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

Alan Burns, *Fellow, IEEE*, Marina Gutiérrez, Mario Aldea Rivas, and Michael González Harbour, *Member, IEEE*

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 current absolute deadline is subject to an immediately reduction 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, removes the need to support preemption levels and we demonstrate that it can be implemented more efficiently.

Index Terms—Real-time systems, embedded systems, concurrency control

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 (FP) 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 [14] 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 real-time operating system (RTOS). 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 [17]. The task is said to be *blocked* by the lower priority task.

For uniprocessor fixed priority 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 [3], [2] 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 SRP 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 [7]. However, DFP is arguably much easier to understand and more efficient to implement. It is, at the very least, an

- *A. Burns is with the Computer Science Department, University of York, UK. E-mail: alan.burns@york.ac.uk.*
- *M. Gutiérrez, M.A. Rivas, and M.G. Harbour are with the Universidad de Cantabria, Spain. E-mail: {gutierrezlm, aldeam, mgh}@unican.es.*

Manuscript received 22 Apr. 2013; revised 1 Mar. 2014; accepted 17 Apr. 2014. Date of publication 7 May 2014; date of current version 8 Apr. 2015. Recommended for acceptance by A.K. Somani.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TC.2014.2322619

1. Over 1,000 citations for these two papers are recorded in Google Scholar.

alternative scheme that implementors should evaluate when supporting EDF in real-time operating systems or languages.

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. Conclusions are contained in Section 7.

2 System Model

A hard real-time system comprises a set of n real-time tasks $\{\tau_i, \tau_2, \ldots, \tau_n\}$ 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. We initially assume that tasks do not suffer release jitter. The issue of release jitter is returned to in Section 5.8.

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 *arbitrary* deadlines. For arbitrary deadline tasks (and hence all tasks) it is assumed that no two jobs from the same task are ever active (i.e., executable) 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 (i.e., $U_i = C_i/T_i$), 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 $(r^1, \ldots, r^m)$. 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 $\sum_{j=1}^{m} v_i^j C_i^j$ is included in the parameter C_i; where v_i^j is the maximum number of times any job from τ_i uses resource r^j).

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 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 policy, for example: Stack Resource Policy [3], [2], Dynamic Priority Ceiling [9], Dynamic Priority Inheritance (DPI) [20], 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) [18], the Priority Ceiling Protocol (PCP) [18] and the Immediate Priority Ceiling Protocol (IPCP) [12]. 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 IPCP and SRP a task is blocked at the time when it is released and attempts to preempt a lower priority task. This 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, i.e., $\pi(\tau_i) < \pi(\tau_j) \Leftrightarrow D_i > D_j$.

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

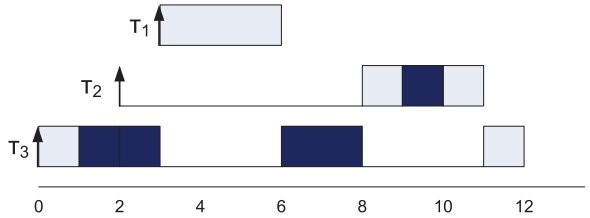

Fig. 1. Example of SRP.

Define $r^1, r^2, \ldots, r^m$ 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 [3], [2] showed that the Stack Resource Policy has the following properties (expressed as a theorem).

Theorem 1 ([2], [3]). *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 (i.e., 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 (i.e., $\pi(\tau_i) > \hat{\pi}$).

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.

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 holds the resource r at time $t = 1$. The highest ceiling of a held 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 held). 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 it 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 Fig. 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.

In this example execution, τ_2 suffers blocking of duration 3 (i.e., 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 accesses 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 (i.e., D unrelated to T). In 1980, Leung and Merrill [13] 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. [5] 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

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

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

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

where

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

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 [19] and Ripoll et al. [15] derived another upper bound (L_b) for the time interval which guarantees we can find an overflow (i.e., deadline miss) if the task set is not schedulable. This interval is called the *synchronous busy period* (the length of the first processor

3. The review material presented in this paper is adapted from [25].

busy period when all tasks are released simultaneously at the beginning of the period). However, Ripoll et al. [15] only considered the situation where $D_i \leq T_i$. The length of the synchronous busy period can be computed by the following process [19], [15]:

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

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

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 [23], [24] it was nearly always the case that $L_a < L_b$.

