Deadline-Aware Programming and Scheduling

Alan Burns and Andy Wellings

Department of Computer Science,
University of York, UK.
emails: alan.burns/andy.wellings@york.ac.uk

Abstract. Deadlines are the most important events in real-time systems. Real-time programs must therefore be aware of deadlines, and be able to identify and react to missed deadlines. Moreover, Earliest Deadline First (EDF) is the most widely studied optimal dynamic scheduling algorithm for uniprocessor real-time systems. In this paper we explore how a resource sharing protocol (called the DFP – Deadline Floor inheritance Protocol), which has been proposed for languages such as Ada, can be incorporated into the language's definition. We also address the programming of systems that have mixed scheduling (e.g. fixed priority and EDF). The incorporation of the DFP into Ada requires some changes to the current predefined packages. These changes are also of use in supporting the programming of deadline-aware systems even when not scheduling by EDF.

1 Introduction

The correctness of an embedded real-time system depends not only on the system's outputs but also on the time at which these outputs are produced. The completion of a request after its timing deadline is considered to be of degraded (potentially no) value, and could even lead to a failure of the whole system. Therefore, the most important characteristic of real-time systems is that they have strict timing requirements that must be guaranteed and satisfied. Schedulability analysis plays a crucial role in enabling these guarantees to be provided.

The *Earliest Deadline First* (EDF) algorithm is one of the most widely studied dynamic priority scheduling policies for real-time systems. It has been proved [18] to be optimal among all scheduling algorithms for a uniprocessor; in the sense that if a real-time task set cannot be scheduled by EDF, then it cannot be scheduled by any other algorithm.

The Ada 2005 standard [20] introduced EDF as one of the supported dispatching policies and the *Stack Resource Policy* (SRP) was specified as the protocol for resource sharing among EDF tasks [11]. SRP is a complex protocol, as has been shown by the initial difficulties with its specification and implementation (see Section 3). Recently, a new protocol for resource sharing in EDF has been proposed [7, 9]. This new protocol, called the *Deadline Floor inheritance Protocol* (DFP), is simpler to understand and more efficient to implement (while keeping all the useful properties of SRP). At the 16th International Real-Time Ada Workshop, the recommendation was agreed [22] that SRP should be deprecated and be replaced by the DFP for single processor systems. In this paper we further explore the motivation for this change, and consider in more detail the impact on the language's definition.

We review, in section 3, the current provision of Ada in its support for EDF and SRP. The new protocol is described in Section 4, and the changes needed to incorporate this definition into the language are outlined in Section 5. Coverage of systems that have both fixed priority and EDF scheduling is given in Section 6. In Section 7 the impact of the proposed changes are considered in the context of Ada's approach to programming real-time abstractions. Then, in Section 8 we briefly consider the implications of multiprocessor systems on the protocol. Conclusions are contained in Section 9. First, however, we give a short definition of a system model.

2 System Model

Although not restricting ourselves to the Ravenscar profile [8], the system model assumed in this paper has many similarities to that profile. A system is assumed to consist of N tasks; all of which are defined to have a period (denoted by the symbol T) that is their minimum inter-arrival time, a relative deadline (D) and a worst-case execution time (C). All tasks are executed on a single processor (or are statically partitioned onto a set of processors). For system correctness, any task τ_i arriving at time t must be able to execute for its maximum computation time (C_i) by its deadline which is at time $t + D_i$.

With fixed priority scheduling, each task is assigned a priority (P), and all protected objects are assigned ceiling priorities and a priority ceiling protocol (PCP) [19] is implemented. The form of PCP usually applied is the 'immediate' version (IPCP) in which a task's priority is raised to the resources's ceiling at the point the resource is accessed. For optimal schedulability, a task's priorities is derived from its relative deadline. Two tasks with relative deadlines D_i and D_j, with $D_i < D_j$ will have priorities with the constraint $P_i > P_j$.

