A Deadline-Floor Inheritance Protocol for EDF Scheduled Embedded Real-Time Systems with Resource Sharing

A. Burns
Department of Computer Science,
University of York, UK.

Departmental Technical Report: YCS-2012-476

Abstract

Earliest Deadline First (EDF) is the most widely studied optimal dynamic scheduling algorithm for uniprocessor real-time systems. For realistic programs, tasks must be allowed to exchange data and use other forms of resources that must be accessed under mutual exclusion. With EDF scheduled systems, access to such resources is usually controlled by the use of Baker's Stack Resource Protocol (SRP). In this paper we propose an alternative scheme based on deadline inheritance. Shared resources are assigned a relative deadline equal to the minimum (floor) of the relative deadlines of all tasks that use the resource. On entry to the resource a task's relative deadline (and hence its current absolute deadline) is immediately reduced to reflect the resource's deadline floor. On exit the original deadline for the task is restored. We show that the worst-case behaviour of the new protocol (termed DFP – Deadline Floor inheritance Protocol) is the same as SRP. Indeed it leads to the same blocking term in the scheduling analysis. We argue that the new scheme is however more intuitive and easier and more efficient to implement.

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 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.

A real-time system comprises a set of real-time tasks; each task consists of a potentially unbounded stream of jobs. The task set can be scheduled by a number of policies including dynamic priority or fixed priority algorithms. The success of a real-time system depends on whether all jobs of all the tasks can be guaranteed to complete their executions before their timing deadlines. If they can then we say the task set is *schedulable*.

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 [13] 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.

Although many forms of analysis (including that reported in the above citation) assume tasks are independent of each other, in realistic systems the tasks need to make use of shared *resources* that must be accessed under mutual exclusion. These resources are typically protected by semaphores or mutexes provided by an RTOS (real-time operating system). If a high-priority task is suspended waiting for a lower-priority task to complete its use of a non-preemptable resource, then priority inversion occurs [15]. The task is said to be *blocked* by the lower priority task.

For uniprocessor fixed priority (FP) scheduled systems, blocking time can be minimised by the use of a *Priority Ceiling inheritance Protocol* (PCP). With this, accesses to resources are serialised, mutual exclusion is furnished without the use of locks and multiple resources can be used in a manner that is guaranteed to be deadlock free. For systems scheduled by the EDF scheme, Baker [2, 1] generalised PCP to define a *Stack Resource Policy* (SRP). This protocol has become the defacto policy to use with EDF to gain effective control over the use of shared resources[1].

In this paper we propose an alternative protocol for EDF scheduled systems. Rather than assigning a *preemption ceiling* to each shared resource (as SRP does), a *deadline floor* is computed. And rather than raise the priority of a task to the ceiling level when it accesses a resource (as PCP does), the task reduces its current deadline to reflect the floor value of the resource. We show that this *Deadline Floor inheritance Protocol* (DFP) has all the key properties of SRP, and leads to the same worst-case blocking. However, DFP is arguable much easier to understand and more efficient to implement. It is, at the very least, an alternative scheme that implementors should evaluate when supporting EDF in real-time operating systems or languages.

[1] Over 850 citation for these two papers are recorded in Google Scholar.

In the remainder of this paper we first introduce a system model in Section 2, resource sharing policies are reviewed in Section 3 and a review of scheduling analysis is included in Section 4. DFP is defined, and its key properties explored, in Section 5. Section 6 then addresses the implementation of DFP. A brief consideration of mixed EDF and fixed priority scheduling is given in Section 7. Conclusions are contained in Section 8.

2 System Model

A hard real-time system comprises a set of n real-time tasks $\{\tau_i, \tau_2, ..., \tau_n\}$ executing on a uniprocessor, each task consists of a potentially unbounded stream of jobs which must be completed before their deadlines. Let τ_i indicate any given task of the system, and let j_i indicate any given job of τ_i. Each task can be periodic or sporadic. Tasks do not suffer release jitter (although this can easily be incorporated into the model [22]).

All jobs of a periodic task have a regular inter-arrival time T_i, we call T_i the period of τ_i. If a job for a periodic task arrives at time t, then the next job of τ_i must arrive at $t + T_i$.

The jobs of a sporadic task arrive irregularly, but they have a minimum inter-arrival time also denoted as T_i, we again call T_i the period of τ_i. If a job of the sporadic task τ_i arrives at time t, then the next job of τ_i can arrive at any time at or after $t + T_i$.

Each job of task τ_i requires up to the same worst-case execution time which equals the task's worst-case execution time C_i. Each job of τ_i has the same relative deadline which equals the task's relative deadline D_i; each D_i could be less than, equal to, or greater than T_i. These three cases being referred to as *constrained* deadlines, *implicit* deadlines and *unconstrained* deadlines. For unconstrained deadline tasks (and hence all tasks) it is assumed that no two jobs from the same task are ever active at the same time. For this reason the term *task* will also be used to refer to the current job from that task.

The smallest relative deadline in the system is denoted by D_{min}; the largest by D_{max}. If a job of τ_i arrives at time t, the required worst-case execution time C_i must be completed within D_i time units, and the absolute deadline of this job (referred to by lower case d_i) is $t + D_i$. The term *deadline* refers to an absolute deadline of some job in the system.

Let U_i denote the utilization of τ_i (ie. $U_i = C_i/T_i$), and define U to be the total utilization of the task set, computed by $U = \sum_{i=1}^{n} U_i$.

Contained within the system are m shared resources $(r^1, ..., r^m)$. Tasks may access (under mutual exclusion) these resources, but we make no assumption as to

when each job accesses these shared resources during its execution. We do assume however that tasks do not self-suspend whilst accessing a resource. The worst-case execution time of task τ_i when using resource r^j is denoted as C_i^j. Note that $C_i^j = 0$ implies that the task does not access the resource. The worst case execution time for each task includes the time it takes executing with the resources it accesses (so the quantity $\sum_{j=1}^{m} C_i^j$ is included in the parameter C_i).

The set of tasks that may *access* resource r^j is denoted by $\mathcal{A}(r^j)$. When a task has access to a resource, the resource is said to be *held*, otherwise it is *free*[2]. In contexts where there is only a single resource the symbol r (without a superscript) will be used.

The inclusion of shared resources in the system model implies that tasks may suffer *blocking* – which must be taken into account in the scheduling analysis.

According to the EDF scheduling algorithm, in the absence of blocking, the job with the earliest absolute deadline has the highest priority and will be executed on the processor. If more than one job has the same deadline then they are scheduled in FIFO order; the one that has been in the system the longest time will execute first. At any time, a released job with an earlier absolute deadline will preempt the execution of a job with a later absolute deadline. When a job completes its execution the system chooses, for execution, the oldest pending (released) job with the earliest deadline.

3 Resource Sharing Policies

There are a number of protocols existing for accessing shared resources under the EDF scheduling environment, for example: Stack Resource Policy (SRP) [2, 1], Dynamic Priority Ceiling [8], Dynamic Priority Inheritance (DPI) [18], and Dynamic Deadline Modification (DDM) [11]. This last approach is closest to the one proposed in this paper as it also involves changing the deadlines of jobs that access resources. A comparison of DDM and DFP is given later in the paper (see Section 5.4).