4.1 Processor Demand Analysis for EDF+SRP

Baker [3], [2] provided a sufficient schedulability condition for EDF+SRP; a system is schedulable if

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

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 [24] 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 [19] 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 [4] 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

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

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

The maximum interval that must be considered for schedulability can again be derived from the minimum of the two methods of obtaining the upper bound; i.e., $L = min(L_b, L_a^*)$ where the L_a term has been modified as a result of the blocking that can occur in the interval up to D_{max} [25]:

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

In a given interval (e.g., 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) [23] has recently been proposed. Extensive experiments [24] 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 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 $(\tau_i, \tau_2, \ldots, \tau_n)$, a set of resources $(r^1, r^2, \ldots, r^m)$ 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 active absolute deadline is (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 eight 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 Immediate Priority Ceiling Protocol [12] for fixed priority scheduled systems shows that the protocols are structurally equivalent. Under IPCP a resource

TABLE 1
Example Task Set

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

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 be reinterpreted 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 $22 < 30$).

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 however that, in general, the two protocols do not give rise to the same execution sequences. If the relative deadline of the first task is changed to 18 (i.e., $D_1 = 18$) rather than 10. The execution sequence of SRP remains the same (as depicted in Fig. 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.

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 holds 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. □

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. □

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. □

Finally in this section we consider the number of preemptions required by DFP when compared with SRP. The example at the end of the previous section showed that there are situations in which SRP will cause a preemption (when a newly released job arrives for execution) whereas DFP does not. In the following we show that the inverse is not possible, and hence we can conclude that DFP will lead to less preemptions and therefore more efficient implementation.

Lemma 4. *If under DFP a newly released job preempts the currently executing job, then SRP would also result in preemption.*

Proof. If no resource is in use the Lemma is obviously true. Therefore, assume task τ_a, released at time f, accesses resource r (with deadline ceiling D^r, so $D^r \leq D_a$) at time t. Its deadline is reduced to $t + D^r$ (if its deadline is not reduced the Lemma is again obviously true), so $t + D^r < f + D_a$.

As time s ($s > t$) let τ_b be released. Assume under DFP τ_b preempts τ_a. It follows that $s + D_b < t + D^r$. Hence $s + D_b < f + D_a$, and so $D_b < D_a$.

Under SRP, the system preemption level at time s is given by $\hat{\pi} = \Pi(r) \geq \pi(\tau_a)$. If preemption levels are assigned in inverse relative deadline order then

$\pi(\tau_b) > \pi(\tau_a)$. If $\Pi(r) = \pi(\tau_a)$ then $\pi(\tau_b) > \hat{\pi}$ and the rules of SRP will lead to preemption. If $\Pi(r) > \pi(\tau_a)$ then there must exist another task, τ_x with relative deadline $D_x < D_a$ that accesses the resource, r. If DFP gives preemption then $s + D_b < t + D_x$; it follows that $\Pi(r) = \pi(\tau_x)$, and $\pi(\tau_b) > \pi(\tau_x)$ and again the preconditions for preemption are true . □

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 arguably 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 extended to accommodate release jitter.
- Shows how the protocol can be implemented.

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 active absolute deadline is (potentially) reduced; that is $d_j \leftarrow min\{t + D^i, d_j\}$. Its current 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, or system-wide single, stack (of maximum size equal to the depth of the resource nesting, see Section 5.9).

5.6 Further Properties of DFP

First we note that Lemmas 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 5. *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 6. *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 5 shows that this situation cannot occur . □

The final property to note concerns a bound on the blocking suffered by the most urgent task. The most urgent job is the one with the shortest deadline. If two or more jobs share the same deadline then the one that was released first is the most urgent; it is termed the *oldest earliest deadline job*. If two jobs also share the same arrival time then a simple static tie-breaker (such as task index) is used to define the oldest task.