The term 'deadline' can be overloaded in scheduling papers. Here we explicitly use *relative* deadline when concerned with the task's D parameter. The use of *deadline* on its own refers to the absolute deadline of the current invocation of the task. It could be argued that Ada fails to fully support deadline-aware programming as relative deadlines for tasks cannot currently be directly represented in the program's code using language defined types and subprograms.

3 Earliest Deadline First Dispatching and SRP in Ada

Baker [3, 2] proposed the Stack Resource Policy (SRP) for bounding priority inversion when accessing resources in real-time systems scheduled under EDF. SRP is a generalisation of IPCP.

With SRP, each task is assigned a number called the *preemption level* that correlates inversely to its relative deadline: the shorter the relative deadline the higher the preemption level. Shared resources are also assigned a preemption level that is the highest of the preemption levels of all the tasks that may use that resource. The use of SRP imposes a new rule to basic EDF scheduling: *a task can only be chosen for execution*

if its preemption level is strictly higher than the preemption levels of the resources currently locked in the system. The basic rule is, of course, *a task can only be chosen for execution if it has the earliest deadline*.

The most complex part of the EDF dispatching definition in Ada is the integration of the base Ada dispatching model (based on fixed priorities for the tasks and priority ceilings for the protected objects) with the SRP rules and the *preemption level* concept. EDF is defined to work in a given band of priority levels, which may cover the whole range of system priorities, or a specific sub-interval of these priorities. The Ada Reference Manual (ARM) defines a means of integrating preemption levels and priorities: preemption levels of tasks and protected objects are mapped to priorities in the EDF priority band.

The ARM defines that, by default, the active priority of an EDF task is the lowest priority in its EDF priority band. The task will inherit priorities as any other Ada task; in particular, when an EDF task executes a protected operation it will inherit the priority (preemption level) of the protected object. But, for EDF tasks, the ARM defines a further source of priority inheritance (for arbitrary task T): *the highest priority P, if any, less than the base priority of T such that one or more tasks are executing within a protected object with ceiling priority P and task T has an earlier deadline than all such tasks; and furthermore T has an earlier deadline than all other tasks on ready queues with priorities in the given EDF_Across_Priorities range that are strictly less than P*.

This rule has proved difficult to specify correctly (the original definition was shown to be incorrect [24]) and to implement correctly [15] or efficiently [1].

There is one further drawback that is specific to the Ada definition of SRP: the limited number of distinct preemption levels. The number of distinct preemption levels that can be used for tasks in an EDF priority range is the size of the range minus one. In a system with few priority levels or in a narrow EDF range this limitation could jeopardize the schedulability of the system by causing more blocking than is necessary. It could be argued that the implementation should provide more priority levels, but priority levels are expensive because they affect the size and performance of many of the run-time data structures such as the delay queues, ready queues and the entry queues.

4 The Deadline Floor Protocol

Recently, Burns introduced a new protocol for resource sharing in EDF, called the Deadline Floor inheritance Protocol (DFP) [7]. The DFP has all the key properties of SRP; specifically, causing at most a single blocking effect from any task with a longer relative deadline, which leads to the same worst-case blocking in both protocols. In an EDF-scheduled system, the DFP is structurally equivalent to IPCP in a system scheduled under fixed priorities.

Under the DFP, every resource has a relative deadline equal to the shortest relative deadline of any task that uses it. The relative deadline of a resource is called *deadline floor*, making clear the symmetry with the *priority ceiling* defined for the resources in any PCP.

The key idea of the DFP is that the absolute deadline of a task could be temporarily shortened while accessing a resource. Given a task with absolute deadline d that ac-

cesses a resource with deadline floor D^F at time t, the absolute deadline of the task is (potentially) reduced according to $d := min(d, t + D^F)$ while holding the resource.

