Using Hardware Support for Scheduling with Ada

Rod White

MBDA Missile Systems (UK) Ltd.
rod.white@mbda-systems.com

Abstract. Embedded hard real-time systems are often under severe pressure from a number of directions — limited processing capacity being a common issue. Having to handle a large number, or high rate, of external stimuli (interrupts) exacerbates the processor loading problem by compromising ideal processor behaviour through the disruption of performance enhancing features, such as of pipelines and cache memories. This paper reports on the use of Ada with a novel architecture that promotes the better utilisation of the processing resources for the "real" task of executing the application. The approach described in this paper is to use dedicated hardware facilities, a "Butler", to assist the processor in the management of events and the scheduling of tasks. Allied with a cooperative (non-preemptive) approach to scheduling the application, this allows the processing capacity available to the application to be increased by a significant factor. Whilst the approach is largely language independent its integration with the Ada tasking features provides for a very powerful and sympathetic platform for demanding and high-integrity applications.

1 Introduction and Scope

Embedded real-time systems are always under pressure in terms of space, weight and thermal constraints, and as if this was not enough there are ever increasing processing demands to provide enhanced functionality and higher integrity. Together these factors require that the maximum performance is extracted from the processing platform whilst ensuring deterministic behaviour.

An approach to addressing this problem, and the one that is described in this paper, is to offload key aspects of functionality to specialised hardware. Two contrasting approaches for the provision of such support are:

- Provide bespoke hardware that implements a key application specific function — an example here might be a dedicated fast Fourier transform block [1];
- Provide hardware that provides support to tasks common across the domain of real-time embedded systems — the classic example here would be the provision of a floating point co-processor [2].

The former approach can be an ideal solution for a specific subset of the domain of embedded real-time systems, however it suffers in the light of the need for

J. Real and T. Vardanega (Eds.): Ada-Europe 2010, LNCS 6106, pp. 125–138, 2010.

these systems to evolve over their lives, potentially rendering the specialised hardware obsolete. The latter approach, conversely, does not suffer from this problem as its function supports a more generalised, lower-level "need" of the system — it is thus the latter approach that is considered in this paper.

In this case, novel hardware is used to support the scheduling of threads/tasks and the related management of external events, with the goal being to reduce the processing overheads, and associated overall degradation in system performance often attributed to these functions, thereby maximising the available processing capacity for the "real" application. It has been developed and deployed in a number of systems over the last ten years. Whilst essentially language neutral, the approach described naturally integrates with the Ada tasking model and supports the provision of high performance, suitable for high integrity and safety critical applications.

Having a device to support the management of external events, and then integrating these with the internal (software-software) task interactions also directly supports the adoption of a cooperative (cf. preemptive) approach to scheduling the tasks of the application. As noted by Burns and Wellings [3] this cooperative approach in itself provides for an improvement in schedulability, reducing processor utilisation thus ensuring better margins on deadlines, also as observed here it also leads to additional benefits due to the reduction in disruption of the processor caches and pipelines.

The remainder of this paper is divided into five parts, addressing the following areas:

1. A brief survey of related work on hardware supported scheduling;
2. Description of the various forms of scheduling support devices;
3. The use and integration of the scheduling support device with Ada;
4. Practical results and issues;
5. Conclusions and opportunities for further enhancement.

2 Related Work

There have been a number of published accounts of the use of hardware supported scheduling for software systems. Most recently, for example by Nácul *et al.* [4], there has been a focus on hardware support for the scheduling of applications on symmetric multi-processor platforms. Earlier examples, are those by Nakano *et al.* [5], and Lai *et al.* [6]; in the latter case the hardware element of the scheduler is restricted to the management of the queues. Scheduling and communication functions are implemented in software.

In the Ada domain Burns and Wellings [3] note that there have been a number of examples of the use of hardware to directly support the execution of Ada programs. Perhaps the most notable here is the support provided to Ada tasking by ATAC described by Roos [7]; other examples of "Ada machines" being those from Runner and Warshawsky [8], and Ardö [9].

3 Scheduling Support Devices

The scheduling support device is not a single physical device — there have been several incarnations each "driven" by the technology of the processing platform; to-date the most important implementations are:

- Embedded as one of many functions in a semi-custom ASIC using asynchronous design techniques, tightly coupled to the processor;
- Implemented as logic and state machines in an FPGA coupled to the processor over a PCI bus;
- Implemented as a distributed solution across the two PowerPC processors embedded in a System-on-Programmable-Chip (SoPC) FPGA — in this case one of the processors is dedicated to handling the external events, with the other being reserved for the application and software-software scheduling controls. Communication between the two processors is through dual-ported shared memory.