Lemma 7. *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^o) but at 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 (i.e., 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, 5, 6 and 7. □

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 [4]. 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:

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

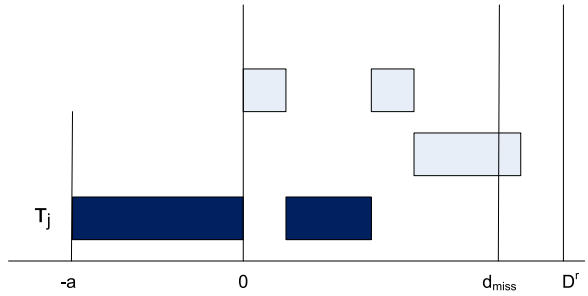

Fig. 2. Case 1.

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 before time 0 and actually accesses the resource r just before time 0 then blocking could occur at time t if $D_j > t$ (otherwise, τ_j could not be a contributor to the blocking time) 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 8. *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, i.e., $b(D^r) = B = C_j^r$. Also assume that the resource r is accessed at a before time 0; i.e., 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 accessed at time $-a$ and time 0 (see Fig. 2 for a simple three task system that has this property, again the darker shaded boxes implies that a task is executing while holding the resource). Let B^1 be the duration of execution before time 0, and B^2 after, so $B^1 + B^2 = B$.

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

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

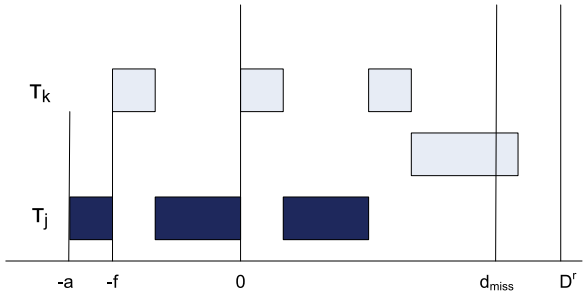

Fig. 3. Case 2.

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 Fig. 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 at 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 $t - h(t)$, i.e., $b(t) \leq t - h(t)$. 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:*

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

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 from the point of view of their worst-case blocking.

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

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

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 $C_\alpha^r = C_{\alpha,k}$. 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 in terms of their worst-case blocking; but they are not identical. They can give rise to different execution sequences at run-time as illustrated by the extended example in Section 5.2.

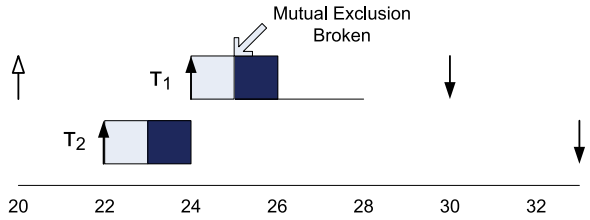

Fig. 4. Jitter example.

5.8 Adding Release Jitter to the Model

If a task should theoretically be released for execution at time t, but is not actually released for execution until a later time s, then the task is said to suffer release jitter of, in this case, $s - t$. Consider a periodic task with period 9 that is implemented on a RTOS that has a clock which only ticks in even time. Starting at time 0 the task will be released at 0 but its next release will be at 10, not 9. The subsequent release will then be 18 (followed by 28 etc.). The task has a release jitter of 1. In general, the maximum release jitter of task τ_i is denoted by the symbol J_i.

For DFP, release jitter has the potential to undermine the safety of the protocol. Under the usual definition of EDF, a task that should be released at time t (but is actually released later at time s) is given the absolute deadline of $t + D$ (not $s + D$). Consider a simple example of a two task system, where the tasks share a single resource. The first task (τ_1) has period 10, deadline 10, computation time 5 and jitter 4. The second task (τ_2) has period 22, deadline 20, computation time 5 but no jitter. The shared resource therefore has a deadline floor of 10. At time 20 τ_1 should be released, it will have deadline 30 and will clearly run in preference to τ_2 which, when released at time 22, will have deadline 42. However if the release of τ_1 is delayed by four ticks the following might occur.

- Task τ_2 is released at time 22 and begins its execution.
- At time 23 τ_2 accessed the shared resource and its deadline is reduced to 33.
- At time 24 τ_1 is released with deadline 30 (not 34) and hence preempts τ_2.
- At time 25 τ_1 accessed the shared resource; breaking mutual exclusion.

This is illustrated in Fig. 4. Fortunately this potential problem is easily nullified by a simple modification to the protocol.