To give a concrete example, assume two tasks (A and B) with relative deadlines of 20 and 30 share a resource. The deadline floor of the resource is therefore 20. Assume task B is released at time 100: its absolute deadline is therefore 130. At time 103 it accesses the resource: its absolute deadline is therefore reduced to 123. While B holds the resource, task A is released at time 105: its absolute deadline is 125, this is not sufficient to preempt B (as $125 > 123$). If B exits the resource at time 107, its absolute deadline will change from 123 to 130; as a result A will preempt as it now has the earlier deadline (i.e. $125 < 130$).

The action of the protocol results in a single block per task, deadlock free execution and works for nested resource usage. Whilst a task accesses a resource its deadline is reduced so that no newly released task can preempt it and then access the resource. See [7] for details and proof of the key properties. This is equivalent to the use of a priority ceiling; again the only tasks that can preempt a task executing within a protected object are tasks that are guaranteed not to use that object (unless there is a program error, which can be caught at run-time).

The DFP does not add any new rule to the EDF scheduling, thus it leads to simpler and more efficient implementation than the SRP [1].

5 Required Language Simplifications and Modifications

To embed the rules for the DFP within Ada, the following issues must be addressed:

- All tasks must have a relative deadline assigned via an aspect/pragma or a routine defined in a library package.
- Protected objects must have also a relative deadline (floor) assigned via an aspect/pragma.
- Default relative deadline values must be defined for tasks and protected objects (and their types).
- Rules for EDF scheduling must be extended to include a new locking policy: `Floor_Locking`.
- Rules for EDF scheduling need simplifying to remove the 'across priorities' feature of the current definition.
- For completeness (and parity with priority ceilings) means of modifying the relative deadline attribute of tasks and protected objects should be defined.

First, however, some changes to library packages are needed to make the notion of deadline (relative and absolute) first class within the tasking model. Currently, relevant definitions are coupled to the specification of EDF scheduling. Whilst deadlines are key to EDF scheduling, they have a wider purpose; deadlines are relevant to all forms of real-time scheduling. Moreover, programs that wish to catch and respond to missed deadlines need to be able to manipulate deadlines directly.

5.1 Changes to Existing Library Packages

The 2005 version of Ada introduced EDF scheduling and the subtype `Deadline`. Unfortunately, we feel, it only introduced this, as we noted above, for the support of EDF scheduling. We feel that *deadline* and *relative deadline* are fundamental concepts in real-time and deadline-aware programming. We therefore propose that the whole package `Ada.Dispatching.EDF` be renamed, repositioned and extended to support relative as well as absolute deadlines. The new package could be as follows.

```
with Ada.Real_Time;
with Ada.Task_Identification;
use Ada;
package Ada.Deadlines is
  subtype Deadline is Real_Time.Time;
  subtype Relative_Deadline is Real_Time.Time_Span;
  Default_Deadline : constant Deadline :=
              Real_Time.Time_Last;
  Default_Relative_Deadline : constant Relative_Deadline :=
              Real_Time.Time_Span_Last;
  procedure Set_Deadline(D : in Deadline;
            T : in Task_Identification.Task_ID :=
            Task_Identification.Current_Task);
  function Get_Deadline(T : in Task_Identification.Task_ID :=
           Task_Identification.Current_Task) return Deadline;
  procedure Set_Relative_Deadline(R : in Relative_Deadline;
            T : in Task_Identification.Task_ID :=
            Task_Identification.Current_Task);
  function Get_Relative_Deadline(T : in Task_Identification.Task_ID :=
                                 Task_Identification.Current_Task)
                                 return Relative_Deadline;
   procedure Delay_Until_And_Set_Deadline(
            Delay_Until_Time : in Real_Time.Time;
            TS : in Real_Time.Time_Span :=
            Get_Relative_Deadline);
end Ada.Deadlines;
```

Key changes are:

- Change of name and library position.
- Introduction of a type for relative deadline and a default value.
- Set and Get routines added for relative deadlines.
- A default relative deadline provided for `Delay_Until_And_Set_Deadline`.