Other, software only, versions have also been implemented to allow the same scheduling behaviour to be realised on non-embedded platforms and those where provision of the dedicated hardware support is not a practical or cost effective option — generally these are not platforms where the absolute real-time behaviour is important. The generic term used to describe these devices is a "Butler" [10]; so-called because it has a role very similar to that of the butler in a large house, it organises activities according to some protocol but defers to others for the final decisions, in this case the final decisions are made by a minimal software kernel.

3.1 Concept — Structures and Operations

Whilst each of the above technologies is quite different the fundamentals of operation remain constant across all of them. The basic structure of a Butler device consists of four groups of resources:

- A set of activities — schedulable items;
- A set of pollsets — groupings of activities which are considered to be at the same priority. Within each pollset a simple round robin approach to scheduling its activities is adopted;
- A set of stim-wait (or control) nodes — providing a pool of resources for each activity that allow it to wait for, and be stimulated by, particular events;
- A mechanism to determine the next activity to be executed.

The number of each of the above is dependent on the implementation technology, and the demands of the target applications; typically the device would allow for 64 .. 128 activities and pollsets, and 24 stim-wait nodes per activity. Each activity exists in the context of a pollset, which gives it an effective priority — multiple activities can be associated with a single pollset hence assigning them the same priority.

In addition to the structures a number of "instructions" are provided to manipulate these structures allowing a simple, software scheduler kernel to be layered over the device. These include:

- `Suspend` — causes the currently running activity to release control and make itself schedulable, this allows another activity at either a higher or the same priority (i.e. in the same pollset) to run if it is ready;
- `Wait` <*BitVector*> — causes the current activity to become blocked (setting the *Wait* bits defined in the *BitVector*), the *BitVector* determines which events can cause it to become schedulable again;
- `Stim` <*Activity, BitVector*> — sets the stim bits defined in the *BitVector* for the *Activity*, if there is a match between the *Stim* and *Wait* bits of any stim-wait node for the *Activity* it becomes schedulable;
- `Next_Activity` <*Activity*> — returns the next activity to be scheduled;
- `Curract` <*Activity*> — returns the number of the activity that is currently running (this is not maintained by the scheduler kernel as maintaining multiple copies of the information could lead to inconsistencies);
- `AMI` <*Boolean*>— *Anything More Important*, used with cooperative scheduling to determine if there is a ready activity at a higher priority than the current one;

Other instructions are provided for the initialisation of the Butler and to support "housekeeping" operations.

In addition to the above, each activity is associated with a "watchdog" timer that can be programmed to trigger if the activity executes for longer than a desired duration, these appear to provide direct support for an efficient implementation of the `Ada.Execution_Time.Timers` features introduced in Ada 2005 [11].

The Butler sets *Stim* bits in response to hardware events and the software `Stim` "instruction" described above, it evaluates these to determine if there are any stim-wait matches. When such matches are found, and the pollset (priority) of the activity with the match is higher than that currently scheduled, the AMI status is set *True* and, if in preemptive mode, the processor will be interrupted. At this point the actual activity to be scheduled is undecided, the decision is only finalised once a `Suspend` or `Wait` instruction has been executed. These cause the device to freeze the next activity selection logic and identify the highest priority ready activity. Where there are multiple ready activities in a single pollset the selection logic is conditioned by the last activity which executed in that pollset. The `Next_Activity` instruction returns the chosen activity and unfreezes the selection logic — any events that occurred between the `Wait/Suspend` and the `Next_Activity` instructions are now evaluated.

When no activity is schedulable the `Next_Activity` instruction returns the value `Activity'Last` (e.g. 64 or 128); this is a special case activity that is always considered to be schedulable. It is also the activity that the Butler considers to be "active" when the device is in a reset state — nothing else can run until a `Suspend`–`Next_Activity` sequence has been executed.