The key property of DFP as given in Lemma 1 is that once a task is running it cannot be preempted by any other task that might use the same shared resources. DFP achieves this by giving the running task (when it accesses a resource) a deadline shorter than any such task. With release jitter this can be undermined. The deadline floor value needs to reflect the real relative deadlines of any user task that experiences release jitter.

Consider again the example. Task τ_1 is released at time 24 and has a deadline of 30. In practice its relative deadline is really 6 (not 10). Hence the deadline floor value for the shared resources should be 6. Now when τ_2 accesses the resources at time 23 its deadline will be reduced to 29. So when τ_1 is released for execution at time 24 and is given a deadline of 30 it will not preempt.

To cater for release jitter, the first element of the definition of DFP must become:

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

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

All results given earlier for DFP are easily reasserted for this extended definition. Note that SRP must also be modified to cater for jitter; preemption levels must be assigned according to each task's $D - J$ value.

It must be emphasised that ignoring release jitter, which for fixed priority scheduling can undermine schedulability, can with DFP also lead to mutual exclusion being broken unless a run-time check is made. As all implementations have some jitter over the implementation of the clock it is important that a jitter term is included in all calculations that are used to give the deadline floor values for any shared resource.

5.9 Implementation Issues for Programming Languages and RTOSs

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 at 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. An evaluation of an actual implementation of DFP is given in the next section.

Any implementation of DFP is no more complex than the immediate priority ceiling protocol for fixed priority scheduling which is available via many RTOSs (e.g., it is specified in real-time POSIX) and programming 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 [8]. 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 [22]. 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 [6]: 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.[4]

6 Implementation and Evaluation of DFP in MaRTE OS

MaRTE OS [16] is a real-time operating system developed by the computers and real-time group at the University of Cantabria. Most of its code is written in Ada with some C and assembler parts. MaRTE OS provides support for Ada and C/C++ concurrent applications. In the case of C/C++ applications, they can make use of the POSIX/C interface provided by MaRTE OS [1].

4. The replacement of SRP by DFP for the Ada language was discussed at the 2013 International Real-Time Ada Workshop (IRTAW); the workshop recommended that DFP is adopted and SRP deprecated[21].

MaRTE OS supports many advanced real-time features not present in the POSIX standard. In particular, it implements the EDF scheduling policy along with the Stack Resource Protocol.

In this section we describe two implementations of EDF+DFP and one implementation of EDF+SRP. The implementations differ in the data structures used to implement the ready queue:

- *SRP&Queue+Stack*. SRP implementation based on a queue and a stack.
- *DFP&Queue+Stack*. DFP implementation based on a queue and a stack.
- *DFP&Queue*. DFP implementation based only on a queue.

These three alternative implementations are compared in terms of performance, complexity and code size.

6.1 Basic EDF Implementation

MaRTE OS uses a hierarchical scheduler with two levels. The base scheduler uses fixed priorities as defined in the POSIX standard. Each task (called thread in POSIX) is assigned an integer called the priority that is used as the primary criterion for scheduling. A higher priority task will always preempt a lower priority task. Within each priority level POSIX allows different behaviours (such as FIFO or round-robin ordering). MaRTE OS adds a further secondary scheduler that uses the EDF policy for tasks of the same priority.

The main data structure managed by the MaRTE scheduler is the ready queue. Originally, the MaRTE ready queue is an array of queues implemented with doubly-linked lists, a queue for each system priority.

When a task becomes ready, it must be added to the queue that corresponds to its priority. At each activation, an EDF task is assigned a new absolute deadline. EDF tasks are placed in the queue that corresponds to its priority according to their absolute deadlines (and preemption levels if SRP is in use).

The original structure of the ready queue has been modified to include a deadline sorted queue to support the DFP&Queue implementation and a deadline sorted queue and a stack to support the SRP&Queue+Stack and DFP&Queue+Stack implementations.

The deadline sorted queues have been implemented using a "binary heap". A binary heap is a binary tree with two additional constraints: all levels of the tree, except possibly the last one (deepest) are fully filled, and each node is less than or equal to each of its children according to a *total order* relation. This data structure is an efficient implementation of a sorted queue. Both inserting an element in the heap and removing the element at the head take $O(log(n))$ time, while peeking the head takes constant time $O(1)$.