All tasks will have a deadline and a relative deadline; default values being used if the program does not specify specific values. As with priority, where a task has a base and an active priority, a task will also have a base (absolute) deadline and an active (absolute) deadline – see definition of the locking policy below. A call of `Get_Deadline` returns the base deadline of the task.

The existing aspect/pragma `Relative_Deadline` should be redefined to take an expression of type `Relative_Deadline`. Note, although the same name is used here, this is the same situation with subtype `Priority` and aspect/pragms `Priority`. However, the definition of the aspect `Relative_Deadline` should really be moved

from D.2.6. We suggest that it be placed, with the above package, in a new section D.8.1. perhaps entitled *Deadline-Aware Programming*.

5.2 New Locking Policy

Initially, when EDF was added to Ada, the existing locking policy `Ceiling_Locking` was modified so that is accounted for EDF dispatching, FP (fixed priority) dispatching and combined EDF and FP dispatching. Although there are some clear advantages in having only a single protocol, it is now considered to have been a mistake [22], due to the complex rules required. Here we propose a new locking policy `Floor_Locking`. We will not attempt to give here a full definition sufficient for the ARM, but the following points define the semantics for this new policy.

- Whenever a task is executing outside a protected action, its active deadline is equal to its base deadline.
- When a task executes a protected action its active deadline will be reduced to (if it is currently greater than) 'now' plus the deadline floor of the corresponding protected object.
- When a task completes a protected action its active deadline returns to the value it had on entry.
- When a task calls a protected operation, a check is made that there is no task currently executing within the corresponding protected object; `Program_Error` is raised if this check fails.

A protected object is given an initial deadline floor value using the `Relative_Deadline` aspect/pragma. Dynamic deadline floors could be defined in a similar way to dynamic ceiling priorities (see Section D.5.2 of the ARM). We do not consider this here.

With this definition of a new locking policy, the definition of `Ceiling_Locking` can return to its pre-2005 wording.

Note the semantics requires a check on non-concurrent access to the protected object. It is not sufficient to check that the relative deadline of the task is not less than the deadline floor of the object. This points to a difference with `Ceiling_Locking` where a comparison based on priorities is sufficient. To implement the check on inappropriate usage over the corresponding protected object requires only a simple 'occupied' flag to be checked and modified. Usefully, if there is an attempt to gain access to an occupied protected object then the task 'at fault' is forced, on a single processor, to be the second task that is attempting to gain access, and it will therefore be this task that has the exception raised. The correct task will be unaffected.

Interestingly, a simple check on non-concurrent access would also be sufficient for the priority ceiling case. And again the exception is bound to be raised in the task 'at fault'. Of course, checking concurrent access, rather than correct priority/ceiling values will only catch an actual error rather than a potential one. Inappropriate ceiling values will be caught on first usage, inappropriate concurrent access may be very difficult to create during testing. Although not a sufficient test, it might be advisable to also include in the definition of `Floor_Locking` a static check on the relative deadlines of user tasks and the deadline floors of the used protected objects.

To ensure that locking protocols work correctly, the programmer must give the correct values for deadline floors and ceiling priorities. A run-time check prevents concurrent access, but a compiler-based check cannot be undertaken and hence the use of the correct values can only be asserted by code inspection or static analysis.

5.3 New Dispatching Policy

Currently EDF dispatching is supported via the policy `EDF_Across_Priorities`. A range of priorities is needed to account for the different priority ceilings needed for the protected objects. The tasks themselves only execute at the base priority of this range when they are not executing within a protected action. All ready queues are ordered by the (absolute) deadline of the ready tasks.

To prevent confusion, and to emphasis the fact that with the new protocol only a single priority is needed for all EDF dispatched tasks (regardless of the number of protected objects they use), we propose a new dispatching policy. And to accommodate hierarchical dispatching (see Section 6) we define the new policy as `EDF_Within_Priorities`. Again we will not attempt to give a full definition appropriate for the ARM[1].