As indicated above, the SRP was proposed for accessing shared resources as a generalisation of the Priority Inheritance Protocol (PIP) [16] and the Priority Ceiling Protocol (PCP) [16]. It has the advantage that it can be integrated into the EDF scheduling framework. Under PIP a task is blocked at the time when it attempts to enter a critical section, while under PCP and SRP a task is blocked at the time when it is released and attempts to preempt a lower priority task. This

[2]We do not use the terms *locked* and *unlocked* as actual operating system locks are not necessary to ensure mutual exclusive access.

property of SRP reduces context switches and stack usage (hence the name of the protocol).

As SRP is the most popular protocol to use with EDF we now describe in more detail SRP for EDF-based systems. An example of the use of the protocol is also provided. Note that SRP, as introduced by Baker, is a more general protocol that can deal with other forms of dispatching urgency and resources with alternative synchronisation constraints. Here we are only concerned with its use for EDF scheduled systems and resources requiring mutual exclusion synchronisation.

3.1 The SRP Algorithm

Under SRP each job j_i of task τ_i is assigned a preemption level $\pi(\tau_i)$. Under EDF scheduling, the preemption level of a job correlates inversely to its relative deadline, ie. $\pi(\tau_i) < \pi(\tau_j) \Leftrightarrow D_i > D_j$.

Define $r^1, r^2, ..., r^m$ to be the non-preemptable shared resources in the system. Each resource, r^j, is assigned a ceiling preemption level denoted as $\Pi(r^j)$ which is set equal to the maximum preemption level of any job that may access it. Let $\hat{\pi}$ denote the highest ceiling of all the resources which are held by some job at any time t, that is:

$$\hat{\pi} = max\{\Pi(r^j) \parallel r^j \; is \; held \; at \; time \; t\}.$$

Baker [2, 1] showed that the Stack Resource Policy (SRP) has the following properties (expressed as a theorem).

Theorem 1 *([2, 1]) If no job j_i is permitted to start execution until $\pi(\tau_i) > \hat{\pi}$, then:*

1. *no job can be blocked after it starts;*
2. *there can be no transitive blocking or deadlock;*
3. *no job can be blocked for longer than the execution time of one outermost critical section of a lower priority job;*
4. *if the oldest highest-priority (ie. shortest deadline) job is blocked, it will become unblocked no later than the first instant when the currently executing job is not holding any non-preemptable resource.*

As a result of these properties, a job j_i released at time t can start execution only if:

- the absolute deadline of this job $(t + D_i)$ is the earliest deadline of the active requests in the task set; and

- the preemption level of j_i is higher than the ceiling of any resource that is held at the current time (ie. $\pi(\tau_i) > \hat{\pi}$).

This two stage test is in contrast to the single test required in DFP (see later discussions).

3.2 Example usage of SRP

Consider a three task (τ_1, τ_2, τ_3), one resource (r) system, defined in the table below. Note the 'Access Time' in the table refers to the time each task takes in accessing the resource r (it is the duration of its critical section). Note τ_1 does not access the resource. The 'Arrival Time' is when the current job of each task is released for execution.

Task	C	D	T	Access Time	Arrival Time
τ_1	3	10	20	0	3
τ_2	9	20	30	1	2
τ_3	10	30	40	4	0

Table 1: Example task Set

As $D_1 < D_2 < D_3$, the preemption levels are related as follows: $\pi(\tau_1) > \pi(\tau_2) > \pi(\tau_3)$. Since only τ_2 and τ_3 access r, the ceiling preemption level of this resource is given by $\Pi(r) \leftarrow \pi(\tau_2)$.

Assume the job of τ_3 arrives at $t = 0$ and locks the non-preemptive resource r at time $t = 1$. The highest ceiling of a locked resource at this time is now given by: $\hat{\pi} = \Pi(r) = \pi(\tau_2)$. Let the job of τ_2 be released at time, $t = 2$ (while the resource is still locked). This job is not allowed to preempt τ_3 as its preemption level is not high enough. At time $t = 3$, τ_1 is released and does preempt τ_3 as its preemption level is high enough ($\pi(\tau_1) > \Pi(r)$) and its deadline is earlier than that of τ_3 ($13 < 30$) and τ_2 ($13 < 22$).

The job of τ_1 will execute from $t = 3$ to, say, $t = 6$ when it completes. Now τ_3 can resume execution. It will execute until $t = 8$ at which point is frees the resource and as a result τ_2 can preempt and continue its execution. At some point it will access r but it is now guaranteed to be available. When the job of τ_2 terminates, τ_3 can continue. See Figure 1 for a simple representation of the execution timeline of these three jobs. Note the darker shared boxes represent the execution of a job while holding the resource.

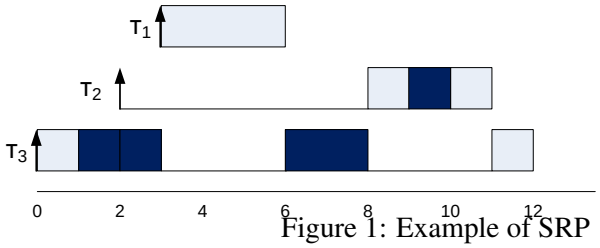

Figure 1: Example of SRP

In this example execution, τ_2 suffers blocking of duration 3 (ie. this is the interval during which a task with a later deadline is executing). The worst-case occurs when this task is released just after τ_3 locks the resource. In this situation the blocking time would be 4.

4 Review of EDF Schedulability Analysis

This section [3] describes the previous research results on exact schedulability analysis for EDF scheduling with arbitrary relative deadlines (ie. D unrelated to T). In 1980, Leung and Merrill [12] noted that a set of periodic tasks is schedulable if and only if all absolute deadlines in the interval [0,max$\{s_i\}$+2H] are met, where s_i is the start time of task τ_i, min$\{s_i\} = 0$ and H is the least common multiple of the task periods. In 1990, Baruah et al [4] extended this condition for sporadic task systems, and showed that the task set is schedulable if and only if: $\forall t > 0, h(t) \leq t$, where $h(t)$ is the processor demand function given by:

$$h(t) = \sum_{i=1}^{n} \max\{0, 1 + \left\lfloor \frac{t - D_i}{T_i} \right\rfloor\} C_i. \tag{1}$$

Using the above necessary and sufficient schedulability test, the value of t can be bounded by a certain value, we refer to this value as the *upper bound* for task schedulability. The following theorem introduces one of these upper bounds (note the total utilisation of the task set has to be strictly less than 1).

Theorem 2 *([20]) An arbitrary deadline task set with $U < 1$ is schedulable if and only if*

$$\forall t < L_a, \;\; h(t) \leq t,$$

where

$$L_a = \max\{(D_1 - T_1), ..., (D_n - T_n), \frac{\sum_{i=1}^{n} (T_i - D_i)U_i}{1 - U}\}. \tag{2}$$

[3] The review material presented in this paper is adapted from [22].