6.2 SRP Implementation Based on Stack and Queue

In [3], [2] Baker proposed an efficient implementation of the ready queue for a system using the SRP protocol. In this proposal, the ready queue is composed by two data structures: a stack and a queue.

TABLE 2
SRP and DFP Implementation Summary

	SRP	*DFP*
Mutex Fields	2	2
Mutex Operations	4	4
Task Fields	1	0
Task Operations	4	0
Code Lines	54	34

The queue contains all the tasks that have not been yet able to execute in their current activation. They are sorted according to their absolute deadlines, more urgent deadlines first.

The top of the stack contains the running task (the task that is currently executing). Also on the stack, and below the running task, is the task that was preempted when the running task was activated and so on.

All the tasks holding resources are on the stack. Each task has an Effective Preemption Level (EPL) which is the maximum of the task's preemption level and the preemption levels of the resources held by the task.

When a task becomes ready, if its deadline is not more urgent than any other task in the queue, it is inserted in the queue and no scheduling decision is necessary. A decision is needed when:

1) A task becomes ready and has a more urgent deadline than any other task in the queue.
2) The running task suspends.
3) The running task frees a resource.

In situation #1 the candidate to become the new running task is the new task, while in the other two situations it is the task at the head of the queue. This candidate task is then compared with the task at the top of the stack. The candidate task will become the running task if its absolute deadline is more urgent than the deadline of the task at the top of the stack and its preemption level is strictly higher than the EPL of the task at the top of the stack. In such cases the candidate task is pushed into the top of the stack (and removed from the head of the queue in situations #2 and #3). Otherwise, in situation #1 the candidate task is inserted in the queue.

6.3 DFP Implementation Based on Stack and Queue

The DFP can also be implemented as explained above for SRP using a stack and a queue. The DFP implementation is simpler because there is no need for any parameter equivalent to the effective preemption level required by SRP. In the DFP implementation the comparison between the candidate and the task at the top of the stack is simply based on their absolute deadlines.

The implication of this fact on the performance of the system is discussed in Section 6.6.

6.4 DFP Implementation Based on a Queue

The definition of DFP does not introduce any new scheduling rule to an EDF scheduled system. In consequence, the ready queue in an EDF+DFP system can be implemented directly using just a queue sorted by deadline. This queue can be efficiently implemented using a binary heap.

TABLE 3
SRP and DFP Efficient Data Structures

	SRP	*DFP*
Efficient implementation based on queue+stak	Yes	Yes
Efficient implementation based on queue	NO	Yes

It is important to note that this approach cannot be applied to an EDF+SRP system in an efficient way: As explained in Section 3.1 a new scheduling restriction is added by SRP to the basic EDF scheduling: "a task is chosen to execute if it has the earliest deadline *and if its preemption level is strictly higher than the ceiling of any resource that is currently held in the system*".

This restriction forces a deep change in the ordering criterion of the queue. While in plain EDF (and in EDF+DFP), tasks are ordered according to their absolute deadlines, when using SRP the order relation must also take into account the preemption levels of the tasks: when a new task is added to the ready queue it must be placed after all the tasks that are more urgent or have an EPL equal or higher than the new task.

The described order criterion produces a non-total order that prevents the use of efficient implementations of sorted queues such as the binary heap, forcing the use of linear structures with linear time queue and dequeue operations.

6.5 Complexity of the Implementation

To measure the complexity of implementing the SRP and DFP protocols we have counted the number of specific attributes and operations needed for their implementation. Attributes and operations that are common to both protocols are not taken into account.

The implementation of both protocols requires two specific mutex fields. In the case of the DFP these fields are the original deadline of the task holding the mutex (see Section 5.9) and the mutex deadline floor. The specific fields required for the implementation of the SRP are the original preemption level of the task holding the mutex and the ceiling preemption level of the mutex.

For the DFP implementation, two operations have been added to the POSIX operating system interface in order to set and get the deadline floor parameter in the attribute object (`pthread_mutex_attr_t`) used at mutex creation. Two other operations have been added to dynamically set and get the deadline floor of an already created mutex. The implementation of SRP also requires adding four equivalent operations to manage the ceiling preemption level instead of the deadline floor.