3.2 Use

The basic approach to using the device is to allocate each thread in the system to a separate activity, and organise the activities into pollsets according to the priorities of the associated threads. The activities are then all initialised and allowed to execute until they set a *Wait* condition for one of their set of stim-wait nodes, resulting in the activity becoming stopped; this state persists until a corresponding *Stim* bit is set. The *Stim* bit may be set by a request to the Butler either from another software thread through execution of a `Stim` instruction, by some external event such as an interrupt, or a timer expiring. Once a corresponding pair of *Stim* and *Wait* bits for an activity are both set the activity again becomes ready.

In a preemptive environment, if the priority of the ready activity is higher that that of the one currently executing the Butler support device will deliver an interrupt the processor, the service of which will force a context switch to the higher priority activity. The preempted activity will be made ready and returned to the end of its pollset, i.e. it will be eligible to run again only after all other ready members of its pollset.[1] This is clearly a different behaviour from that required in Ada where the preempted task is placed at the head of its priority queue — however the "normal" Ada behaviour can be obtained if all of the activities are simply placed in different pollsets (equivalent to a unique priority case).

Whilst a preemptive approach has been used successfully on some projects an alternative way of using the Butler is to adopt a cooperative, or non-preemptive, scheduling model; this approach results in "better" processor utilisation — a point which will be returned to later in the paper. When using a cooperative approach the application code is seeded with "calls" which will allow activities to be switched if certain conditions are satisfied. The density of the seeding of the application is dependent on the worst case responsiveness required to any event — the seeding process needs to be driven by experiment and measurement and is also discussed later in the paper.

Each of the seeded calls executes one of two primitive operations: in the first the activity will yield to any activity of higher priority than the current one — `Yield_To_Higher`; or, in the second, it will yield to any of higher or equal priority to the current activity — `Yield`. Generally the former is used as it results in the better utilisation of the processor, and provides a behaviour closer to that of fixed priority scheduling. In cases where the activities have unique priorities the `Yield` and `Yield_To_Higher` primitives are, of course, equivalent.

The activity with the value `Activity'Last`, as has already been mentioned, is always schedulable. In general this is mapped to some "Null" thread which in non-interruptible and simply loops repeatedly performing `Yield_To_Higher` operations.

[1] At the point of preemption it is not possible to know how many activities will run before the preempted one, the only safe assumption is that it will be the number of activities in the pollset minus one.

4 Integration with Ada

Several different integrations of the Butler with Ada have been successfully implemented. These range from a "bare" solution where only a minimal Ada runtime system is used; through an approach where a Ravenscar [12] run-time system is used simply to create the tasks contexts and provide the context switching service; and finally, to one where there is a "complete" integration of the device with a Ravenscar-like runtime. In the latter case the use of tasks and protected objects and protected procedures as interrupt handlers is fully supported; the reason for the runtime being described as Ravenscar-like is due to the task dispatching scheme being a bespoke, cooperative, round-robin one rather than the *FIFO_Within_Priorities* scheme, and the *Priority_Ceiling* approach to locking dictated by the Ravenscar profile. The term cooperative is used here to distinguish it from the non-preemptive approach defined for Ada 2005 where FIFO ordering is still required [11].

Rather than consider all of these integrations in some depth just the two extremes, the minimal runtime and full Ravenscar support, are discussed in more detail in the sections below.

4.1 Minimal Runtime System

Here a very restricted Ada runtime system is provided, the majority of the tasking system is not available, only the basic `delay until` construct is supported; this runtime is used in circumstances where the highest levels of safety integrity are required, for example DO178B level A [13], or DefStan 00-56 [14]. In other areas the runtime is also severely limited with no support for dynamic memory allocation in general and access types in particular — the runtime is derived from the GNAT "zero-footprint" runtime. In this case all of the scheduling services are provided by a support library `(scheduler.kernel)` with a very limited set of interfaces exposed to the application; these interfaces include support for:

- Thread creation;
- Thread wait — for any of a specific set of events;
- Signal event — to a particular thread;
- Yield — allowing any thread, which is ready and whose priority is at or above the current priority to run;
- Yield-to-higher — allowing any thread that is ready and whose priority is above the current priority to run;
- Delay until — absolute time;
- Mutex lock/unlock (only when preemptive dispatching is used).

There is no fine-grained control of priority provided with this approach; if mutual exclusion is required then processor level mutex operations are used — of course in a cooperative environment (the normal use of the solution), where there is no preemption, there is no need for explicit mutual exclusion controls, thus these are only required when used for a preemptive solution.