As the processor demand function can only change at the absolute deadlines of the tasks, only the absolute deadlines require to be checked in the upper bounded interval.

In 1996, Spuri [17] and Ripoll et al [9] derived another upper bound (L_b) for the time interval which guarantees we can find an overflow (ie. deadline miss) if the task set is not schedulable. This interval is called the *synchronous busy period* (the length of the first processor busy period when all tasks are released simultaneously at the beginning of the period). However, Ripoll et al [9] only considered the situation where $D_i \leq T_i$. The length of the synchronous busy period can be computed by the following process [17, 9]:

$$w^0 = \sum_{i=1}^{n} C_i, \tag{3}$$

$$w^{m+1} = \sum_{i=1}^{n} \left\lceil \frac{w^m}{T_i} \right\rceil C_i, \tag{4}$$

the recurrence stops when $w^{m+1} = w^m$, and then $L_b = w^{m+1}$.

Since the calculation of L_b has an iterative form, compared with the low complexity ($O(n)$) of the calculation of L_a, we should avoid using L_b, whenever $U \neq 1$. Moreover, in extensive simulation studies [20, 21] it was nearly always the case that $L_a < L_b$.

4.1 Processor Demand Analysis for EDF+SRP

Baker [2, 1] provided a sufficient schedulability condition for EDF+SRP; a system is schedulable if

$$\forall_{k=1,\ldots,n} \quad \left(\sum_{i=1}^{k} \frac{C_i}{D_i} + \frac{B_k}{D_k} \right) \leq 1,$$

where B_k is the maximum blocking time of τ_k; note for this equation the tasks are indexed according to their relative deadline parameter.

This sufficient test requires that $D_i \leq T_i$ for all tasks, and it is utilization based; a set of experiments [21] showed that nearly all task sets which are randomly generated cannot be accurately evaluated by such a test. Hence, an exact schedulability analysis which is based on the processor demand analysis is needed by the EDF+SRP scheduling framework.

Let $b(t)$ be a function representing the maximum time a job j_k with relative deadline $D_k \leq t$ may be blocked by job j_α with relative deadline $D_\alpha > t$ in any given time interval [0,t].

Spuri [17] showed that a condition for the schedulability of a task set is that for any absolute deadline d_i in a synchronous busy period:

$$h(d_i) + b(d_i) \leq d_i.$$

The definition of $b(t)$ given by Baruah [3] is more intuitive. Let $C_{\alpha,k}$ denote the maximum length of time for which task τ_α needs to hold some resource that may also be needed by task τ_k. Then $b(t)$ can be defined and calculated by:

$$b(t) = \max\{C_{\alpha,k} \| D_\alpha > t, D_k \leq t\}. \tag{5}$$

Note that if t is greater than the maximum relative deadline (ie. $t \geq D_{max}$) then the blocking term, $b(t)$, is zero.

The maximum interval that must be consider for schedulability can again be derived from the minimum of the two methods of obtaining the upper bound; ie. $L = min(L_b, L_a^*)$ where the L_a term has been modified as a result of the blocking that can occur in the interval up to D_{max} [22]:

$$L_a^* = \max\left\{(D_1 - T_1), ..., (D_n - T_n), \right.$$
$$\left. \frac{\max_{d_i < D_{max}}\{b(d_i)\} + \sum_{i=1}^{n}(T_i - D_i)U_i}{1 - U} \right\}. \tag{6}$$

In a given interval (eg. between 0 and L), there can be a very large number of absolute deadlines that need to be checked. This level of computation has been a serious disincentive to the adoption of EDF scheduling in practice. Fortunately a much less computation-intensive test known as Quick convergence Processor-demand Analysis (QPA) [20] has recently been proposed. Extensive experiments [21] reported that the required volume of calculations needed to perform an exact schedulability analysis can be exponentially decreased by the use of QPA.

5 Definition of the Deadline Floor Protocol

For ease of presentation we first define the Deadline Floor inheritance Protocol (DFP) for systems that do not have nested resource usage. This restriction is then removed in Section 5.5.

5.1 Initial Definition of DFP

Given an application defined by a set of tasks $(\tau_i, \tau_2, ..., \tau_n)$, a set of resources $(r^1, r^2, ..., r^m)$ and the task-resource access relation, $\mathcal{A}$, the Deadline Floor Protocol is defined as follows:

1. Each resource, r^i, has a relative deadline D^i given by:

$$D^i \ = \ min\{D_j \ : \ \tau_j \in \mathcal{A}(r^i)\}.$$

2. When a task τ_j released at time s accesses resource r^i at time t (so $s < t$) its relative deadline is immediately reduced to D^i, as a result its active absolute deadline is also (potentially) reduced; that is $d_j \leftarrow min\{t + D^i, s + D_j\}$.

3. When this task frees the resource its deadline immediately returns to its original value, that is $d_j \leftarrow s + D_j$.

Note that a task accessing a resource close to its deadline may not have its deadline reduced. For example, a task released at time 42 with an absolute deadline of 84, that accesses a resource with a deadline floor value of 8 will have its deadline reduced to 60 if it accesses the resource at time 52, but will stay with its deadline of 84 if it accesses the resource at time 80.

The static absolute deadline of a job released at time t is termed the job's *base* deadline. A task also has a dynamic *active* deadline. When accessing a resource the task's active deadline may be reduced to reflect the resource's relative deadline floor. When no resources are held, the active deadline of a job is the same as the base deadline. Tasks are scheduled according to their active deadlines.

A comparison with the Priority Ceiling Protocol (PCP) for fixed priority (FP) scheduled systems shows that the protocols are structurally equivalent. Under PCP a resource has a priority equal to the highest priority of any task that uses it. On entry to the resource the task's priority is raised to the ceiling value, on exit its priority returns to its previous value. As dispatching urgency is reflected by higher priority under FP, and earlier deadline under EDF, the use of a ceiling value for the former and a floor value for the latter is to be expected.

5.2 An example of the use of DFP

Before proving the significant properties of DFP, the example used earlier to illustrate SRP (see Table 1) will re-interpreted for DFP.

First the resource r must be given a deadline floor. It is used by τ_2 and τ_3, so its relative deadline is given by $D^r \leftarrow min(20, 30) = 20$. At $t = 1$, τ_3 (which was released at $t = 0$) accesses r and as a result its active deadline is reduced from 30 to 21. At $t = 2$, τ_2 is released with deadline 22, it will not preempt as its deadline is later than τ_3's current active deadline. Again at time $t = 3$, τ_1 is released with deadline 13, it will preempt (as $13 < 21$). This job will execute until it completes at which point τ_3 will continue until it releases the resource; its active deadline will then change from 21 to 30 and as a result τ_2 will preempt (as $22 < 30$).