The difference in favour of DFP arises when considering the number of specific task fields required. While for the DFP implementation no extra fields are needed, in SRP it is necessary to add the preemption level of the tasks and the corresponding four operations to set and get this parameter either dynamically or in the thread's attribute object (`pthread_attr_t`).

We have also counted the number of lines of code required to implement the protocols. We only consider the lines that are specific to the implementation of each protocol, not including the lines of code related to the basic

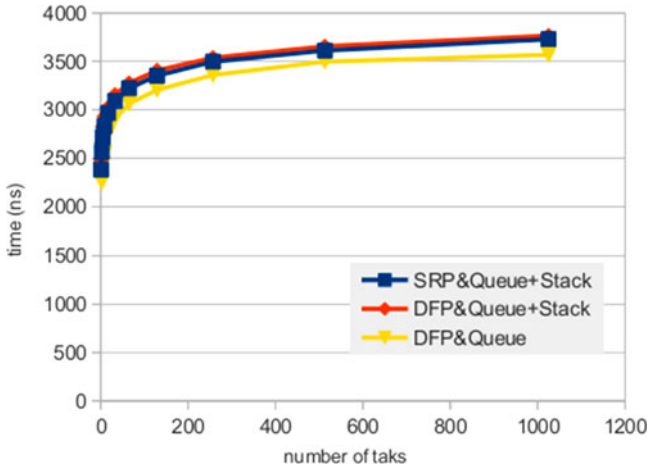

Fig. 5. Time required for a thread to lock and immediately unlock a mutex.

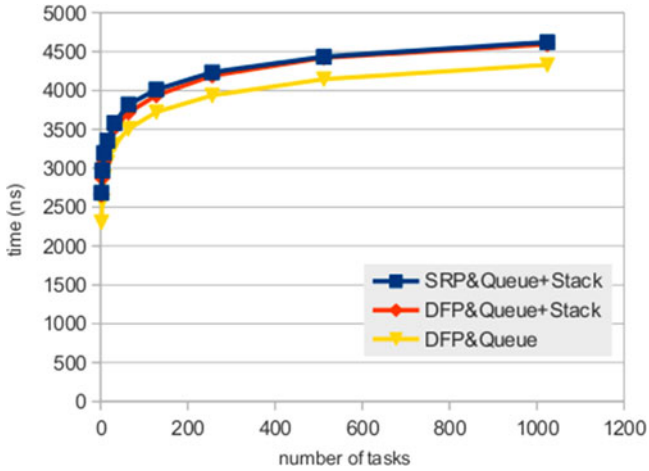

Fig. 6. Sum of the times required by the operating system to activate and suspend a thread.

operation of the locks in MaRTE, nor the lines of code of the data structures used to implement the ready queue in each particular protocol implementation.

In Table 2 we can the see a summary of those implementation metrics. The implementation of DFP is clearly simpler than the implementation of SRP.

Another aspect in which the DFP is simpler than SRP is in the data structure required for the ready queue: while the DFP can be implemented using just a queue, an efficient implementation of the SRP requires a queue and a stack as shown in Table 3.

For our performance analysis, the queue has been implemented using a binary heap and the stack with a chain of singly-linked cells. Both data structures have been implemented with abstract data types using Ada generic packages, with each data structure operation provided via a subprogram.

6.6 Performance Analysis

The performance of SRP and DFP has been analysed in four different situations: lock a mutex, unlock a mutex, thread activation and thread suspension. In order to simplify the presentation of the results, the four situations have been grouped in two representative experiments:[5]

- Time required by a thread to lock and immediately unlock a mutex (Fig. 5).
- The sum of the times required by the operating system to activate and suspend a thread (Fig. 6).

The figures show the performance of each experiment with different numbers of threads in the system. Measurements have been performed in the worst case system configuration for each situation, that is, the system configuration that involves the most complex operations on the ready queue. In particular, they always include enqueue and dequeue operations that affect all the levels of the heap.

All the graphs present a logarithmic behaviour since the most time-consuming operations involved on each measurement are the enqueue or dequeue operations on the binary heap.