There is no support for protected operations in this minimal runtime system, posing the question: to what are external events mapped? In this case each external event is mapped to (one or more) Butler activities, and hence to corresponding threads. This might appear to be a rather "heavy-weight" approach but given the generally sporadic nature of the applications this is not the case.

In the absence of protected and suspension objects the inter-thread communications, and cross-stimulation, are encapsulated in a set of generic packages that, using the facilities of the scheduler kernel described earlier, provide a range of possible inter-thread interactions, both blocking and non-blocking. These interactions are the focus for most of the software-software `Wait` and `Stim` scheduling operations.

In the non-preemptive case the basic principle of operation is that all of the threads are created by calls to the scheduler kernel in the main subprogram, which has the lowest possible "priority". In the creation process all of the threads are made ready, once this is complete the main procedure enters an endless loop within which it performs a simple `Yield_To_Higher` operation allowing anything else to be run. This general sequence is shown in the code fragment below for the creation of two threads, each at different priority.

```
procedure Main is
begin
   Initialise_Kernel;  --  Ensures that the Butler is initialised
                       --  and that Activity'Last is the current activity
   Create_Thread (Activity_Ident   => Act_Id_1,
                  Pollset_Boundary => True,
                  Entry_Point      => Thread_1_Main'Access);
   Create_Thread (Activity_Ident   => Act_Id_2,
                  Pollset_Boundary => True,
                  Entry_Point      => Thread_2_Main'Access);

   --  All threads are now created but none have yet run
   --
   loop
      Yield_To_Higher; --  Allows the highest priority thread to
                       --  run, it will only return when there is
                       --  nothing else that is ready to run
   end loop;
end Main;
```

The `Yield` and `Yield_To_Higher` primitives are quite simple and are shown in the code fragments below.

```
procedure Yield is
   Current, Next : Butler.Activity_Id;
begin
   Butler.Suspend;
   Butler.Curract (Current);
   Butler.Next_Activity (Next);
   if Next /= Current then
```

```
      Switch_To (Next);  --  No point in switching to self...
   end if;
end Yield;

procedure Yield_To_Higher is
   Current, Next : Butler.Activity_Id;
begin
   if Butler.AMI then
      Butler.Suspend;
      Butler.Curract (Current);
      Butler.Next_Activity (Next);
      Switch_To (Next);  --  If the priority is higher it cannot be self
   end if;
end Yield_To_Higher;
```

The advantage of this approach is that the interface from the application to the scheduler kernel is thin and hence introduces a very low processing overhead; the disadvantage is that whilst relatively simple to implement in a bare-board target it becomes somewhat more problematic to implement over an existing operating system in either host or target environments; making it difficult to replicate the bare-board scheduling behaviour in these environments.

4.2 Ravenscar-Like Runtime System

At the other extreme of the spectrum from the minimal runtime approach is that where a significant, but not necessarily full, Ada runtime is provided, allowing Ada to be used in a more "traditional" fashion. In this case the runtime used is a specially tailored variant of the GNAT Ada runtime that supported the Ravenscar profile.

At the application level the program comprises a number of tasks which interact through protected objects and suspension objects (as provided by the package `Ada.Synchronous_Task_Control`) — much as they would for any "standard" Ravenscar application. The fact that there is some special scheduler kernel is transparent to the application, as it is only the Ada runtime that is interfaced to the primitive scheduler operations. These operations are largely the same as those for the kernel when using the minimal runtime system described above, but in this case they are presented somewhat differently.

In this case the facilities provided by the scheduler kernel have to be presented in a manner compatible with the requirements of the Ada runtime system.[2] In this case a single "operating system" interface package is provided at the lowest level of the Ada runtime to provide a focus through which to interface to the underlying kernel. The sections below discuss each of the major Ada language (tasking) features and how they are realised using the scheduler kernel.

[2] Clearly the Ada runtime could be reformulated to match the scheduler primitives, but this route was not chosen as it was considered to be more difficult to implement.

Ada Tasks. The Ada tasks are implemented over the threads provided by the scheduler kernel; every task being associated with a unique thread, and each thread in turn being associated with a unique activity provided by the Butler. Thus there is a direct mapping from Ada task to Butler activity, with the priority of the activity (i.e. its pollset) determining the particular activity to be used.