Note, in this example, the same order of execution of the tasks occurs for DFP and SRP, however DFP only manipulates deadlines, SRP requires preemption levels as well. Under DFP, like SRP, τ_2 suffers its blocking at its release before it actually starts executing. Note also that, in general, the two protocols do not give rise to the same execution sequences.

5.3 Initial Properties of DFP

First we show that the protocol itself ensures mutual exclusive access to any resource. And that a task/job is never blocked once it starts executing.

Lemma 1 *Whilst accessing a resource, a task cannot be preempted by any other task that could access the same resource.*

Proof. Assume task τ_j accesses resource r (with deadline ceiling D^r) at time t. Assume, to construct a contradiction, that task τ_k preempts τ_j (either directly or preempts some other task that has preempted τ_j etc.) at time t' and then attempts to access r.

To preempt, $d_k < d_j$ and $t' > t$. At t', τ_j hold r and hence $d_j = t + D^r$. If $\tau_k \in \mathcal{A}(r)$ then $D^r \leq D_k$. Hence $d_k < t + D^r \leq t + D_k < t' + D_k = d_k$, which provides the contradiction $\Box$.

Lemma 2 *No task can be blocked after it starts executing.*

Proof. Assume task τ_j is released at time t, starts executing and will subsequently access resource r. Assume, to construct a contradiction, that task τ_k released before t holds r. As tasks cannot self-suspend, τ_k must be runnable.

As both tasks access the resource, $D^r \leq min\{D_j, D_k\}$. Let τ_k access the resource at time $t', t' < t$. As $\tau_k \in \mathcal{A}(r)$ then $d_k = t' + D^r$. To preempt τ_k, τ_j must have an earlier deadline, $d_j < d_k$. Hence $d_j = t + D_j < t' + D^r$. But $t' + D^r \leq t' + D_j < t + D_j = d_j$; so $d_j < d_j$ which provides the contradiction $\Box$.

Next we note that when released at most one resource can be held.

Lemma 3 *When released for execution at most one resource needed by the released task (τ_j) will be held by another task with a longer deadline than τ_j.*

Proof. Assume task τ_j is released at time t. Let a resource (r) be held by task τ_k with access time t' and $t' < t$. For a second resource to be accessed by another task, τ_p, released at time t'', it must preempt τ_k; so $t' < t'' < t$.

To preempt, $d_p = t'' + D_p < t' + D^r < t' + D_j < t + D_j = d_j$. So any task that preempts another task that holds a resource needed by τ_j cannot have a deadline greater than τ_j. This is a stronger property than that required by the Lemma □.

It is now possible to state, in the form of a theorem, the basic property of DFP.

Theorem 3 *When any task τ_i is released for execution at most one other task with a base deadline greater than that of τ_i will have an active deadline less than that of τ_i.*

Proof. Follows from the proof of the previous lemma □.

5.4 Comparing DFP and DDM

Having introduced DFP it is now possible to compare it with the protocol (DDM) introduced by Jeffay in 1992 [11]. The formulation of DDM is very different, but the effect for non-nested resources is similar. Under DDM a job is split into a number of *phases*. Each phase involves the use of at most one resource. For each phase a phase-specific deadline is computed based on the shortest deadline of all the other jobs that use that phase's resource. In effect this means that a deadline floor value is computed for each resource. Under DDM each phase of a job has a distinct deadline, under DFP each resource access has a distinct deadline. In modeling terms these are equivalent; but in terms of practice, DFP by its association of a deadline with the resource, provides an easy implementation scheme (see later discussion) and allows complex program structures to be accommodated. For example, branching structures where each branch accesses a different resource. More significantly, DFP is defined to work with nested resource accesses (see next section), DDM does not support phases within phases. Also the treatment of DFP in this paper:

- Uses a different formulation and proof structure; one that is arguable more straightforward to follow (the paper on DDM [11] does not include full details of the proofs for multiple phased tasks).
- Shows an equivalence between DFP and the optimal SRP.
- Shows how to compute the optimal blocking term for schedulability analysis.
- Shows how the protocol can be incorporated into a concurrent programming language.

5.5 Nested Resource Usage

In a general system, resources can make use of other resources and hence nested relationships are possible. This could lead to transient blocking and even deadlocks. Here we show that DFP, like SRP, prevents these conditions from arising. To achieve these useful properties, however, resources must be used in a strictly nested way. So, for resources A and B:

access(A) access(B) ... release(B) release(A)

is acceptable, but

access(A) access(B) ... release(A) release(B)

is not.

To cater for nested resource usage the definition of the protocol must be modified slightly.

1. Each resource, r^i, has a relative deadline D^i given by:

$$D^i = min\{D_j : \tau_j \in \mathcal{A}(r^i)\}.$$

2. When a task τ_j accesses resource r^i at time t its relative deadline is immediately reduced to D^i, as a result its active absolute deadline is also (potentially) reduced; that is $d_j \leftarrow min\{t + D^i, d_j\}$. Its initial deadline (before being reduced) is held in the variable d_j^i

3. When this task frees the resource its deadline immediately returns to its previous value, that is $d_j \leftarrow d_j^i$.

An OS implementation could store the d_j^i values as part of the resource or on a per-task stack (of maximum size equal to the depth of the resource nesting).

5.6 Further Properties of DFP

First we note that Lemma 1 and 2 hold for the extended definition of the protocol. Lemma 3 needs to be reformulated. Where resource usage is nested we introduce, following Baker, the term *outermost* resource to indicate the one that is called directly by the task (not via another resource, or while the task is holding another resource). Note the execution time within an outermost resource includes the time spent executing within the inner resources.

Lemma 4 *When released for execution at most one outermost resource needed by the released task (τ_j) will be held by another task with a longer deadline than τ_j.*

Proof. Assume task τ_j is released at time t. Let an outermost resource (r^o) be held by task τ_k with access time t' with $t' < t$. For a further resource to be accessed by another task, τ_p, released at time t'', τ_p must preempt τ_k; so $t' < t'' < t$.

To preempt, $d_p = t'' + D_p < t' + D^o < t' + D_j < t + D_j = d_j$. So any task that preempts another task that holds an outermost resource needed by τ_j cannot have a deadline greater than τ_j. This is sufficient to prove the Lemma □.

Next we show the protocol leads to behaviour that is free of transitive blocking and deadlocks.

Lemma 5 *The DFP protocol is free from transitive blocking and deadlocks.*

Proof. In order to get transitive blocking or deadlock it must be the case that a task gains access to a resource and then attempts to access another resource but that resource is held by another task. Lemmas 2 and 4 shows that this situation cannot occur □.

The final property to note concerns a bound on the blocking suffered by the most urgent task.

Lemma 6 *If the oldest earliest deadline job is blocked, it will become unblocked no later than the first instant when the currently executing job is not holding any resource.*

Proof. Assume task τ_j is released at time t. It is blocked by task τ_k as τ_k is holding an outermost resource r^o that it accessed at time t'. So, $t' < t$, $d_j < t' + D_k$, but $d_j > t' + D^o$ (as τ_k is blocking τ_j). Task τ_k may be preempted by shorter deadline tasks (that by the action of the protocol will not be accessing r^i) but as some time t'' it will free the resource. At this time the deadline of τ_k will return to its basic value ($t' + D_k$) and all blocked tasks (including τ_j) will have deadlines earlier than this value. The one with the shortest deadline will execute next. If there are a number of tasks with the same (earliest) deadline then the one that has been in the system the longest time (ie. the oldest job) will execute (unless a shorter deadline job is released at the same instant) □.

