Dual Priority Scheduling: A Means of Providing Flexibility in Hard Real-time Systems.

Robert Davis

Real-Time Systems Research Group,
Department of Computer Science, University of York, Y01 5DD, England.

Email: robd@minster.york.ac.uk

ABSTRACT

In this paper, we present a new strategy for providing flexibility in hard real-time systems. This approach, based on *dual* priorities, retains the offline guarantees afforded to crucial tasks by fixed priority scheduling. Further, it provides an efficient method of responsively scheduling soft tasks and a means of providing online guarantees for tasks with firm deadlines. An effective $O(n)$ acceptance test is derived in the paper. Supported by the analysis, acceptance tests and mechanisms described, the dual priority approach provides a basis for combining the benefits of guaranteeing hard requirements with the flexibility inherent in the best-effort paradigm.

1. Introduction

The requirement to support dynamic, adaptive and intelligent behaviour whilst also providing the 100% guarantees needed by crucial hard real-time services, has been identified as one of the key challenges presented by the next generation of real-time systems [8].

In this paper, we present a minimally dynamic scheduling strategy based on *dual* priorities. This strategy provides effective and efficient support for many of the techniques which are now available for enhancing system utility. Further, it retains the predictability afforded to crucial tasks by fixed priority scheduling.

Analysis of fixed priority scheduling has now progressed to the point where it provides an appropriate framework for the construction of many of today's hard real-time systems. The assumptions and restrictions imposed on hard tasks by early work [28, 23] have been lifted, allowing tasks to have deadlines less than their periods [4], deadlines greater than their periods [34], arbitrary offsets [7], precedence constraints [5], multiple priorities [5], shared resources [30] and mode changes [35]. Further, the overheads of the operating system kernel calls have also been considered [10].

Despite these advances, we still expect *spare capacity* to be available at run-time due to guaranteed tasks not taking their worst case execution time, hardware speedups such as cacheing and pipelining and sporadic tasks not executing at their maximum rate [6]. A number of techniques have been developed which enable system utility to be improved by

exploiting this spare capacity. These include, milestone methods [22], sieve functions [22], multiple versions [33] and approximate processing [15, 14]. These techniques can be characterised in terms of the optional or alternate tasks which they give rise to, and further by the timing constraints placed on these tasks. Below, we identify four classes of task.

(1) *Mandatory hard:* Critical tasks which require an *a priori* guarantee that their time constraints will be met. Such tasks are required to have a bounded worst case execution time.

(2) *Alternate hard:* Tasks which may be afforded a 100% guarantee online and then execute instead of a mandatory hard task. These tasks are also required to have a bounded worst case execution time however, this time may be calculated online from the input parameters of a specific invocation.

(3) *Alternate / optional firm:* Tasks with a deadline after which their completion is of no value to the system. Such tasks may have bounded or unbounded worst case execution times and may require either probabilistic guarantees or simply to execute as soon as possible.

(4) *Optional soft:* Tasks with no hard or firm deadline. Typically, these tasks are of most value to the system if they are executed as soon as possible.

To support the above classes of task, offline analysis and online facilities are required which provide:

(1) *A priori* guarantees for mandatory hard tasks.

(2) An efficient online acceptance test which can afford 100% guarantees to alternate tasks.

(3) An efficient online acceptance test which can afford probabilistic guarantees to alternate or optional tasks.

(4) A mechanism for responsively scheduling soft tasks, whilst ensuring that the deadlines on all hard tasks are still met.

In this paper, we show how a dual priority scheduling strategy can be used to meet all four of the above objectives.

Research into online acceptance tests has been carried out by Chetto and Chetto [11] and Schwan and Zhou [27], with respect to earliest deadline scheduling. However, the tests presented in [11] and [27] are pseudo-polynominal in complexity: in the worst case the time taken to perform the tests depends on the number of task invocations within the least common multiple (LCM) of hard task periods. Further, the model used by Schwan and Zhou requires that all task invocations undergo an online acceptance test, leading to unnecessarily high overheads. These factors limit the practical application of the above tests. There are also stability problems in using earliest deadline scheduling for crucial tasks: it is not possible to determine which tasks will miss their deadlines under transient overload. This stability problem has however, been addressed for fixed priority scheduling [29].

Within the context of fixed priority scheduling, an acceptance test has been developed by Ramos-Thuel and Lehoczky [26]. This test assumes that tasks are scheduled using the (static) Slack Stealing algorithm [20]. Unfortunately, the $O(n)$ (where n is the number of tasks) test originally published is insufficient: It may give guarantees to tasks which will then miss their deadlines. Subsequent correction of this test shows it to be of pseudo-polynominal time complexity. Further, the space requirements of the static Slack Stealing algorithm are also pseudo-polynominal: a value needs to be stored for each invocation of each task over the LCM. (Note, sporadic tasks are not catered for by the static Slack

Stealer). The potentially high time and space complexity of this approach makes it unsuitable for many applications.

By comparison, the dual priority scheduling strategy presented in this paper lends itself to the construction of an efficient *O(n)* acceptance test. Whilst this test is not optimal, it is sufficient and effective (see section 5).