5. The experiments were performed on a 800 MHz Pentium III.

Measured times for SRP&Queue+Stack and DFP&Queue+Stack are almost identical in both experiments. The small difference in Fig. 5 in favour of SRP& Queue+Stack is due to the necessity of reading the clock in the DFP lock operation. On the other hand, in Fig. 6 DFP&Queue+stack behaves slightly better than SRP&Queue+Stack due to the fact that in DFP the comparison between the tasks at the head of the queue and at the top of the stack is simpler than in SRP (in SRP not only the deadlines have to be compared, but also the preemption levels).

In both experiments DFP&Queue performs better than SRP&Queue+Stack and DFP&Queue+Stack. This is because, besides the heap queue or dequeue operations, the implementations that use the stack require, in the worst case scenario, the comparison between the the head of the queue and the top of the stack and a push or pop operation on the stack.

7 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. We have also shown that when tasks suffer release jitter it is necessary to decrease all the deadline floors by the release jitter value. When a task accesses a resource at time t its absolute deadline is immediately reduced to the value $t + DF$ (if it is not already less than this value); where DF is 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 IPCP) 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). We have demonstrated the advantage of the DFP rules using the MaRTE real-time operating system.

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

Acknowledgments

The authors would like to thank Sanjoy Baruah for a number of very useful discussions on the topic of this paper. We would also like to thank the anonymous reviewers for recommendations concerning the implementation of the SRP. This work has been funded in part by the Spanish Government and FEDER funds under Grant TIN2011-28567-C03-02 (HI-PARTES).

References

[1] *IEEE Standard for Information Technology - Portable Operating System Interface (POSIX) Base Specifications, Issue 7*, POSIX: IEEE 1003.1-2008, 2008.

[2] T. P. Baker, "A stack-based resource allocation policy for realtime processes," in *Proc. IEEE Real-Time Syst. Symp.*, 1990, pp. 191–200.

[3] T. P. Baker, "Stack-based scheduling of realtime processes," *J. Real-Time Syst.*, vol. 31, pp. 67–99, Mar. 1991.

[4] S. K. Baruah, "Resource sharing in EDF-scheduled systems: A closer look," in *Proc. IEEE Real-Time Syst. Symp.*, 2006, pp. 379–387.

[5] S. K. Baruah, A. K. Mok, and L. E. Rosier, "Preemptive scheduling of hard real-time sporadic tasks on one processor," in *Proc. IEEE Real-Time Syst. Symp.*, 1990, pp. 182–190.

[6] R. Brukardt, ed., "Ada 2005 reference manual," Tech. Rep. ISO/IEC 8526, ISO, 2006.

[7] A. Burns, "A deadline-floor inheritance protocol for EDF scheduled real-time systems with resource sharing," Dept. Comput. Sci., Univ. York, Heslington, U.K., Tech. Rep. YCS-2012-476, 2012.

[8] A. Burns, A. J. Wellings, and T. Taft, "Supporting deadlines and EDF scheduling in Ada," in *Proc. Ada Eur. Conf. Rel. Softw. Technol.*, 2004, pp. 156–165.

[9] M. Chen and K. Lin, "Dynamic priority ceilings: A concurrency control protocol for real-time systems," *J. Real Time Syst.*, vol. 2, no. 4, pp. 325–346, 1990

[10] M. L. Fairbairn and A. Burns, "Implementing and validating EDF preemption-level resource control," in *Proc. 17th Ada Eur Int. Conf. Rel. Softw. Technol.*, 2012, pp. 193–206.

[11] K. Jeffay, "Scheduling sporadic tasks with shared resources in hard-real-time systems," in *Proc. IEEE Real-Time Syst. Symp.*, 1992, pp. 89–99.

[12] B. W. Lampson and D. Redell, "Experience with processes and monitors in Mesa," *Commun. ACM*, vol. 23 no. 2, pp. 105–117, 1980.

[13] J. Y. T. Leung and M. L. Merrill, "A note on preemptive scheduling of periodic real-time tasks," *Inform. Process. Lett.*, vol. 11, no. 3, pp. 115–118, 1980.