It is now possible to prove a theorem equivalent to the one for SRP.

Theorem 4 *The Deadline Floor inheritance Protocol has the following properties*

1. *no job can be blocked after it starts;*

2. *there can be no transitive blocking or deadlock;*

3. *no job can be blocked for longer than the execution time of one outermost critical section;*

4. *if the oldest earliest deadline job is blocked, it will become unblocked no later than the first instant when the currently executing job is not holding any resource.*

Proof. Follows directly from Lemmas 1, 2, 4, 5 and 6 □.

5.7 Feasibility Analysis for EDF+DFP

Here we derive schedulability analysis for EDF+DFP by following the strategy employed by Baruah for EDF+SRP [3]. The general approach is to postulate the circumstances in which a deadline is missed, and then use this situation to derive a worst-case bound for schedulability. We will show that this bound for EDF+DFP is the same as that of EDF+SRP.

In order to concentrate on the blocking term, assume the task set under consideration is schedulable if resource usage is ignored. That is, $\forall s : \ h(s) \leq s$. Assume a first deadline miss occurs at time t. So $h(t) + b(t) > t$. The function $h(t)$ is defined by equation (1), we need a formula for $b(t)$ for DFP equivalent to the one given earlier for SRP (equation 5).

Let t' be the last time, before t, that there were no pending job executions with arrival times before t' and deadlines before or at t. So the processor is busy between t' and t with jobs that have deadlines at or before t. Without loss of generality assume t' is time 0. The maximum load on the system at time t assumes all tasks giving rise to jobs with deadlines at or before t are released at time 0.

For $b(t) > 0$, a job released at or before 0 with a deadline after t must have accessed a resource before time 0 and inherited a deadline so that its deadline is reduced to be at or before t. It follows from the properties of DFP discussed earlier that there is at most one job with a deadline after t that executes and accesses a resource at or before time 0 and executes with the resource in the interval [0,t). A formula for $b(t)$ is therefore of the form:

$$b(t) = \max\{C_j^r\},$$

where the $\max$ is taken over all tasks and all resources that can give rise to blocking being experienced at the time (t) a deadline is missed.

Let any job that can cause blocking come from task τ_j. If τ_j is released and accessed resource r just before time 0 then blocking could occur at time t if $D_j > t$

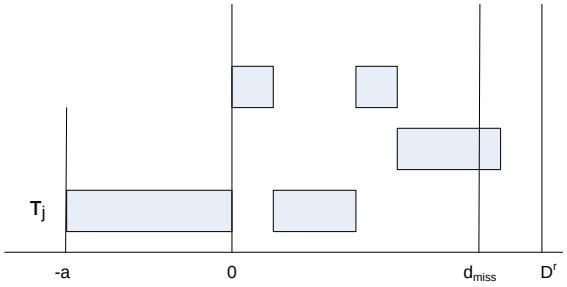

Figure 2: Case 1

(the deadline of τ_j is $0 + D_j$) and $D^r \leq t$ (the deadline of τ_j will be reduced to $0 + D^r$ when the resource is held).

However, this assumes that the resource is accessed at time 0. If it is accessed earlier then its inherited deadline could have a smaller value and hence blocking could occur for values of $t < D^r$. For example, if $D^r = 10$ and the resource is accessed at time -6 then the active deadline of τ_j would be 4. Nevertheless, we will now show that accessing the resource at time 0 is the worst-case, and that this leads to the result that the worst-case blocking of EDF+DFP is the same as that for EDF+SRP. The notion of worst-case here means that if task τ_j when accessing resource r can cause a deadline to be missed before time D^r then it will also cause a deadline miss at D^r; hence potential blocking times before D^r do not need to be considered.

Lemma 7 *If a task τ_j accesses a resource r at time 0 and there is no deadline miss at or after D^r then there will be no deadline miss before D^r even if τ_j accesses r before time 0.*

Proof. Assume that the size of the blocking factor, the duration of the critical section of resource r, is B, ie. $b(D^r) = B = C_j^r$. Also assume that the resource r is accessed at a before time 0; ie. at time $-a$. To construct a counter example assume that there is a deadline miss at time d_{miss}, with $d_{miss} < D^r$, but no deadline miss at D^r.

The proof will be structured into three cases (the better to illustrate the intuition behind the proof).

Case 1. Assume only τ_j executes between the resource being accesses at time -a and time 0 (see Figure 2 for a simple three task system that has this property). Let B^1 be the duration of execution before time 0, and B^2 after, so $B^1 + B^2 = B$.

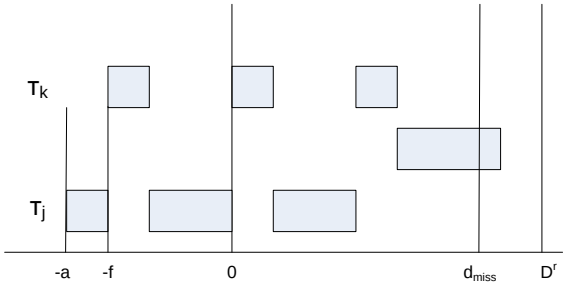

Figure 3: Case 2

To cause blocking at time d_{miss} the inherited deadline of τ_j must be less than or equal to d_{miss}. That is, $-a + D^r \leq d_{miss}$, implying $-B^1 + D^r \leq d_{miss}$ and hence $d_{miss} + B^1 \geq D^r$.
To cause a deadline miss at time d_{miss} then

$$h(d_{miss}) + B^2 > d_{miss}$$

hence

$$h(d_{miss}) + B^2 + B^1 > d_{miss} + B^1$$

giving

$$h(d_{miss}) + B > d_{miss} + B^1 \geq D^r.$$

Now $h(D^r) \geq h(d_{miss})$ so

$$h(D^r) + B > D^r$$

which implies a deadline miss at D^r and provides the contradiction.
Case 2. To bring the deadline miss even earlier, the resource must be accessed earlier. This can only happen if τ_j is preempted by a shorter deadline task (τ_k) after it has accessed the resource (after $-a$). In Case 2 we assume a single preempting job from task τ_k (see figure 3 for an illustration of this possibility in which τ_k is released at time $-f$ with $-f > -a$).

