessor support, the new implementation takes advantage of the compile-time optimizations provided in GNAT.

With the following construction the code will be statically optimized. The call to the *Lock* procedure will only remain if the run-time is configured to handle more than one processor.

```
if Multiprocessor then
   Fair_Locks.Lock (Any_Lock);
end if;
```

where the *Multiprocessor* condition is defined as:

```
Multiprocessor : constant Boolean := Max_Number_Of_CPUs /= 1;
```

and *Max_Number_Of_CPUs* is a parameter of the run-time representing the maximum number of CPUs available on the target system.

Therefore, multiprocessor-specific code becomes deactivated code in monoprocessor systems, which is never present in the final application binary. The same run-time sources are used for the monoprocessor and multiprocessor targets, and the only thing that needs to be modified is the constant *Max_Number_Of_CPUs*. Once the run time is recompiled, it can be used on the new target.

4 Performance

Here are presented the results of some performance tests used to measure the impact of the new implementation. The test platform is a 40MHz LEON3 FPGA.

These measurements were taken over a large number of iterations. For each of them there will be a table with the lowest values observed (Min) and the highest ones (Max).

The first three tests compare performances of the new multiprocessor implementation with the previous run-time. Since the previous implementation does not support multiprocessor systems, these tests only run tasks on one CPU.
Tests are executed on four different run-times:

- Monoprocessor : Previous implementation
- Multiprocessor (1) : New run time, configured for 1 processor
- Multiprocessor (2) : New run time, configured for 2 processors
- Multiprocessor (16) : New run time, configured for 16 processors

4.1 Measurements

Delay until + context switch.

What is measured here is the elapsed time between the last statement executed at a task dispatching point (a “delay until”) until the first statement in the next running task.

Min:

run-time	time (μs)	ratio	diff (μs)
Mono	52.000	1.000	0.000
Mp (1)	48.625	0.935	-3.375
Mp (2)	70.125	1.348	18.125
Mp (16)	86.500	1.663	34.500

Max:

run-time	time (μs)	ratio	diff (μs)
Mono	64.500	1.000	0.000
Mp (1)	59.000	0.914	-5.500
Mp (2)	78.000	1.209	13.500
Mp (16)	96.000	1.488	31.500

Protected objects.

This is the time to switch from one task to another using a protected object (one task waits on an entry and the other task release it). We measure the time between opening the barrier and the first statement after the entry call in the waiting task.

Min:

run-time	time (μs)	ratio	diff (μs)
Mono	56.125	1.000	0.000
Mp (1)	50.000	0.890	-6.125
Mp (2)	83.500	1.487	27.375
Mp (16)	116.625	2.077	60.500

Max:

run-time	time (μs)	ratio	diff (μs)
Mono	72.500	1.000	0.000
Mp (1)	68.250	0.941	-4.250
Mp (2)	101.000	1.393	28.500
Mp (16)	134.250	1.851	61.750

Alarm precision.

These numbers correspond to the delay until lateness [1],D.9(13), which is the difference between the requested time of delay expiration and the resumption time actually attained by a task following an absolute time suspension.

Min:

run-time	time (μs)	ratio	diff (μs)
Mono	50.375	1.000	0.000
Mp (1)	54.875	1.089	4.500
Mp (2)	70.000	1.389	19.625
Mp (16)	105.875	2.101	55.500

Max:

run-time	time (μs)	ratio	diff (μs)
Mono	76.125	1.000	0.000
Mp (1)	76.000	0.998	-0.125
Mp (2)	95.000	1.247	18.875
Mp (16)	136.250	1.789	60.125

Alarm precision on slave CPU.

This show the overhead introduced by the alarm mechanism (section 3.5, "Alarms"), when alarms are served on slave CPUs. It is the same test as "Alarm precision", except that the test task is first assigned to the Alarm_CPU and then to a salve CPU.

Min:

Alarm on	time (μs)	ratio	diff (μs)
Alarm_CPU	70.000	1.000	0.000
Slave	87.750	1.253	17.750

Max:

Alarm on	time (μs)	ratio	diff (μs)
Alarm_CPU	95.000	1.000	0.000
Salve	132.125	1.390	37.125

4.2 Analysis

When the run-time is configured for one processor, we observe comparable performances, even a slight improvement. This is due to the optimization described earlier (subsection 3.7) and also to the improvements made on the run-time beside multiprocessor implementation.

With run-times configured for more than one CPU, the overhead looks like non-negligible, but once we consider that the measured elapsed times are very short, even a not very big difference represent a noticeable percentage. Looking at the actual time differences, we can see that they are in the range of a few tens of microseconds. For a 40MHz processor (the hardware used), the time difference represents a few hundreds of CPU instructions, which is actually a very low overhead.

5 Conclusions

This paper contains a description of a simple and natural extension to the Ravenscar model to address multiprocessor systems. The idea behind it is to take an Ada application (the whole partition in the Ada sense), and to statically allocate each task to a processor. Any given processor will then have a set of tasks, protected objects and interrupts to handle, that can be modelled and analyzed as a separate monoprocessor Ravenscar system.