Many approaches have appeared in the literature which facilitate the responsive scheduling of soft tasks whilst ensuring that hard tasks still meet their deadlines [29, 21, 31, 32, 20, 13]. Later in this paper, we show that the dual priority approach is a widely applicable method of responsively scheduling soft tasks. Further, we compare its performance with that of other methods such as the Extended Priority Exchange [32], and Slack Stealing algorithms [20, 13].

The dual priority scheduling strategy presented in this paper, was inspired by previous work on dual priorities by Burns and Wellings [9] and the Earliest Deadline Last algorithm of Chetto and Chetto [11]. We note that Harbour *et al* [16] have also considered the scheduling of tasks comprising precedence constrained subtasks with multiple priorities. However, the approach presented here differs, from that investigated by Harbour *et al*: tasks have their priority promoted a fixed interval of time after their release, as in [9]. In essence, the dual priority scheme facilitates the responsive execution of soft tasks by running all hard tasks immediately, when there are no soft tasks to execute, or as late as possible, if there are soft tasks present.

The remainder of this paper is organised as follows. In section 2, we describe the computational model used, our assumptions and notation. Section 3 shows how analysis of fixed priority scheduling also applies to the basic dual priority model, thus providing *a priori* guarantees for mandatory hard tasks. We then follow the techniques given in [4, 34] and [1] extending analysis of the dual priority approach to tasks which exhibit blocking, release jitter and have arbitrary deadlines. A simple sufficient acceptance test for alternate tasks is formulated in section 4. From this test we also derive a probabilistic test for optional firm tasks. The results of simulation studies into the performance of the dual priority approach are presented in section 5. Two metrics are examined: the mean response time of soft tasks and the percentage of alternate firm tasks guaranteed. Comparative results are also provided for background, Extended Priority Exchange and Slack Stealing algorithms. In Section 6, we discuss practical considerations such as execution time overheads. Finally, section 7 offers our conclusions.

2. Computational Model, Assumptions and Notation

The dual priority model considered in this paper is similar to the widely used fixed priority model. We assume that there is a range of unique priorities split into three bands: *Upper*, *Middle* and *Lower*. Any priority level in the upper band is considered higher than any in the middle or lower bands. Crucial (hard) tasks are assigned two priorities, one each from the upper and lower bands. At run-time, other tasks, typically with firm or soft deadlines, are assigned priorities in the middle band.

In the next section, we consider the scheduling of n hard deadline tasks on a single processor. (However, the analysis given is equally applicable to multiprocessor systems with a static allocation of tasks to processors.) Each hard task has an *initial* priority in the lower band and a unique *promoted* priority i where $1 \leq i \leq n$, in the upper band. Thus 1 is the highest and n the lowest priority level in the upper band. Note that the lower band also comprises n priority levels, however, assignment of lower band priorities may be arbitrary: it need not reflect the upper band priority order. For notational convenience, we refer to hard tasks by their upper band priority level, thus task i is the hard task with promoted

priority i. We use $hp(i)$ to denote the set of tasks with a higher promoted priority than i and $lp(i)$ to denote the tasks with promoted priority lower than i. For convenience, the set of all n hard tasks is denoted by $lp(0)$.

Each hard task gives rise to an infinite sequence of invocation requests, separated by at least the minimum inter-arrival time T_i. We assume that invocation requests can arrive at any time after their minimum inter-arrival interval, but be delayed for a variable amount of time, bounded by the maximum *release jitter*, J_i, before being released (i.e. placed in a notional run-queue). Each invocation of task i performs an amount of computation between 0 and C_i (its bounded worst case execution time) and has a deadline D_i measured relative to the time of the request (i.e. arrival). Further, each task has a fixed promotion time U_i, $(0 \le U_i \le D_i)$ measured relative to its release. Upon release, a task i assumes its initial priority (in the lower band), however after U_i time units, its priority steps up to its promoted priority i (in the upper band).

We assume that the cost of scheduling overheads and context switches are subsumed into the worst case execution times of the tasks and further, that tasks are not allowed to suspend themselves.

Finally, the analysis presented in the next section uses the concept of *busy* periods [19], defined as follows: A priority level i busy period is a continuous time interval during which the notional run-queue contains one or more tasks whose priorities have been promoted to level i or higher.

3. Dual Priority Scheduling

In this section, we apply analysis developed for fixed priority scheduling to the basic dual priority model. Hence we show that the worst case response times of hard tasks scheduled using dual priorities can be bounded and their time constraints guaranteed. Initially, we assume that all tasks are independent, do not exhibit release jitter and have deadlines which are less than or equal to their periods. These restrictions are subsequently relaxed, permitting tasks to synchronise with other tasks on the same processor, exhibit release jitter and to have arbitrary deadlines.

3.1. Basic Analysis

First we give a brief description of dual priority scheduling. When a hard task i is first released, it has a priority which is in the lower band. It may be pre-empted by other tasks which have higher, lower band priorities. It may be pre-empted by any soft, optional or alternate task executing at a priority level in the middle band or finally by any hard task executing at an upper band priority. Once time U_i has elapsed, measured from the release of task i, its priority is promoted to the upper band. Once this has occurred, task i can only be pre-empted by other hard tasks which have also had their priority promoted and have a higher upper band priority than i.