[14] C. L. Liu and J. W. Layland, "Scheduling algorithms for multiprogramming in a hard real-time environment," *J. ACM*, vol. 20, no. 1, pp. 46–61, 1973.

[15] I. Ripoll, A. Crespo, and A. K. Mok, "Improvement in feasibility testing for real-time tasks," *J. Real-Time Syst.*, vol. 11, no. 1, pp. 19–39, 1996.

[16] M. A. Rivas and M. G. Harbour, "MaRTE OS: An Ada kernel for real-time embedded applications," in *Proc. 6th Ada Eur. Int. Conf. Rel. Softw. Technol.*, 2001, pp. 305–316.

[17] L. Sha, R. Rajkumar, S. Son, and C.-H. Chang, "A real-time locking protocol," *IEEE Trans. Comput.*, vol. 40, no. 7, pp. 793–800, Jul. 1991.

[18] L. Sha, R. Rajkumar, and J. P. Lehoczky, "Priority inheritance protocols: An approach to real-time synchronisation," *IEEE Trans. Comput.*, vol. 39, no. 9, pp. 1175–1185, Sept. 1990.

[19] M. Spuri, "Analysis of deadline schedule real-time systems," INRIA, Valbonne, France, Tech. Rep. 2772, 1996.

[20] J. A. Stankovic, K. Ramamritham, M. Spuri, and G. Buttazzo, *Deadline Scheduling for Real-Time Systems*, Norwell, MA, USA: Kluwer, 1998.

[21] A. J. Wellings, "Session summary: Locking protocols," *ACM SIGAda Ada Letters*, vol. 33, no. 2, pp. 123–125, 2013.

[22] A. Zerzelidis, A. Burns, and A. J. Wellings, "Correcting the EDF protocol in Ada 2005," in *Proc. 13th Int. Workshop Real-Time Ada* 2007, vol. XXVII, no. 2, pp. 18–22.

[23] F. Zhang and A. Burns, "Schedulability analysis for real-time systems with EDF scheduling," *IEEE Trans. Comput.*, vol. 58, no. 9, pp. 1250–1258, Sept. 2008.

[24] F. Zhang and A. Burns, "Schedulability analysis for real-time systems with EDF scheduling," Univ. York, Heslington, U.K., Tech. Rep. YCS 426, Sept. 2009.

[25] F. Zhang and A. Burns, "Schedulability analysis of EDF scheduled embedded real-time systems with resource sharing," *ACM Trans. Embedded Syst.*, vol. 12, no. 3, pp. 67.1–67.18, 2013.

Alan Burns is a professor of real-time systems in the Department of Computer Science at the University of York, United Kingdom. He has served as a chair of the IEEE Technical Committee on Real-Time Systems and has published widely more than 450 papers and articles, and 10 books. His successful book (co-authored by Andy Wellings), *Real-Time Systems and Programming Languages*, has been published in its fourth edition. In 2009, he was elected a fellow of the Royal Academy of Engineering, United Kingdom. In 2011, he was elected a fellow of the IEEE.

Marina Gutiérrez received the physics degree and a master's degree in computer science by the University of Cantabria. She is a former researcher at the University of Cantabria. Now she works in a private company developing real-time applications for embedded systems. The implementation and testing of DFP in MaRTE OS was part of her master thesis.

Mario Aldea Rivas is an associate professor in the Department of Mathematics, Statistics, and Computer Science at the Universidad de Cantabria. His main research interests include the real-time systems, with special focus on flexible scheduling, real-time operating systems, and real-time languages. He has been involved in several industrial projects to build real-time controllers for robots. He is the main developer of MaRTE OS an operating system that has served as platform to provide support for advanced real-time services.

Michael González Harbour is a professor in the Department of Mathematics, Statistics, and Computer Science at the University of Cantabria. He works in software engineering for real-time systems, and particularly in modelling and schedulability analysis of distributed real-time systems, real-time operating systems, and real-time languages. He is a co-author of *A Practitioner's Handbook on Real-Time Analysis*. He has been involved in several industrial projects using Ada to build real-time controllers for robots. He has participated in the real-time working group of the POSIX standard for portable operating system interfaces. He is one of the principal authors of the *MAST suite for modelling and analysing real-time systems*. He is a member of the IEEE.