With `EDF_Within_Priorities`, all tasks with the same priority compete for the processor using the rules for EDF dispatching. The ready queue is ordered by *active* deadline. A collection of EDF dispatched tasks and the set of protected objects they use/share will all have the same priority (and ceiling priority). But they will have different relative deadlines (and deadline floors).

A task that has not been given an explicit deadline or relative deadline will get the default values of `Default_Deadline` (equal to : `Real_Time.Time_Last`) and `Default_Relative_Deadline` (equal to `Real_Time.Time_Span_Last`). The default value for the deadline floor of any protected object is 0 (actually `Time_Span_Zero`). This will have the effect of making all protected actions non-preemptive (as does the default priority ceiling).

5.4 Ravenscar-like Profile

The facilities provided by the policies `EDF_Across_Priorities` and `Floor_Locking`, and the library package `Ada.Dispatching.EDF` allows a Ravenscar-like profile for EDF scheduling to be defined.

For periodic (time-triggered) tasks the profile would only allow a task to be delayed by the use of `Delay_Until_And_Set_Deadline` using the default parameter for relative delay (which is the task's relative delay). A task would be forced to set its relative deadline using an aspect/pragma and use the default parameter for the above delay statement. It would be unable to use the `Set_Deadline` and `Set_Relative_Deadline` routines.

[1] For example, consideration would need to be given to whether deadline inheritance should occur during a rendezvous and task activation, and whether entry queues can be deadline ordered.

For sporadic tasks, which are typically released by the action of an interrupt handler, `Set_Deadline` would need to be used, but could be restricted to be allowed only within that context.

To accomplish these restrictions is may be useful to partition `Dispatching.EDF` into the part needed for a restricted profile, and that which is available to all programs.

6 Hierarchical and Mixed Scheduling

One of the advantages of the new `EDF_Within_Priorities` policy is that it unifies Ada's use of priority as the primary dispatching policy. It is no longer necessary to reserve a range of priorities for a single EDF domain. If we ignore the non-preemptive policy, we now have a clear means of supporting mixed scheduling in a hierarchical manner:

- At all times, the task at the head of the highest priority non-empty ready queue is the one chosen to be executed.
- Each ready queue has its own discipline to determine which task is at its head.

The disciplines supported are: FIFO, Round Robin (RR) and now EDF; i.e. `FIFO_Within_Priorities`, `Round_Robin_Within_Priorities` and now `EDF_Within_Priorities`.

So, for example, one could have the top 16 priority levels reserved for pure FP tasks, then the next level for EDF and next (lowest) priority level for RR. At the EDF and RR levels there may be many task allocated. For the FP there many be few, perhaps only one task per priority.

If two priority levels are designated EDF then tasks from the higher priority level will always run in preference to tasks at the lower level (irrespective of deadlines). Only if the ready queue at the higher priority is empty will the task with the shortest active deadline from the lower ready queue be chosen for execution.

To allow tasks from any priority level to share their use of protected objects (POs) it is necessary to ensure the locking policies are appropriately defined. First the fundamental priority based dispatching policy must be supported by `Ceiling_Locking`. If two tasks of different priority use the same PO then the ceiling priority of the PO must be no lower that the highest priority of the client tasks. Within an EDF ordered priority level, the policy `Floor_Locking` must apply. It follows therefore that when hierarchical dispatching is used both `Ceiling_Locking` and `Floor_Locking` will need to be specified. The current definition of `Ceiling_Locking` must be changed to reflect this.

To further illustrate the behaviour of a mixed dispatching scheme, consider two situations on a single processor system. First, assume the highest priorities are reserved for FIFO (FP) and a single EDF ready queue is at a lower priority. Let an EDF task, τ_e execute and call a PO used by a FP task, τ_f. The following points appertain.

- The priority of τ_f is higher than τ_e.
- If τ_e is executing then τ_f must be suspended.

- When τ_e calls the PO its active priority will be set to the priority of τ_f. Its active deadline may also shorten, but this is irrelevant in this example.
- If τ_f is released, it will not execute (its base priority is not greater than the active priority of τ_e and dispatching at this priority level is FIFO).
- When τ_e leaves the PO its priority will return to its original base level, and if τ_f had been released it would now preempt.