A *pragma CPU* is used for task partitioning, a concept already supported by most operating systems (affinity management). The GNAT Pro compiler and run-time already implement this Ada 2012 extension.

With respect to the implementation, handling the access to shared resources (hardware and data) in a simple and efficient way is the main issue. Fair locks have been added for ensuring inter-processor mutual exclusion. Inter-processor interrupt facilities are the mechanisms typically used to enforce scheduling and dispatching operations on different processors.

Configuring the run-time support from a single CPU to any number of CPUs is a matter of simply specifying the required value for the constant *Max_Number_Of_CPUs*.

The proposed partitioned approach provides a simple and analyzable model which can be supported by a streamlined run-time system. It can be implemented directly on a bare-board machine or on top of operating systems supporting affinity assignments. The associated run time overhead remains small.

References

1. Tucker Taft, S., Duff, R.A., Brukardt, R.L., Plödereder, E., Leroy, P.: Ada 2005 Reference Manual. LNCS, vol. 4348. Springer, Heidelberg (2006)
2. Aeroflex Gaisler: LEON3 Multiprocessing CPU Core (2010), `http://www.gaisler.com/doc/leon3_product_sheet.pdf`
3. Andersson, B., Baruah, S., Jonsson, J.: Static-priority scheduling on multiprocessors. In: RTSS 2001: Proceedings of the 22nd IEEE Real-Time Systems Symposium. IEEE Computer Society, Los Alamitos (2001)
4. ARG: Pragma CPU and Ravenscar Profile. Tech. rep., ISO/IEC/JTC1/SC22/WG9 (2010), `http://www.ada-auth.org/cgi-bin/cvsweb.cgi/ai05s/ai05-0171-1.txt`
5. ARINC: ARINC Specification 653, Avionics Application Software Standard Interface. Aeronautical Radio, Inc. (2005)
6. Baker, T.P.: An analysis of fixed-priority schedulability on a multiprocessor. Real-Time Systems 32(1-2), 49–71 (2006)
7. Baruah, S.K., Cohen, N.K., Plaxton, C.G., Varvel, D.: Proportionate progress: A notion of fairness in resource allocation. Algorithmica 15, 600–625 (1994)
8. Carpenter, J., Funk, S., Holman, P., Srinivasan, A., Anderson, J., Baruah, S.: A categorization of real-time multiprocessor scheduling problems and algorithms. In: Handbook on Scheduling Algorithms, Methods, and Models. Chapman Hall/CRC, Boca Raton (2004)
9. Dhall, S.K., Liu, C.L.: On a real-time scheduling problem. Operations Research 26(1), 127–140 (1978)
10. Mezzetti, E., Panunzio, M., Vardanega, T.: Preservation of timing properties with the ada ravenscar profile. In: Real, J., Vardanega, T. (eds.) Ada-Europe 2010. LNCS, vol. 6106, pp. 153–166. Springer, Heidelberg (2010)
11. Giering, E.W., Mueller, F., Baker, T.P.: Implementing ada 9X features using POSIX threads: Design issues. In: Proceedings of TRI-Ada 1993, pp. 214–228 (1993)
12. Joseph, M., Pandya, P.: Finding response times in real-time systems. BCS Computer Journal 29(5), 390–395 (1986)
13. Liu, C.L., Layland, J.W.: Scheduling algorithms for multiprogramming in a hard-real-time environment. J. ACM 20(1) (1973)
14. Oh, Y., Son, H.: Tight performance bounds of heuristics for a real-time scheduling problem. Tech. rep., Department of Computer Science, University of Virginia (1993)
15. RTCA: RTCA/DO-297: Integrated Modular Avionics (IMA) Development Guidance and Certification Considerations. RTCA (August 2005)

16. Ruiz, J.F.: GNAT pro for on-board mission-critical space applications. In: Vardanega, T., Wellings, A.J. (eds.) Ada-Europe 2005. LNCS, vol. 3555, pp. 248–259. Springer, Heidelberg (2005)
17. Ruiz, J.F.: Towards a Ravenscar extension for multi-processor systems. Ada Letters 30, 86–90 (2010)
18. Shen, H., Baker, T.: A Linux kernel module implementation of restricted Ada tasking. Ada Letters XIX(2), 96–103 (1999); Proceedings of the 9th International Real-Time Ada Workshop
19. SPARC International, Inc.: The SPARC Architecture Manual (1992), version 8
20. Swaminathan, S., Stultz, J., Vogel, J.F., McKenney, P.E.: Fairlocks — a high performance fair locking scheme. In: International Conference on Parallel and Distributed Computing Systems, pp. 241–246 (2002)
21. Zamorano, J., Ruiz, J.F., la de Puente, J.A.: Implementing Ada.Real_Time.Clock and absolute delays in real-time kernels. In: Craeynest, D., Strohmeier, A. (eds.) Ada-Europe 2001. LNCS, vol. 2043, p. 317. Springer, Heidelberg (2001)