Priority Inheritance. As noted above when a cooperative scheduling approach is used there is no need to provide mutual exclusion for resource access. When used in a preemptive environment the approach to providing mutual exclusion is dependent on the capabilities of the particular variant of Butler. Ideally the device supports the notion of temporary priority elevation allowing a ceiling inheritance protocol to be implemented; however with more primitive versions of the device this feature is not provided and a simple processor-based mutex is used — clearly this is far from ideal as it can lead to excessive blocking as in effect the "ceilings" are set unnecessarily high.

Protected Objects. There are two aspects to protected objects that require support from the underlying kernel: protected entries; and protected procedures attached to "interrupts". In the first case the entry uses a Butler stim-wait pair dedicated to the interface to the Ada runtime system.

The attachment of a protected procedure to an interrupt is effected by associating the procedure with its own particular Butler activity, this is "achieved" through the pragma `Attach_Handler`.

The use of Butler activities to support handlers mapped to protected procedures gives an opportunity to make the `Yield_To_Higher` and `Yield` operations more efficient. If the "core" of the operations shown above are replaced by the code fragment below it allows the invocation of the protected procedure to be reduced to what is, in effect, a procedure call, reserving the need to perform a context switch for the task switching case alone.

```
Suspend;
loop
   Curract (Current);
   Next_Activity (Next);
   exit when Type_Of (Next) = Thread;
   Invoke (Next);  --  Protected operation executed in context
                   --  of the current task
end loop;
Switch_To (Next);  --  New task
```

Where the procedure `Invoke` is:

```
procedure Invoke (Act : Activity_Id) is
begin
   Entry_Points (Act).all;    -- Execute the protected procedure
   Wait (Release_Mask (Act)); -- Make the protected procedure's
                              -- activity ''sleep'' again
end Invoke;
```

Suspension Objects. The `Suspension_Object`, which is provided by the package `Ada.Synchronous_Task_Control`, maps very directly onto the concept of the stim-wait node provided by the Butler — they are very similar concepts.

Delay Until. The `delay until` language construct is supported using the timers of the Butler. Unfortunately these only support relative delays, thus the absolute demands have to be converted (by the kernel) into this form.

Cooperative Scheduling Support Primitives. Clearly if the `Yield` operations are to be used in the application then these need to be provided in some way. There are two approaches that can be followed, either: provide a scheduler interface package that exposes the operations directly; or, use "standard" elements of the Ada language to obtain the desired effects.

The latter approach can only accommodate the yield operation a straightforward manner — this is achieved through the use of a `delay until` to either now, or some time in the past. This has the effect of placing the current task to the back of the ready queue for its priority and allows any ready thread of higher or equal priority to run. The `Yield_To_Higher` primitive is not supported directly; though, if the tasks have unique priorities the effect of `Yield` and `Yield_To_Higher` are identical.

The former approach could be improved if an extension to the runtime were to be provided to incorporate these facilities. In Ada 2005 there is the concept of extending the real-time features through a child package to `Ada.Dispatching`, so a package `Ada.Dispatching.Non_Preemptive` with a specification as shown below would allow the operations to be exposed in a useful manner retaining application independence from the scheduler.

```
package Ada.Dispatching.Non_Preemptive is

   procedure Yield;

   procedure Yield_To_Higher;

end Ada.Dispatching.Non_Preemptive;
```

Such a package has been suggested for inclusion in a future revision of the Ada language [15].

Advantages and disadvantages. Again as with the zero footprint runtime approach there are both advantages and disadvantages. The advantages are clearly that the application only makes use of the Ada tasking features for scheduling and inter-task interactions (notably the discussion is around the task and not the thread), and the mapping of the protected procedure to an activity and its dispatching as a simple procedure call is a very efficient solution. The use of these two features leads to an inherently more portable application.

On the other side of the coin there are two main disadvantages, these are:

– The runtime is bigger and the interactions with the scheduler are, when compared to those with the no runtime option, marginally less efficient;

– The provision of the yield operation is via a rather crude, and possibly obscure approach, and the yield to higher primitive is not provided at all, unless this is through a scheduler interface package; an approach which degrades the aforementioned advantage of portability.

Protected procedure "efficiency" is adversely affected by the lack of fine-grained priority management available in some variant of the hardware, hence this Ravenscar approach is best suited to environments utilising the more sophisticated devices.

5 Some Practical Results and Observations

In terms of results there are two main areas to consider: the effectiveness of the use of the device in place of a traditional software scheduler and hardware interrupt handler; and the impact of a cooperative scheduling approach over a preemptive one. Each of these areas is considered in the sections below. In all these cases the comparisons have to consider large applications, small detailed studies do not generally provide results that scale to bigger systems.