If the releases of τ_j and τ_k are now postponed (moved to the right in the figure) by the amount ($D^r - d_{miss}$), which can occur as $D^r - d_{miss} \leq a$, then the initial processor demand at time d_{miss} will be moved to time D^r. Moreover there will also be the additional demand coming from the (partial) executions of τ_j and τ_k moving beyond time 0: this is equal to the duration of the release postponements ($D^r - d_{miss}$). It follows that

$$h(D^r) + b(D^r) \geq h(d_{miss}) + b(d_{miss}) + (D^r - d_{miss}),$$

but as there is a deadline miss at d_{miss}, so

$$h(d_{miss}) + b(d_{miss}) > d_{miss}$$

hence

$$h(D^r) + b(D^r) > d_{miss} + D^r - d_{miss}$$

implying

$$h(D^r) + b(D^r) > D^r,$$

which implies a deadline miss as D^r and provides the contradiction.

Case 3. Finally we now consider a number of jobs from any number of tasks interfering with τ_j before time 0 while it has hold of the resource. By the same argument used in Case 2 if all the job executions before time 0 are postponed by $(D^r - d_{miss})$ then the deadline miss at d_{miss} will move to a deadline miss at D^r.

This completes the proof □.

This lemma shows that the assumption that the resource is accessed by τ_j just before time 0 captures the worst-case. Moreover, the interval over which this task can cause blocking is maximised by also assuming that τ_j is actually released just before time 0. It can then cause blocking at any point in the interval $[D^r, D_j)$. The magnitude of the blocking term is determined by the action of τ_j whilst accessing resource r. Let the blocking term identified as B in the above proof be represented more precisely by C_j^r (the time task τ_j is executing with resource r).

Returning to the deadline miss at time t. For this to occur there must be a blocking value ($b(t) > 0$). To prevent a deadline miss the maximum blocking term must be bounded by $t - h(t)$, ie $b(t) \leq t - h(t)$. As only one task can be causing blocking, the maximum blocking term at time t is given by:

$$b(t) = \max \{C_j^r \| D_j > t, D^r \leq t\}, \tag{7}$$

where the $\max$ is taken over all tasks and all resources.

Theorem 5 *The following condition is sufficient for guaranteeing that all deadlines are met under EDF+DFP:*

$$\forall t > 0 : b(t) + \sum_{i=1}^{n} \max\{0, 1 + \left\lfloor \frac{t - D_i}{T_i} \right\rfloor\} C_i \leq t, \tag{8}$$

where $b(t)$ is computed by equation(7).

Proof. Follows directly from above □.

It is now possible to show that the blocking term for EDF+DFP is the same as that for EDF+SRP. The protocols are therefore equivalent.

Theorem 6 *The worst-case processor demand for EDF+DFP is the same as that computed by EDF+SRP.*

Proof. Recall the definition of the blocking term for SRP given in Section 4.1 (equation 5).

$$b^{SRP}(t) = \max\{C_{\alpha,k} \| D_\alpha > t, D_k \leq t\}.$$

where $C_{\alpha,k}$ denote the maximum length of time for which task τ_α needs to hold some resource that may also be needed by task τ_k (with $D_k \leq t$).

For DFP the blocking term (using the same task names) is

$$b^{DFP}(t) = \max\{C^r_\alpha \| D_\alpha > t, D^r \leq t\}.$$

If in $b^{SRP}(t)$ task τ_k may access a resource, r, then the deadline floor of this resource (D^r) must have the property: $D^r \leq D_k \leq t$. And hence $b^{DFP}(t)$ would also contain this term and $C^r_\alpha = C_{\alpha,k}$. Similarly if a resource is contained in the $b^{DFP}(t)$ term then there will be a corresponding task in $b^{SRP}(t)$. This completes the proof □.

Note that the above shows that the two protocols (DFP and SRP) are equivalent; but they are not identical. They can give rise to different execution sequences at run-time. Consider the example given in Table 1 (which resulted in identical behaviour from the two protocols). Now change the relative deadline of the first task to 18 (ie. $D_1 = 18$) rather than 10. The execution sequence of SRP remains the same (as depicted in Figure 1). But under DFP, τ_1 will not preempt τ_3 as its deadline (21 = 3 + 18) is not strictly less that the current inherited deadline of τ_3 which is also 21. Nevertheless τ_3 will still complete before its deadline.

Observation

Although the worst case blocking occurs when the resource access occurs at the same time as the task is released, this does not mean that there is an equivalent protocol in which the deadline of the job, when accessing a resource, is reduced to 'job release time + deadline floor' (rather than 'now + deadline floor'). With this protocol a job could be released 'early', be preempted by jobs with deadlines greater than t (the hypothesised time of the deadline miss) and then be allowed to access the resource at time 0 (with an inherited deadline close to, or even before 0. The blocking term would now have to be included for arbitrary small deadlines.

The DFP approach works because the deadline inherited value is only computed at the time the resources is actually accessed. So simultaneously released jobs with deadlines less that D^r do not suffer blocking.

6 Implementation of DFP

To implement EDF scheduling, the associated language run-time support system or RTOS must keep a *ready* queue, ordered by absolute deadline, of all the runnable tasks. When a task is released for execution (for example when a delay statement expires) its absolute deadline must be computed and the task is then inserted at the appropriate place in the ready queue. The task as the head of the queue has the earliest deadline and is therefore chosen for execution.

To extend this implementation scheme to incorporate DFP is straightforward. Each resource access requires a *pre* and *post* protocol that manipulates the deadline of the client task. The primitive to change the task's deadline is already present in the RTOS (if it supports EDF at all). On exiting a resource the *post* protocol must return the task's deadline to the value stored during the *pre* protocol. This simple scheme clearly deals with nested resource usage as long as the relationship between the resources is one of strict nesting. The following gives pseudo code (using Ada) that would need to be executed in the RTOS kernel for these pre and post protocols:

```
-- pre protocol:
D := Get_Deadline;  -- read the absolute deadline of the task
New_Deadline := Clock + Deadline_Floor;
if New_Deadline < D then
  Set_Deadline(New_Deadline); -- set new absolute deadline
end if;

-- code for accessing the resource

-- post protocol:
Set_Deadline(D); -- re-set old absolute deadline
```

The constant `Deadline_Floor` holds the deadline floor value for the resource (it would be initialised at the beginning of the program). The subprograms `Get_Deadline`, `Clock` and `Set_Deadline` deliver the behaviours implied by their names.

The overheads of the protocols are simply the cost of reading the local real-time clock plus the cost of two deadline changes to the executing task. Note the first deadline change, as it is to the executing task and is a deadline reduction, cannot lead to a context switch. Only the re-setting of the old deadline could result in a task switch (if a more urgent task had been released during the execution of the resource's code) and this is a task switch that would occur anyway. The re-setting of a task's deadline could be accommodated efficiently by retaining a link to the original or previous 'position' of the task in the ready queue. No sorting of the ready queue or expensive insertion is required.

The implementation of DFP is no more complex than the priority ceiling protocol for fixed priority scheduling which is available via many RTOSs and program-

ming languages such as Java and Ada. By comparison, under SRP a task must have a deadline and a consistent preemption level. The Ada programming language has implemented SRP as part of its support for EDF scheduling [6]. To give a complete implementation for SRP a programming language must specify what happens to tasks that are released but which do not preempt the currently executing task. For example, a task could chain through a set of nested resources, during this time a number of other tasks could be released with different preemption levels and deadlines. To ensure that the right order of execution is maintained Ada uses ready queues at each preemption level. The protocol is not intuitive; indeed an early version of the protocol was shown to be incorrect [19]. Moreover, an initial implementation of the run-time was shown to be inconsistent with the language rules[10].