To provide an *a priori* guarantee for task i, we seek to obtain its worst case response time. We assume that in the worst case, task i will be unable to execute any computations at its lower band priority, (perhaps due to the execution of soft tasks in the middle priority band). Once promoted to its upper band priority level, task i will only be subject to interference from tasks $j \in hp(i)$ which have also had their priorities promoted. As priority promotion for each task j occurs a fixed time U_j after its release, the worst case response time R_i of task i may be calculated, using analysis derived for fixed priority scheduling [1].

We assume that task i was released at time $-U_i$ and has its priority promoted at time

0. Further, we consider all tasks $j \in hp(i)$ to have been released at their respective values of $-U_j$, hence these tasks also undergo priority promotion at time 0, giving rise to a critical instant [23] for task i. The recurrence relation given below iteratively computes the length w_i, of the priority level i busy period starting at time 0 and culminating in the completion of task i.

$$w_i^{m+1} = C_i + \sum_{j \in hp(i)} \left\lceil \frac{w_i^m}{T_j} \right\rceil C_j \qquad (1)$$

Iteration starts with $w_i^0 = 0$ and ends when $w_i^{m+1} = w_i^m$ or $w_i^{m+1} > D_i - U_i$, in which case task i is unschedulable. (Note, proof that solutions to the above equation exist for task sets with a processor utilisation ≤ 100% is given in [17]). The worst case response time of task i is given by:

$$R_i = w_i + U_i$$

3.2. Synchronisation

We now extend analysis of the dual priority model to permit hard tasks to lock and unlock semaphores according to the Immediate Priority Ceiling Protocol first alluded to by Lampson *et al* in [18], with analysis subsequently given by Rajkumar *et al* in [25]. First we outline the operation of the Immediate Priority Ceiling Protocol and show how it can be applied to tasks with dual priorities. We then present analysis which bounds the worst case response time of each task.

Under the Immediate Priority Ceiling Protocol, each semaphore has an associated ceiling priority which is equal to the highest priority of any task which accesses that semaphore. With the dual priority approach, ceiling priorities are assigned on the basis of promoted (upper band) priorities. At run-time, when a process is granted a semaphore, its priority is immediately increased to the ceiling priority associated with the semaphore. Under the Immediate Priority Ceiling Protocol, the worst case blocking time which an invocation of task i, executing at its promoted priority, can experience due to tasks in the set $lp(i)$ is denoted by B_i. Where B_i is the longest time any task of lower promoted priority than i can lock a semaphore with a ceiling priority of i or higher. Note, we assume that no alternate or soft tasks executing at priorities in the middle band share any semaphores with the hard tasks.

In deriving the worst case response time of task i, we need to consider two cases.

Case 1: Immediately before the priority of task i is promoted, task $k \in lp(i)$, executing at either its upper or lower band priority, locks a semaphore with a ceiling priority higher than i. Note, again we assume that task i has been unable to carry out any computations at its lower band priority level. In this case the worst case response time of task i follows directly from analysis of fixed priority scheduling.

$$w_i^{m+1} = C_i + B_i + \sum_{j \in hp(i)} \left\lceil \frac{w_i^m}{T_j} \right\rceil C_j \qquad (2)$$

Note, the iteration start and completion conditions are the same as equation (1). The worst case response time of task i remains:

$$R_i = w_i + U_i$$

Case 2: Immediately before the priority of task i is promoted, task $j \in hp(i)$, executing at its low band priority, locks a semaphore with a ceiling priority higher than i. In this case,

task i is apparently subject to blocking not accounted for in the derivation of equation (2). However, we can show that equation (2) does in fact still give us the worst case response time of task i:

Due to the operation of the Immediate Priority Ceiling Protocol, only one task which has not yet reached its priority promotion time, can have locked any semaphores. We refer to this task as task j. To find the worst case response time of task i, we consider the situation where task j locks a semaphore immediately before the priority promotion of task i at time 0, Let u_j (≥ 0) be the time at which task j is promoted to its upper band priority. Further, we assume that z_j is the length of time for which task j will continue to hold any semaphores. The worst case time in a priority level i busy period w_i during which task j prevents task i from executing is given below: [1]

$$I_j(w_i) = \min(z_j, u_j) + \left\lceil \frac{w_i - u_j}{T_j} \right\rceil_0^1 (C_j - \min(z_j, u_j)) + \left\lceil \frac{w_i - u_j - T_j}{T_j} \right\rceil_0 C_j$$

As $z_j \leq C_j$ and $u_j \geq 0$, the maximum value of $I_j(w_i)$ occurs when $u_j = 0$, and the above equation reduces to:

$$I_j(w_i) = \left\lceil \frac{w_i}{T_j} \right\rceil C_j$$

The worst case response time of task i can therefore again be found via equation (2).

3.3. Release Jitter

We now relax the restriction that all tasks are released as soon as they arrive. We assume that tasks can arrive at any time after their minimum inter-arrival interval, but may then be delayed for a variable time, bounded by the maximum release jitter J_i, before being released.