5.1 A Comparison of Hardware-Supported and Software-Only Approaches to Scheduling

The alternative, traditional approach considered here is one with tasks being managed in queues, and hardware events being fielded by relatively simple interrupt controllers with associated software interrupt handlers. Overall a significant improvement in processing performance has been observed across a number of systems where hardware assisted scheduling has been employed.

Providing precise figures for this improvement is not simple as there are a number of interacting aspects, the major two of which are: the lower levels of software overhead in the maintenance of the scheduler data structures; and, the "secondary" resultant impact on the effectiveness of the data and instruction caches. The impact of the latter should not be underestimated as relatively small changes to the caching effectiveness can have a significant impact on the available processing capacity.

In general the saving directly attributable to the hardware support is quite small, in the order of 1% of load; however there are further improvements when the cooperative approach is also used — this is described in the next section.

5.2 A Comparison of the Cooperative and Preemptive Approaches

At the extreme, in a system with no external events (including timers), these two approaches will lead to the same scheduling behaviour. However it would be highly unlikely that any real-time embedded system would exhibit those attributes.

The comparison is not a simple one, the differences between the two approaches are driven by a number of factors, the most significant of which are:

the rate of preemption driven context switches; and the degree to which the density of the yield points is excessive. The first of these is, in the general case, the dominant factor.

Preemption has two significant impacts on a modern processor: it disrupts the instruction pipeline; and causes the cache to be less effective. Whilst the former is a relatively straightforward effect the second is more complex especially where multi-level caches are used.

As an example of the possible savings, on one system changing from the cooperative to a preemptive model decreased the processor loading (as measured relative to the time spent in the "idle" activity) by 5%. This might not appear to be a significant saving but in a system where there are high levels of processor utilisation it can make the difference between the system working or failing, it should also be remembered that this saving is in addition to that achieved by simply having hardware support.

5.3 Issues of Seeding the Application with Cooperative Dispatching Points

Perhaps the biggest impediment to the use of the co-operative approach is the need to seed the application with calls to the `Yield_To_Higher` and `Yield` primitives. The required density of these calls relates to two factors: the speed of the processor, and the required responsiveness to any event. If the density of these calls is excessive then the benefits of the cooperative approach are eroded. Whilst this seeding might seem an onerous task in reality it only becomes problematic when the required responsiveness is very demanding — response times of the order of 30μs (using a relatively slow 133 MHz processor) are easily achieved whilst gaining significant benefits in available processing capacity.

Some experiments using the compiler to automatically implant the cooperative scheduling calls have been made. In general the results were mixed, overall the density was acceptable but there were areas of over- and under-seeding. At this time a simple automated approach does not seem possible, however the development of more sophisticated tools might be a possibility for the future.

6 Opportunities for Further Enhancements

As implementation technologies continue to evolve so too will the Butler type devices. The provision of multiple processor cores inside FPGAs present both a challenge and an opportunity. In the latter case FPGAs can provide a platform on which a more flexible scheduler can be constructed; whilst in the former there is the issue of scheduling applications distributed over many processors. Extensions to the Butler to coordinate the scheduling of tasks across a number of processors is one area that needs to be explored and developed if the technology is to remain relevant.

Leaving aside these more distant future issues there are several areas where, in the short term, the capability of the current scheduler can usefully be extended

whilst retaining the advantages discussed earlier — in large degree these are inspired by the changes introduced in Ada 2005 [11]. There are three of particular interest: the provision of different scheduling policies; the use of the Butler's watchdog activity timers to support the concept of execution budgets; and the development of very low overhead design approaches based on the use of timers and protected procedures. Taking each of these in turn.

Support of additional scheduling policies. Clearly the support for a round-robin scheduling regime is rather limited if a more deterministic ordering is desired. Whilst the original hardware could only support this approach the more recent "softer" variants of the Butler have been adapted to support at least a FIFO-within-priorities approach. Equally the current lack of adherence to the Ada preemption semantics can also be corrected in these later variants.

Beyond this, support for an EDF mechanism is an interesting option as, if it can be implemented in an efficient manner, it should allow for better utilisation of the processors. This could be extremely useful given the levels of processor utilisation are already very high.