The correct rule for preemption is defined by the following text in the reference manual for Ada [5]: A task T is placed on the ready queue for priority level P (note a resource is represented by a protected object in Ada), where P is defined by

> *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 all other tasks on ready queues with priorities strictly less than P.*

Of course this quote is without its context, nevertheless it illustrates the complexity of embedding SRP into the semantics of a programming language. By comparison the priority ceiling protocol (for fixed priority scheduling) is straightforward to define.

6.1 Preliminary Evidence

It has not been possible to modify an existing operating system (or language run-time support system) to implement DFP. Nevertheless some preliminary experimentation has been possible using the Marte [14] run-time system for Ada[4] (which, as noted above, supports SRP).

To give an indication of the overheads involved in supporting SRP and DFP a simple program is executed involving a single task repeatedly entering and leaving a simple resource. The resource chosen for the experiments is a shared integer variable that is updated by the task. The experiments were undertaken on a Pentium 4 board with a single 2.4GHz processor. Both elapse time and execution time were measured as Ada provides an API for both of these metrics (for a single task program these values should, and indeed were, almost identical).

[4] Available from http://marte.unican.es/.

The experiment had three phases, all used EDF scheduling. First, the resource is encapsulated in an unprotected procedure, this gives a base-line cost for the behaviour of the program. Second, the resource in encapsulated in a protected object that implements SRP (as defined by the Ada language). Finally, the unprotected procedure is augmented by the *pre* and *post* protocols outlined in the pseudo code above. This implements DFP at the application-level[5].

The program (included in the Appendix) was run a number of times, leading to results being presented to four significant figures. Table 2 has these results. All times are in seconds and are for 1,000,000 calls on the resource.

Phase	Elapse Time	Execution Time
Procedure	0.008607	0.008607
SRP	2.094	2.094
DFP	1.555	1.555

Table 2: Experimental Results

These results indicate that SRP and DFP lead to similar timings. Both give rise to execution costs significantly greater than that of the standard procedure call. For these experiments DFP is approximately 75% of the cost of SRP. This is a significant gain. If DFP was directly supported by the RTOS it is expected that its overhead would be much lower as a significant proportion of the cost in the experiments is involved in converting the reading of the hardware clock into a value of the appropriate application-level type. If the protocol were executed in kernel mode these cost would not arise.

There are also further costs involved with implementing SRP within a programming language's semantics. Although in theory a task should never access a resource with an active priority higher than the ceiling, at run-time a test must be made to check that this is indeed the case (as programs cannot be assumed to be correct). If the priority is too high then an exception could be raised or a error return value used to identify the error. This test has a cost. With DFP no equivalent test is needed as a task can enter a resource with an active deadline shorter than the floor value. In this situation all that occurs is that the task's active deadline is not reduced – a saving in overhead.

The results presented above are however preliminary and involve a single Ada implementation. Nothing more is claimed than that DFP has the potential to have

[5]To be reliable, this code must not be preempted between reading the clock and setting the new deadline. This cannot be guaranteed for application-level code but would be easy to implement in the kernel of an RTOS or language run-time support system.

much lower overheads than SRP.

In terms of understandability, it is difficult to argue that one protocol is intrinsically easier to understand than another. However the experience of supporting SRP within the Ada language did show that SRP is open to misinterpretation. Moreover, in an educational context within courses on scheduling, the intuition behind SRP does seem to be difficult for students to grasp, particularly with nested resource usage.

7 Systems Combining EDF and FP

A comparison of EDF and FP (fixed priority) could reasonable conclude that EDF has the advantage of optimal processor utilisation, whilst FP is more deterministic and hence predicable in overload conditions. This leads to a hybrid scheduling scheme in which a small number of high integrity tasks run under FP, but the bulk of the tasks execute under EDF at a system priority below that of the FP tasks. This type of scheme can by programmed in Ada and analysis has been provided for the combined set of tasks [7].

In this work ([7]) tasks are divided into two sets (EDF and FP) but they are allowed to share resources (protected objects in Ada). This sharing in managed by an implementation of SRP. Under DFP a different approach could be defined in which a protected object (PO) has both a priority ceiling *and* a deadline floor. As a task enters a PO both its priority and deadline are changed to the PO's ceiling/floor values. In this way the requirements of the EDF and FP protocols are simultaneously satisfied.

8 Conclusion

In this paper we have introduced a new protocol for controlling access to shared resources within the EDF scheduling framework. We have shown that this protocol is equivalent to the Stack Resource Protocol which is the defacto protocol to use with EDF. The new protocol requires all shared resources to have a relative deadline defined; this is the minimum (floor) of the relative deadlines of all tasks that use the resource. When a task accesses a resource at time t its absolute deadline is immediately reduced to the value $t+$ the deadline floor of the resource. The resulting *immediate deadline floor inheritance protocol* is identified here by the shorter title: **Deadline Floor Protocol**, DFP. It has an identical form to the *immediate priority ceiling inheritance protocol* (usually shortened to PCP) that is the standard approach to use within the fixed priority scheduling framework.

The Deadline Floor Protocol has all the effective properties of the Stack Resource Protocol. On a uniprocessor this means that tasks suffer at most one block from a longer deadline task, this block occurs before the task actually starts executing, all resources are accessed under mutual exclusion without the need for further locks, and the protocol itself ensures deadlock free execution.

The motivation for defining DFP is that it leads to a straightforward and efficient means of implementation. The single notion of a task's deadline is all that is needed to define and support the protocol. By comparison, the Stack Resource Protocol requires deadlines and preemption levels, and these preemption levels must be assigned in a manner consistent with the deadlines. The implementation must then keep track of both deadlines (for EDF scheduling) and the maximum system preemption level (for SRP control).

This paper has not consider multiprocessor systems. It is however possible to envisage multiprocessor versions of the protocol in the same may that PCP has give rise to a number of such multiprocessor protocols. The development of these protocols for DFP is part of future work.

Acknowledgements

This work is supported in part by the EPSRC funded Tempo project in the UK. The author would like to thank Sanjoy Baruah for a number of very useful discussions on the topic of this paper.

References

[1] T.P. Baker. A stack-based resource allocation policy for realtime processes. In *Proceedings IEEE Real-Time Systems Symposium (RTSS)*, pages 191–200, 1990.

[2] T.P. Baker. Stack-based scheduling of realtime processes. *Journal of Real-Time Systems*, 3(1), March 1991.

[3] S.K. Baruah. Resource sharing in EDF-scheduled systems: A closer look. In *Proceedings of IEEE Real-Time Systems Symposium (RTSS)*, pages 379–387, 2006.

[4] S.K. Baruah, A.K. Mok, and L.E. Rosier. Preemptive scheduling of hard real-time sporadic tasks on one processor. In *Proceedings of IEEE Real-Time Systems Symposium (RTSS)*, pages 182–190, 1990.