As the priority promotion point for each invocation of a task i occurs a fixed time U_i after its release, the analysis given by Audsley *et al* in [1] is directly applicable. The maximum interference $I_j(w_i)$ occurs when an invocation of task j with maximum release jitter J_j has a priority promotion point corresponding to the start of the level i busy period and subsequent invocations of task j have no release jitter. Thus the worst case response time of task i is calculated as follows:

$$w_i^{m+1} = C_i + B_i + \sum_{j \in hp(i)} \left\lceil \frac{w_i^m + J_j}{T_j} \right\rceil C_j \qquad (3)$$

Iteration again starts with $w_i^0 = 0$ and ends when $w_i^{m+1} = w_i^m$ or $w_i^{m+1} > D_i - U_i - J_i$. The worst case response time of task *i*, measure from its arrival, is given by:

$$R_i = w_i + U_i + J_i$$

---

[1] Note, $(x)_0$ is notational shorthand for $\max(x, 0)$, similarly, $(x)^1$ is shorthand for $\min(x, 1)$. Hence $(x)_0^1$ can only take values in the range 0 to 1.

3.4. Arbitrary Deadlines

Here we relax the assumption that task deadlines are no greater than their periods, thus permitting tasks to have arbitrary deadlines ($D_i \leq T_i$ or $D_i > T_i$). Again, by virtue of considering priority promotion at a fixed offset from task release, we are able to directly apply the analysis given by Tindell *et al* in [34]. We assume that earlier invocations of a given task i are executed in preference to later invocations of the same task, even though they share the same upper and lower band priority levels. Note that as earlier invocations always have their priority promoted in advance of later invocations this assumption remains valid with priority promotion.

The analysis given by Tindell, which we reproduce below, examines q ($q = 0,1,2,3...$) level i busy periods to determine the worst case response time of task i. The length of each busy period $w_i(q)$ is found according to the following recurrence relation:

$$w_i^{m+1}(q) = (q+1)C_i + B_i + \sum_{j \in hp(i)} \left\lceil \frac{w_i^m(q) + J_j}{T_j} \right\rceil C_j \qquad (4)$$

the response time of each invocation is then given by:

$$R_i(q) = w_i(q) - qT_i + J_i + U_i$$

Examination of increasing values of q may stop if:

$$w_i(q) \leq (q+1)T_i$$

In which case, the worst case response time of task i is given by;

$$R_i = \max_{q=0,1,2,3...} \left(w_i(q) - qT_i \right) + J_i + U_i$$

3.5. Choosing the Priority Promotion Times.

Our motivation in developing the idea of dual priority scheduling is to provide a strategy which enables system utility to be increased whilst also ensuring that crucial hard tasks are guaranteed to meet their deadlines. To achieve this, we need to facilitate the responsive scheduling of soft tasks and the online guarantee of alternate tasks (both of which are assigned priorities in the middle band). We therefore choose to set the values of U_i as large as possible. To do this requires the worst case response times R_i, to be calculated assuming all priority promotion times are zero (i.e. using exact analysis of fixed priority scheduling). The largest priority promotion times which still result in a feasible task set are then given by:

$$U_i = D_i - R_i$$

We note that using these values of U_i potentially leads to the situation where a small computational overrun by a task j with a high promoted priority may cause its deadline to be missed. However, we argue that this is indicative, not of a problem with dual priority scheduling but either, inaccurate worst case execution time analysis for tasks of promoted priority j or higher, or inaccurate analysis of overheads. In any case, the value of U_j could be reduced to provide an engineered margin for error. Further, the dual priority approach retains the stability property of fixed priority scheduling: computational overrun by a task of low promoted priority (outside of a critical section) cannot cause tasks with higher upper band priorities to miss their deadlines.

It is interesting to note that we have not placed any restrictions on the priority

assignments used within the lower and middle bands. Within the framework of the dual priority model, it is therefore possible to schedule tasks in these priority bands according to dynamic value density or best effort policies [24]. This issue is however, beyond the scope of this paper.

3.6. Example

We now give a simple example of dual priority scheduling using the hard task set detailed in the table below. In our example, tasks i and j are initially released at time t=0. Further, we assume that a soft task A with an execution time of 6 is released at time $t = 1$. Figure 1 shows the scheduling of these tasks under the dual priority scheme. Here, soft task A completes at time $t = 15$. By comparison, under normal fixed priority scheduling, task A would complete at time $t = 22$, (assuming that it was executed at a background priority level). Recall that priority level 1 is the highest, followed by 2,3,4 etc.

Hard tasks					
Name	Period (T)	Deadline (D)	WCET (C)	Priority	Promotion time (U)
i	8	6	2	4 then 1	4
j	12	12	5	5 then 2	3

4. Acceptance test

In this section, we provide an efficient online acceptance test for alternate tasks with hard deadlines. This test is sufficient in that any task admitted by the test is guaranteed to be afforded its requested computation time before its deadline. Further, we show how a probabilistic test can be derived trivially from this sufficient acceptance test.