For a second example, consider the EDF tasks at the highest priority level (P_{high}) and a set of FP tasks below. The priority ceiling of the PO will be P_{high}. If an EDF task calls the PO, its active priority will not change but its active deadline may. Now consider the lower priority FP task τ_f accessing the PO:

- Priority of τ_e is higher than τ_f.
- If τ_f is executing then τ_e must be suspended (and no other EDF task will be active).
- When τ_f calls the PO its active priority will be set above its current level to P_{high}; its active deadline will be updated according to the DFP; no further FP tasks will run.
- If τ_e is released it will not execute (its base priority is not greater than the active priority of P_{high} and, because of the DFP it cannot have an earlier absolute deadline).
- When τ_f leaves the PO, its priority will return to its original base level (its base deadline will return to its default value), and if τ_e had been released it would now preempt.

It follows from these examples that *both* `Ceiling_Locking` and `Floor_Locking` are needed, but they are not needed together. If a task calls a PO with a higher priority then the ceiling policy applies. And if an EDF dispatched task calls a PO with the same priority then the deadline floor policy applies. Note, of course, that a task cannot call a PO with a lower priority as this would break the priority ceiling protocol.

7 Impact on Real-Time Programming Abstractions

Since its inception, Ada has supported real-time systems' development. Its focus has been on a set of low-level primitive programming mechanisms and support for real-time dispatching policies. The low-level mechanisms (such as the "delay until" statement, the asynchronous select statement, timing events, protected objects etc) have allowed a wide range of real-time programming abstractions to be developed [23]. However, almost paradoxically, up until Ada 2005 the notion of absolute deadline was not explicit in the language. Even at Ada 2005, deadlines were only introduced to support EDF scheduling.

We have shown earlier, that to facilitate integration of the DFP, the notions of absolute and relative deadlines must become more general language concepts that are not confined only to EDF scheduling. Of course, this is correct as even tasks that are being scheduled FIFO within a priority level may have a deadline. Although, this has no effect on the dispatching, the program may need to undertake corrective actions if a task misses its deadline. There are many possible actions (see, for example, Chapter 13 in

Burns and Wellings [12]). Below, we illustrate one task template that can be used to illustrate the impact of having deadlines more explicit in the language. The template is for a periodic task that aborts its current release if the deadline is missed, executes some handling code and then waits for its next periodic release. Note the use of the default relative deadline in the `Delay_Until_And_Set_Deadline` statement.

```
with Ada.Real_Time; use Ada.Real_Time;
with Ada.Deadlines; use Ada.Deadlines;
...

  task type Periodic_Task(Period_In_Milliseconds : Positive;
                    Rel_Deadline_In_Milliseconds : Positive);

  task body Periodic_Task is
    Interval : Time_Span := Milliseconds(Period_In_Milliseconds);
    Rel_Deadline : Time_Span :=
                           Milliseconds(Rel_Deadline_In_Milliseconds);
    Next_Release_Time : Time;
  begin
    Set_Relative_Deadline(Rel_Deadline);
    Next_Release_Time := Clock;
    Set_Deadline(Next_Release_Time + Rel_Deadline);
    loop
      select
        delay until Get_Deadline;
        -- handle deadline miss here
      then abort
        -- undertake the work of the task
      end select;
      Next_Release_Time := Next_Release_Time + Interval;
      Delay_Until_And_Set_Deadline(Next_Release_Time);
    end loop;
  end Periodic_Task;

...
```

Although there is not a significant difference between this and the original template, the programmer's intention is clearer.