Execution time budgets. As noted earlier the Butler has a watchdog timer associated with the execution of each activity and this can trigger an interrupt if there is an overrun. This naturally maps on to the concept of execution time clocks provided in the package `Ada.Execution_Time.Timers`.

A low overhead design approach. Ada 2005 introduces the concept of protected procedures being executed at some particular time, this is provided by the package `Ada.Real_Time.Timing_Events`. These protected procedures can be mapped onto the concept of a Butler activity that is controlled by some hardware timer. How useful this approach would be for real application needs more exploration, but combining these timers with protected procedures for other external events, and using a cooperative approach to scheduling could allow for a very low overhead solution as it could completely eliminate the need for context switches, and allow simplification of the required runtime system.

7 Conclusions

The use of a hardware device to support the role of the scheduler has been shown to both be easy to integrate with an Ada runtime system, and reduce the overheads associated with task scheduling and interrupt handling. Significantly in its practical application reported here the greatest benefits are realised when a cooperative scheduling scheme is adopted.

The examples of use presented in this paper have focused on the issues in the context of no more than a Ravenscar profile, and in this case there is a high degree of compatibility. Attempting to extend the scope beyond Ravenscar to support more fully the Ada tasking model in general does not appear to be particularly straightforward, however this is not really a significant issue as the device was always designed with small platforms and high integrity applications in mind — neither of which are compatible with the full Ada tasking model.

References

1. He, H., Guo, H.: The realization of FFT algorithm based on FPGA co-processor. In: IITA 2008: Proceedings of the 2008 Second International Symposium on Intelligent Information Technology Application, Washington, DC, USA, pp. 239–243. IEEE Computer Society, Los Alamitos (2008)
2. Palmer, J.: The Intel®8087 numeric data processor. In: ISCA 1980: Proceedings of the 7th annual symposium on Computer Architecture, pp. 174–181. ACM, New York (1980)
3. Burns, A., Wellings, A.: Real-Time systems and Programming languages, 3rd edn. Addison-Wesley, Reading (2001)
4. Nácul, A.C., Regazzoni, F., Lajolo, M.: Hardware scheduling support in smp architectures. In: DATE 2007: Proceedings of the conference on Design, automation and test in Europe, San Jose, CA, USA, pp. 642–647. EDA Consortium (2007)
5. Nakano, T., Utama, A., Itabashi, M., Shiomi, A., Imai, M.: Hardware implementation of a real-time operating system. In: TRON 1995: Proceedings of the The 12th TRON Project International Symposium, Washington, DC, USA, pp. 34–42. IEEE Computer Society Press, Los Alamitos (1995)
6. Lai, B.C.C., Schaumont, P., Verbauwhede, I.: A light-weight cooperative multithreading with hardware supported thread-management on an embedded multiprocessor system. In: Proceedings of Asilomar Conference on Signals, Systems, and Computers (2005)
7. Roos, J.: A real-time support processor for ada tasking. SIGARCH Comput. Archit. News 17(2), 162–171 (1989)
8. Runner, D., Warshawsky, E.: Synthesizing Ada's ideal machine mate. VLSI Systems Design, 30–39 (October 1988)
9. Ardö, A.: Hardware support for efficient execution of ada tasking. In: Proceedings of the Twenty-First Annual Hawaii International Conference on Architecture Track, pp. 194–202. IEEE Computer Society Press, Los Alamitos (1988)
10. Campbell, E.R., Simpson, H.R.: Integrated circuits for multi-tasking support in single or multiple processor networks. Patent 6971099 (November 2005)
11. Taft, S.T., Duff, R.A., Brukardt, R.L., Ploedereder, E., Leroy, P. (eds.): Ada 2005 Reference Manual. Language and Standard Libraries, International Standard ISO/IEC 8652:1995(E) with Technical Corrigendum 1 and Amendment 1. Springer (2006)
12. ISO/WG9 Ada Rapporteur Group: Ravenscar profile for high integrity systems. Technical report (2007)
13. RTCA: Software considerations in airborne systems and equipment certification. Radio Technical Commission for Aeronautics (RTCA), European Organization for Civil Aviation Electronics (EUROCAE), DO178-B (1992)
14. UK Ministry of Defence: Defence standard 00-56: Safety management requirements for defence systems – issue 4. Technical report (2007)
15. AI05-0166-1: Yield for non-preemptive dispatching (2009)