4.1. Sufficient Acceptance Test

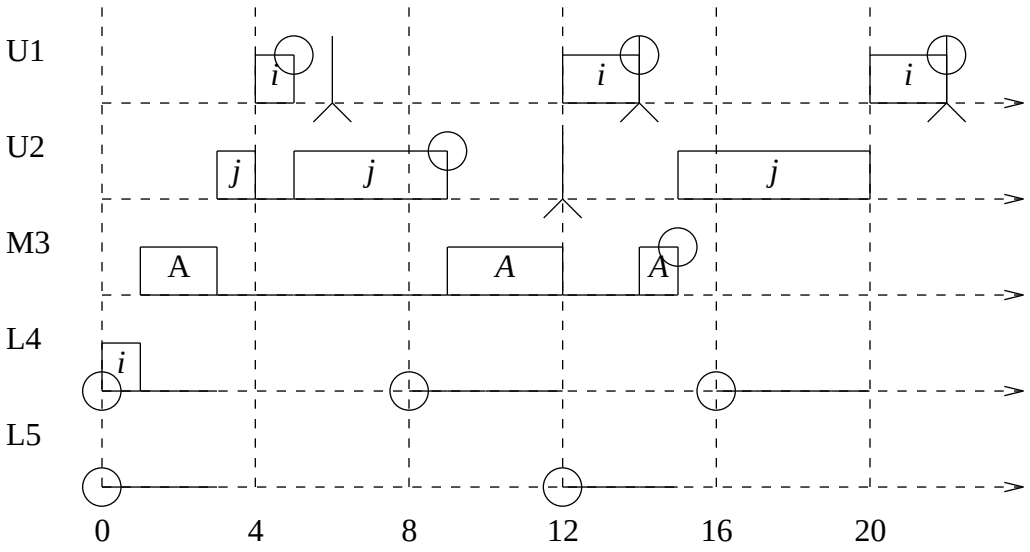
We now derive a lower bound on the execution time afforded to alternate tasks (assigned priorities in the middle band) during some arbitrary time interval. Initially, we assume that alternate tasks are dealt with in FIFO order, thus, at any time, there is only one alternate task which requires a guarantee. Later we show how this assumption can be relaxed.

We assume that at some arbitrary time t, an alternate task A, arrives and requires a guarantee that it will complete C_A units of execution before its deadline at time $t + D_A$. Analysis of the execution time afforded to task A under the dual priority model requires the run-time information given below. We assume that this information may be derived from data which is available within the operating system kernel. Note, we define the current invocation of a hard task i as that invocation which will next utilise the processor. In the case of hard tasks with deadlines greater than their periods, more than one invocation of a task may be runnable at any one time. Here, we require that earlier invocations always take precedence over later ones. Further, the current invocation is defined as the oldest one which is not yet complete.

$c_i(t)$ The remaining execution time budget for the current invocation of task i. This is typically found by subtracting the execution time used from the worst case execution time C_i. Note, if task i is awaiting release at time t, then $c_i(t) = C_i$.

$l_i(t)$ The time relative to t, at which task i was last released. ($l_i(t) \leq 0$).

$x_i(t)$ The earliest possible next release of task i. Again measured relative to t. Typically, $x_i(t) = max(l_i(t) + T_i - J_i\ ,\ 0)$. ($x_i(t) > 0$).



Dual Priority Scheduling:

U1 and U2 are upper band priority levels, M3 is a middle band priority and L4 and L5 are lower band priority levels

t=0, tasks *i* and *j* are released. Task *i* begins executing at its initial priority of L4.

t=1, soft task *A* arrives preempts task *i* and begins executing.

t=3, task *j* is promoted to its upper band priority of U2 and pre-empts soft task *A*.

t=4, task *i* is promoted to its upper band priority of U1 and pre-empts task *j*.

t=5, task *i* completes, task *j* is resumed.

t=8, task *i* is released at its low band priority of L4. It does not therefore preempt task *j*.

t=9, task *j* completes and task *A* is resumed.

t=12, task *i* is promoted to its upper band priority of U1 and pre-empts task *A*. Task *j* is released with priority L5.

t=14, task *i* completes and soft task *A* is resumed.

t=15, soft task *A* completes and task *j* is promoted to priority level U2.

t=16, task *i* is released at its low band priority of L4, again, it does not preempt task *j*.

t=20, task *j* completes, task *i* is promoted to priority level U1

t=22, task *i* completes.

Figure 1: Dual Priority Scheduling.

$u_i(t)$ The time at which the current invocation of task i was or will be promoted to the upper priority band. If at time t, task i is awaiting release, then $u_i(t)$ corresponds to U_i time units after the earliest possible next release $x_i(t)$, of task i. Note that $u_i(t)$ is also measured relative to t and may therefore take both positive and negative values.

$d_i(t)$ The deadline of the current invocation of task i, again measured relative to t.

Note, if task i is awaiting release then $d_i(t) = x_i(t) + D_i$.

$z_i(t)$ The worst case remaining execution time of any critical section which task i is in at time t. Note, $z_i(t)$ is is zero if task i is not within a critical section.