8 Multiprocessor Considerations

Ada 2012 [21] supports the notion of a dispatching domain, which consists of one or more processors on which tasks can be globally scheduled. Each processor in the system can only exists in one dispatching domain. A task can only be allocated to a single dispatching domain. In the case where a task is allocated to a multiprocessor dispatching domain, there is an option to fix that task to only be dispatched on a single processor in that domain. Hence, Ada 2012 has the flexibility to support global, partitioned and semi-partitioned systems, as well as algorithms that fix computational intensive tasks to a single processor and schedule less computational intensive tasks globally around them [13].

Dispatching domains in conjunction with the two-level dispatching model in those domains gives the system developer a significant level of control over how tasks are allocated and scheduled in multiprocessor (and multicore) systems.

Although there has been some success in determining the necessary support for scheduling, the issue of how best to support multiprocessor lock-based resource control protocols is still far from clear. New results are emerging [16, 4, 5, 14, 6, 10], but it is too soon for programming languages/operating systems to adopt a particular approach [17].

This paper has discussed the Deadline Floor Protocol in the context of single processor systems. Just like the Stack Resource Protocol, the desirable properties (mentioned in Section 4) are not maintained when protected objects can be simultaneously accessed from multiprocessors. For example, multiple blocks per task and deadlocks are possible. Furthermore, mutual exclusion is not guaranteed by the protocol itself; hence a lock is required, along with a FIFO spinning-based access mechanism. Until more optimal solutions become available, this approach, in conjunction with the default ceiling and default floor, will ensure that a protected action is implemented non-preemptively with predictable blocking times. The programmer will have to ensure that any nested accesses do not lead to deadlock.

9 Conclusions

Arguably since its inception, Ada has supported a two-level dispatching model. Initially, in Ada 83, this was preemptive priority-based scheduling at the top level, and FIFO at the low level (that is, within a priority). Progressively, over the years, the language has added support for non-preemptive priority-based scheduling (at the top level) and round-robin and EDF (within priority levels). Unfortunately, the introduction of EDF scheduling in Ada 2005 required the support of the Stack Resource Policy and its integration into the two-level scheduling scheme. This corrupted the pure two-level scheduling model and required a range of priorities to support a single EDF secondary dispatching level. Since Ada 2005, better understanding of EDF scheduling has been obtained. One result is a new resource control protocol, the DFP. The integration of this protocol into the Ada dispatching model allows a return to the pure two-level model. With this integration, Ada can support

- preemptive or non-preemptive priority-based dispatching at the top level, and
- a mixture of FIFO, RR or EDF dispatching at the secondary level.

The integration requires that deadlines become a more widely visible concept in the language's definition. This has advantages as even for priority-based systems, the need to recover from deadline misses requires deadlines to be set and manipulated. Having deadlines directly expressible in the language makes deadline-aware programming and scheduling more visible and hence more maintainable.

Acknowledgements

The authors would like to thank Marina Gutierrez, Mario Aldea and Michael González Harbour for useful discussions on the implementation of the DFP.