[5] R. Brukardt(ed). Ada 2005 reference manual. Technical report, ISO, 2006.

[6] A. Burns, A.J. Wellings, and T.Taft. Supporting deadlines and EDF scheduling in Ada. In *Reliable Software Technologies, Proceedings of the Ada Europe Conference*, pages 156–165. Springer Verlag, LNCS 3063, 2004.

[7] A. Burns, A.J. Wellings, and F. Zhang. Combining EDF and FP scheduling: Analysis and implementation in Ada 2005. In F. Kordon and Y Kermarrec, editors, *Proceedings of Reliable Software Technologies - Ada-Europe 2009*, volume LNCS 5570, pages 119–133. Springer, 2009.

[8] M. Chen and K. Lin. Dynamic priority ceilings: A concurrency control protocol for real-time systems. real-time systems. *Journal of Real Time Systems*, 2(4):325–346, 1990.

[9] I. Ripoll A. Crespo and A.K. Mok. Improvement in feasibilty testing for real-time tasks. *Journal of Real-Time Systems*, 11(1):19–39, 1996.

[10] M.L. Fairbairn and A. Burns. Implementing and validating EDF preemption-level resource control. In M. Brorsson and L.M. Pinho, editors, *Proceedings of Reliable Software Technologies - Ada-Europe 2009*, volume LNCS 7308, pages 193–206. Springer, 2012.

[11] K. Jeffay. Scheduling sporadic tasks with shared resources in hard-real-time systems. In *Proceedings of IEEE Real-Time Systems Symposium (RTSS)*, pages 89–99, 1992.

[12] J.Y.T. Leung and M.L. Merrill. A note on preemptive scheduling of periodic real-time tasks. *Information Processing Letters*, 11(3):115–118, 1980.

[13] C.L. Liu and J.W. Layland. Scheduling algorithms for multiprogramming in a hard real-time environment. *JACM*, 20(1):46–61, 1973.

[14] M. Aldea Rivas and M. González Harbour. MaRTE OS: An Ada kernel for real-time embedded applications. In *Reliable Software Technologies, Proceedings of the Ada Europe Conference, Leuven*. Springer Verlag, LNCS 2043, 2001.

[15] L. Sha, Rajkumar R., Son S., and Chang C-H. A real-time locking protocol. *IEEE Transactions on Computers*, 40(7):793–800, 1991.

[16] L. Sha, R. Rajkumar, and J.P. Lehoczky. Priority inheritance protocols: An approach to real-time synchronisation. *IEEE Transactions on Computers*, 39(9):1175–1185, 1990.

[17] M. Spuri. Analysis of deadline schedule real-time systems. Technical Report 2772, INRIA, France, 1996.

[18] J.A. Stankovic, K. Ramamritham, M Spuri, and G Buttazzo. *Deadline Scheduling for Real-Time Systems*. Kluwer Academic Publishers, 1998.

[19] A. Zerzelidis, A. Burns, and A.J. Wellings. Correcting the EDF protocol in Ada 2005. In *Proceedings of IRTAW 13, Ada Letters, XXVII(2)*, pages 18–22, 2007.

[20] F. Zhang and A. Burns. Schedulability analysis for real-time systems with EDF scheduling. *IEEE Transaction on Computers*, 58(9):1250–1258, 2008.

[21] F. Zhang and A. Burns. Schedulability analysis for real-time systems with EDF scheduling. Technical Report YCS 426, University of York, 2008.

[22] F. Zhang and A. Burns. Schedulability analysis of EDF scheduled embedded real-time systems with resource sharing. *ACM Transaction on Embedded Systems – to appear*, 2012.

Appendix - Code for the Experiments

The code used for the experiments discussed in Section 6.1 is as follows:

```
pragma Task_Dispatching_Policy(EDF_Across_Priorities);
pragma Locking_Policy(Ceiling_Locking);
with Ada.Execution_Time; use Ada.Execution_Time;
with Ada.Text_IO; use Ada.Text_IO;
with Ada.Integer_Text_IO; use Ada.Integer_Text_IO;
with Ada.Real_Time; use Ada.Real_Time;
with Ada.Dispatching.EDF; use Ada.Dispatching.EDF;
procedure DFP1 is
  package Dur_IO is new Fixed_IO(Duration);
  use Dur_IO;

  task P is
    pragma Priority(2);
  end P;

  New_Deadline, D : Time;
  Deadline_Floor : Time_Span := Milliseconds(100);

  procedure Work(X : in out integer) is
  begin
    X := X + 1;
  end Work;

  procedure Work2(X : in out integer) is
  begin
    D := Get_Deadline;
    New_Deadline := Clock + Deadline_Floor;
    if New_Deadline < D then
      Set_Deadline(New_Deadline);
    end if;
    X := X + 1;
    Set_Deadline(D);
  end Work2;

  protected Worker is
    pragma Priority(12);
    procedure Work3(X : in out integer);
  end Worker;

  protected body Worker is
    procedure Work3(X : in out integer) is
    begin
      X := X + 1;
    end Work3;
  end Worker;


  task body P is
    CPU, CPU_New : CPU_Time;
    Start, Finish : Time;
    V : integer;
  begin
    delay 1.0;
    V := 0;
    Start := Clock; -- this is the real-time clock
    CPU := Clock; -- this is the execution time clock
    Set_Deadline(Start + Seconds(20));
    for I in 1 .. 1000000 loop
      Work2(V);
```

```
    end loop;
    Finish := Clock;  CPU_New := Clock;
    put(To_Duration(CPU_New-CPU)); new_line;
    put(To_Duration(Finish-Start)); new_line;
    put(V); new_line;

    delay 1.0;
    V := 0;
    Start := Clock;  CPU := Clock;
    Set_Deadline(Start + Seconds(20));
    for I in 1 .. 1000000 loop
      Work(V);
    end loop;
    Finish := Clock;  CPU_New := Clock;
    put(To_Duration(CPU_New-CPU)); new_line;
    put(To_Duration(Finish-Start)); new_line;
    put(V); new_line;

    delay 1.0;
    V := 0;
    Start := Clock;  CPU := Clock;
    Set_Deadline(Start + Seconds(20));
    for I in 1 .. 1000000 loop
      Worker.Work3(V);
    end loop;
    Finish := Clock;  CPU_New := Clock;
    put(To_Duration(CPU_New-CPU)); new_line;
    put(To_Duration(Finish-Start)); new_line;
    put(V);
  end P;
begin
  put_line("Main Started");
end DFP1;
```

A typical output of the program is

```
Main Started
          1.554598314
          1.554600327
    1000000
          0.008607417
          0.008607353
    1000000
          2.094458013
          2.094458002
    1000000
```

The measurements reported in this paper follow from a set of runs. Note that occasionally the times obtained for the SRP measurements (2.094458 in the above) were somewhat smaller (approximately half). This only happened rarely and no explanation for the anomaly could be found.