In order to provide an acceptance test for alternate tasks with fixed priorities in the middle band, we need to derive an upper bound on the execution at upper band priority levels during some arbitrary interval. To achieve this, we modify the sufficient and not necessary offline schedulability test given by Audsley, as equation (7) in [2] and discussed in detail in [3], for use at run-time under the dual priority scheme.

Invocations of a hard task i can only normally interfere with alternate task A once their priority has been promoted to the upper band. However, the current invocation of task i may also block task A by executing a critical section at a raised priority under the Immediate Priority Ceiling Protocol. Recall that the Immediate Priority Ceiling Protocol ensures that only one hard task which has not yet had its priority promoted may be in a critical section. An upper bound on the interference $I_i(t,y)$, due to task i, in the interval $[t\ , y + t)$ is thus given by:

$$I_i(t,y) = z_i(t) + \min\left(y - u_i(t)\ ,\ c_i(t) - z_i(t)\right) + f_i(t)C_i + \min\left(\left(y - u_i(t) - (f_i(t)+1)T_i + J_i\right)_0\ ,\ C_i\right) \quad (5)$$

Where, $f_i(t)$ is the number of complete invocations of task i, excluding the current one, which may complete before the end of the interval.

$$f_i(t) = \left\lfloor \frac{y - u_i(t) - C_i + J_i}{T_i} \right\rfloor_0$$

Thus the interference generally comprises four components:

(1) The remainder of the critical section which task i is in, if any.

(2) Computation due to the current invocation of task i once its priority has been promoted.

(3) $f_i(t)$ further invocations of task i which may undergo priority promotion and complete before the end of the interval.

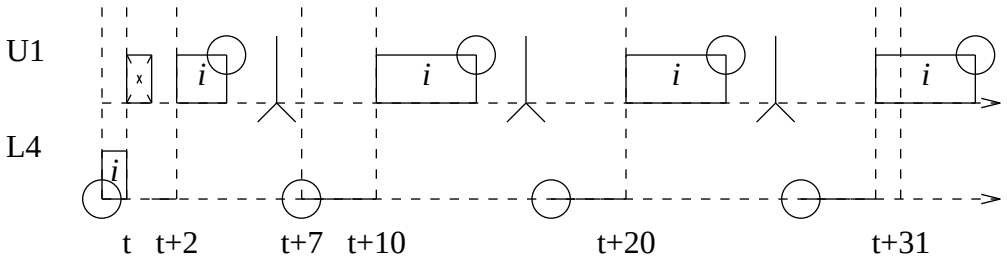
(4) A potentially partially complete final invocation.

These components are illustrated in figure 2. which considers interference due to a single task i detailed in the table below.

Hard tasks						
Name	T	D	J	C	U	Priority
i	10	9	2	4	3	L4 then U1

A lower bound $L_A(t)$, on the execution time available for alternate task A is given by the length of the interval less the upper bound on interference due to all tasks which may execute at an upper band priority level.

$$L_A(t) = \left(y - \sum_{\forall j \in lp(0)} I_j(t,y)\right)_0$$



Acceptance test: Interference over an interval of length $y = 31$.
U1 and L4 are the upper and lower band priority levels of task i.
At time t, we have outstanding computation $c_i(t) = 3$. Further, the remainder of the critical section which task i has just entered is of length $z_i(t) = 1$. The priority promotion time for the current invocation of task i is at $t + 2$, thus $p_i(t) = 2$.
Hence, we have the first two components of $I_i(t,y)$:
$z_i(t) = 1$ and $min(y - p_i(t), c_i(t) - z_i(t)) = min(31 - 2, 3 - 1) = 2$.
Due to release jitter, the earliest the next release of task i may occur is $T_i - J_i$ after the release of the current invocation. This corresponds to time $t + 7$. Hence subsequent invocations of task i may be promoted to the upper priority band at times $t + 10$, $t + 20$, $t + 30$ etc.
The number of subsequent invocations $f_i(t)$ completely contained in the interval is given by: $floor((y - p_i(t) - C_i + J_i)/T_i) = floor((31 - 3 - 4 + 2)/10) = 2$. Hence the third component of $I_i(t,y)$, $f_i(t)C_i = 8$.
The fourth component of $I_i(t,y)$ corresponds to the final, partial invocation of task i within the interval, which begins to interfer at time $t + p_i(t) + (f_i(t) + 1)T_i - J_i = t + 2 + 30 - 2 = t + 30$.
Hence the fourth component of $I_i(t,y)$ is given by $min(31 - 30, 4) = 1$.
The total interference in the interval $[t, t+31)$ due to task i is thus given by: $I_i(t,y) = 1 + 2 + 8 + 1 = 12$.

Figure 2: Acceptance test: Calculating an upper bound on interference.

We note that this approximation is pessimistic: it assumes that all the upper priority band execution of task j that would fall within the interval if only task j were present, will still fall within the interval when all the hard tasks are considered.

Once a value of $L_A(t)$, has been calculated, then task A can be guaranteed provided:

$$L_A(t) \geq C_A \tag{6}$$

The time complexity of the acceptance test formulated as equations (5) and (6) is $O(n)$ where n is the number of hard tasks.