References

1. M. Aldea, A. Burns, M. Gutirrez, and M. G. Harbour. Incorporating the deadline floor protocol in Ada. *ACM SIGAda Ada Letters – Proc. of IRTAW 16*, XXXIII(2):49–58, 2013.
2. T. Baker. A stack-based resource allocation policy for realtime processes. In *Proc. IEEE Real-Time Systems Symposium (RTSS)*, pages 191–200, 1990.
3. T. Baker. Stack-based scheduling of realtime processes. *Journal of Real-Time Systems*, 3(1), March 1991.
4. A. Block, H. Leontyev, B. B. Brandenburg, and J. H. Anderson. A flexible real-time locking protocol for multiprocessors. In *13th International Conference on Embedded and Real-Time Computing Systems and Applications*, RTCSA '07, pages 47–56. IEEE Computer Society, 2007.
5. B. Brandenburg and J. Anderson. Optimality results for multiprocessor real-time locking. In *Real-Time Systems Symposium (RTSS)*, pages 49–60, 2010.
6. B. Brandenburg and J. Anderson. Real-time resource sharing under cluster scheduling. In *Proc. EMSOFT*. ACM Press, 2011.
7. A. Burns. A Deadline-Floor Inheritance Protocol for EDF Scheduled Real-Time Systems with Resource Sharing. Technical Report YCS-2012-476, Department of Computer Science, University of York, UK, 2012.
8. A. Burns, B. Dobbing, and G. Romanski. The Ravenscar tasking profile for high integrity real-time programs. In *Reliable Software Technologies, Proc. of the Ada Europe Conference, Uppsala*, pages 263 – 275. Springer Verlag, 1998.
9. A. Burns, M. Gutirrez, M. Aldea, and M. G. Harbour. A Deadline-Floor Inheritance Protocol for EDF Scheduled Embedded Real-Time Systems with Resource Sharing. *IEEE Transaction on Computers, available online*, 2014.
10. A. Burns and A. Wellings. A schedulability compatible multiprocessor resource sharing protocol - MrsP. In *Proceedings of ECRTS*, pages 282–291, 2013.
11. A. Burns, A. Wellings, and T. Taft. Supporting deadlines and EDF scheduling in Ada. In *Reliable Software Technologies, Proc. of the Ada Europe Conference*, pages 156–165. Springer Verlag, LNCS 3063, 2004.
12. A. Burns and A. J. Wellings. *Real-Time Systems and Programming Languages*. Addison Wesley Longman, 4th edition, 2009.
13. R. Davis and A. Burns. A survey of hard real-time scheduling for multiprocessor systems. *ACM Computing Surveys*, 43(4):35:1 –35:44, 2011.
14. D. Faggioli, G. Lipari, and T. Cucinotta. The multiprocessor bandwidth inheritance protocol. In *Proc. of the 22nd Euromicro Conference on Real-Time Systems (ECRTS)*, pages 90–99, 2010.
15. M. Fairbairn and A. Burns. Implementing and validating EDF preemption-level resource control. In M. Brorsson and L. Pinho, editors, *Proc. of Reliable Software Technologies - Ada-Europe 2009*, volume LNCS 7308, pages 193–206. Springer, 2012.
16. P. Gai, G. Lipari, and M. Di Natale. Minimizing memory utilization of real-time task sets in single and multi-processor systems-on-a-chip. In *Proc. 22nd RTSS*, pages 73–83, 2001.
17. S. Lin, A. Burns, and A. Wellings. Supporting lock-based multiprocessor resource sharing protocols in real-time programming languages. *Concurrency and Computation: Practice and Experience*, 2012.
18. C. Liu and J. Layland. Scheduling algorithms for multiprogramming in a hard real-time environment. *JACM*, 20(1):46–61, 1973.
19. L. Sha, R. Rajkumar, and J. Lehoczky. Priority inheritance protocols: An approach to real-time synchronisation. *IEEE Transactions on Computers*, 39(9):1175–1185, 1990.

20. S. Taft, R. Duff, R. Brukardt, E. Ploedereder, and P. L. (Eds). Ada 2005 Reference Manual, ISO/IEC 8652/1995 (E) with Technical Corrigendum 1 and Amendment 1. Technical report, ISO, available Springer LNCS 4348, 2005.
21. S. Taft, R. Duff, R. Brukardt, E. Ploedereder, P. Leroy, and E. S. (Eds). Ada 2012 Reference Manual, ISO/IEC 8652/2012 (E). Technical report, ISO, available Springer LNCS 8339, 2012.
22. A. Wellings. Session summary: Locking protocols. *ACM SIGAda Ada Letters, Proc. of IRTAW 16*, XXXIII(2):123–125, 2013.
23. A. J. Wellings and A. Burns. Real-time utilities for Ada 2005. *Reliable Software Technologies - Ada Europe 2007, Lecture Notes in Computer Science*, 4498:1–14, 2007.
24. A. Zerzelidis, A. Burns, and A. Wellings. Correcting the EDF protocol in Ada 2005. In *Proc. of IRTAW 13, Ada Letters, XXVII(2)*, pages 18–22, 2007.