The acceptance test is valid for hard tasks with release jitter and arbitrary deadlines. In the case of tasks which exhibit release jitter, extra interference due to the next invocation being released $T_i - J_i$ after the current one is accounted for in the calculation. In the case of tasks with $D > T$, a number of invocations of the same task may be runnable at any given time. These invocations may have their priorities promoted to the

upper band before or during the interval of interest. The interference due to such invocations is accounted for by the third and fourth components of equation (5).

We note that the formulation given is also sufficient for tasks with $J_i \gg T_i$ (and hence also $D_i \gg T_i$). However, in this case the upper bound on interference found for intervals of less than D_i, may be rather pessimistic. This is because of the inclusion of computation due to invocations which would have had to be released out of arrival order, for them to cause the calculated interference. A tighter bound could be achieved by considering the priority promotion times of each runnable invocation separately, along with the earliest possible release time of each subsequent invocation. However, such analysis is beyond the scope of this paper.

4.2. Multiple Guarantees

So far, we have only considered affording a guarantee to a single alternate task. However, in general, a number of such tasks may be runnable at any given time. We now broaden our approach to cater for this situation. Each alternate task is assumed to be assigned a fixed priority in the middle band according to some policy, perhaps based on its value or deadline. We assume that at time t, each previously guaranteed alternate task A has a remaining computation time budget $c_A(t)$ and a *slack budget* $s_A(t)$. Where the slack budget represents the extent of the interval from time t to the deadline of task A which is not accounted for by the execution of tasks with priorities higher than or equal to that of task A. Further, let $mhp(A)$ be the set of tasks in the middle band with priorities which are higher than that of task A. Similarly, $mlp(A)$ denotes the set of tasks in the middle band with priorities lower than A.

We now modify our acceptance test to account for the presence of other alternate tasks. We assume that once an alternate task has been given a guarantee, this guarantee will not be rescinded.

Consider a new alternate task X. At time t, task X requires C_X units of computation time before its deadline at $t + D_X$. In determining whether task X can be accepted, we must examine both the schedulability of task X and that of lower priority alternate tasks which have already been afforded a guarantee but have not yet completed. Task X can only be accepted if it is schedulable and it will not cause the deadlines of lower priority alternate tasks to be missed.

The acceptance test proceeds as follows: first, a lower bound $L_X(t, D_X)$ on the time available for middle priority band execution in the interval $[t\ ,\ t + D_X)$ is found according to equation (5). The slack budget for task X is then given by:

$$s_X(t) = L_X(t, D_X) - C_X - \sum_{\forall A \in mhp(X)} c_A(t)$$

If $s_X(t) \geq 0$ then task X is schedulable.

If task X is schedulable, then before it can be accepted, we must first check that all previously guaranteed but uncompleted alternate tasks are still schedulable. The execution time budget C_X afforded to task X reduces the slack budget of all lower priority alternates. Provided that

$$\forall A \in mlp(X) : \quad s_A(t) - C_X \geq 0$$

then task X can be accepted. In which case, the slack budgets of all lower priority alternate tasks are reduced:

$$\forall A \in mlp(X) : \quad s_A(t) := s_A(t) - C_X$$

The time complexity of the above acceptance test is $O(n + m)$ where n is the number of mandatory hard tasks and m is the number of previously guaranteed but uncompleted alternate tasks. We note that depending upon the application, it may be useful to re-evaluate the slack budget of previously guaranteed alternates when seeking to accept a new task. This is because gain time, produced when mandatory or alternate hard tasks complete in less than their worst case execution time, may increase the slack budgets. Hence re-evaluating such budgets can improve the acceptance of new tasks.

4.3. Soft Tasks

We now consider adding soft tasks to our model. As with alternate tasks, we assume that soft tasks are assigned priorities in the middle band. Typically, soft tasks are of most value to the system if they are executed responsively. However, we must also ensure that they do not cause hard tasks with online guarantees to miss their deadlines. To achieve this, soft tasks are normally assigned a priority below those of tasks with online guarantees. If more responsive execution is required, then soft tasks may be given an execution time budget at a high priority level (in the middle band). This budget is enforced by the kernel and must not exceed the slack on previously guaranteed alternate tasks. Affording soft tasks such a budget reduces the slack budgets of alternate tasks in the manner described previously. When considering the acceptance of new alternates however, the remaining execution time budget of the soft tasks may be rescinded increasing the slack available and the probability of guaranteeing a new task.

4.4. Probabilistic Acceptance

Generally, not all alternate and optional tasks require a 100% online guarantee. Indeed, many may have unbounded execution times and thus can never be given such a guarantee. Even so, it is often useful to only accept tasks for execution if they can be expected to complete by their deadline. This is particularly true of optional firm tasks which are of no value to the system if they miss their deadlines. We now show how the acceptance test described previously can be trivially modified to accept tasks if they are expected rather than guaranteed to complete.

To provide a probabilistic acceptance test, we require that the following information be provided for use by the operating system kernel. In practice, such expected values can be estimated by monitoring actual task release and execution times.

$E(C_i)$ The expected execution time of a complete invocation of a hard task i.

$E(c_i(t))$ The expected remaining execution time of the current invocation of task i.

$E(T_i)$ The expected inter-arrival time of task i.

$E(C_A)$ The expected execution time of alternate task A.

$E(c_A(t))$ The expected remaining execution time of task A.

Further, let E_F be the expected computation time of an optional firm task F with deadline D_F. We assume that at time t, task F requires a probabilistic guarantee. If afforded such a guarantee, then task F will be given a maximum execution time budget of E_F which is enforced by the kernel.

A lower bound on the expected time available for execution at middle band priorities in the interval $[t\ , t + D_F)$ can be found by substituting expected values for the worst case times C_i, $c_i(t)$, T_i in equation (5). The expected time available for task F, $E(L(t\ , D_F))$, can then be found by further subtracting the expected remaining execution time of each higher priority alternate task A $(A \in mhp(F))$ and the execution budgets of soft tasks

assigned to higher priority levels. Finally, before task F can be accepted, we must check that it will not cause lower priority alternate or optional tasks to receive less than their allotted execution budgets before their deadlines.

4.5. Scheduling Policies

There are many trade-offs in choosing which particular alternate, optional or soft tasks to schedule. Once an alternate task has been accepted, then the semantics of such tasks often dictate that the 100% guarantee afforded at acceptance must not be rescinded. In the case of optional firm tasks, any guarantee is often only probabilistic and could be overturned in order to accommodate a more valuable alternate or optional task. Finally, soft tasks may be required to execute at high priority deferring the execution of alternate and optional tasks provided their guarantees are not violated.

Policies for scheduling such diverse sets of tasks are beyond the scope of this paper. However, the mechanisms, described here, which enable new tasks to be accommodated whilst maintaining both online and offline guarantees, form a basic layer of kernel facilities. These facilities provide the flexibility necessary to enable dynamic policies such as best-effort or value-density scheduling [24] to be used for tasks occupying middle band priority levels.

5. Evaluation

In this section, we evaluate the performance of dual priority scheduling, against two metrics:

(1) The mean response time of soft tasks.

(2) The percentage of a given set of optional firm tasks which may be afforded a 100% guarantee.

The first criteria is used to give a measure of how early spare capacity can be made available, which is a key issue when the utility of a system can be improved by scheduling soft tasks as soon as possible. The second criteria is also dependent on the ability of a scheduling policy to defer the execution of hard tasks, however, it also gives an indication of the efficiency of the acceptance test used.

5.1. Simulation 1: Soft Task Scheduling

To evaluate the performance of the dual priority approach against the first metric, we simulated the scheduling of hard and soft tasks under dual priority, background, Extended Priority Exchange (EPE)[2] [32], and the optimal (dynamic) slack stealing algorithms [13].

The soft task load was simulated by a FIFO queue of tasks, each requiring 1 tick of processing time. The arrival times of the soft tasks followed a uniform random distribution over the test duration (100000 ticks). The number of soft tasks was varied to produce a range of mean processor loadings (shown on the x-axis of the graphs).

The hard task load was simulated using groups of task sets with utilisation levels of 50, 70 and 90%. The results presented in subsequent sections represent the averages over a group of ten task sets.

The hard task sets had an approximately *exponential* distribution of task periods and

[2] This is deemed to be the most effective of a family of bandwidth preserving algorithms which also includes the Deferrable Server [21], Priority Exchange [21] and Sporadic Server algorithms [31].

comprised 12 tasks, 4 with periods in the range 10 to 100 ticks, 4 with periods between 100 and 1000 and a further 4 with periods between 1000 and 10000. Each hard task set was generated as follows. First, the periods of the tasks were chosen at random from the desired range. Deadlines were again randomly chosen, but constrained to be less than or equal to the period. The tasks were then sorted into deadline monotonic priority order which was used to determine both initial (lower band) and promoted (upper band) priorities. Next, random computation times were assigned, highest priority first. The computation times were constrained such that the partial task set remained feasible according to a sufficient and necessary schedulability test. The priority promotion time for each task was set to the maximum value such that the task set was still feasible. Finally tasks sets with a utilisation level differing by more than 1% from that required were discarded.

To simulate sporadic task sets, the same hard task sets were used, however, certain tasks were designated as sporadic. Each task *i* designated as sporadic had the following probability of arrival per tick ($Prob_i^{arr}$) at time *t*.

$$Prob_i^{arr} = \begin{cases} 0 & -\ l_i(t) < T_i \\ \dfrac{1}{T_i} & -\ l_i(t) \geq T_i \end{cases}$$

Where $l_i(t)$ is the time relative to *t* of the previous arrival of task *i*. The mean inter-arrival time of each sporadic task *i* was $2T_i$ compared to a minimum inter-arrival time of T_i.

Once released, all tasks executed for their worst case execution time. Further, the overheads involved in scheduling, maintaining counters for extra capacity or slack, and calculating slack were assumed to be zero.

5.1.1. Results for Hard Periodic Task Sets

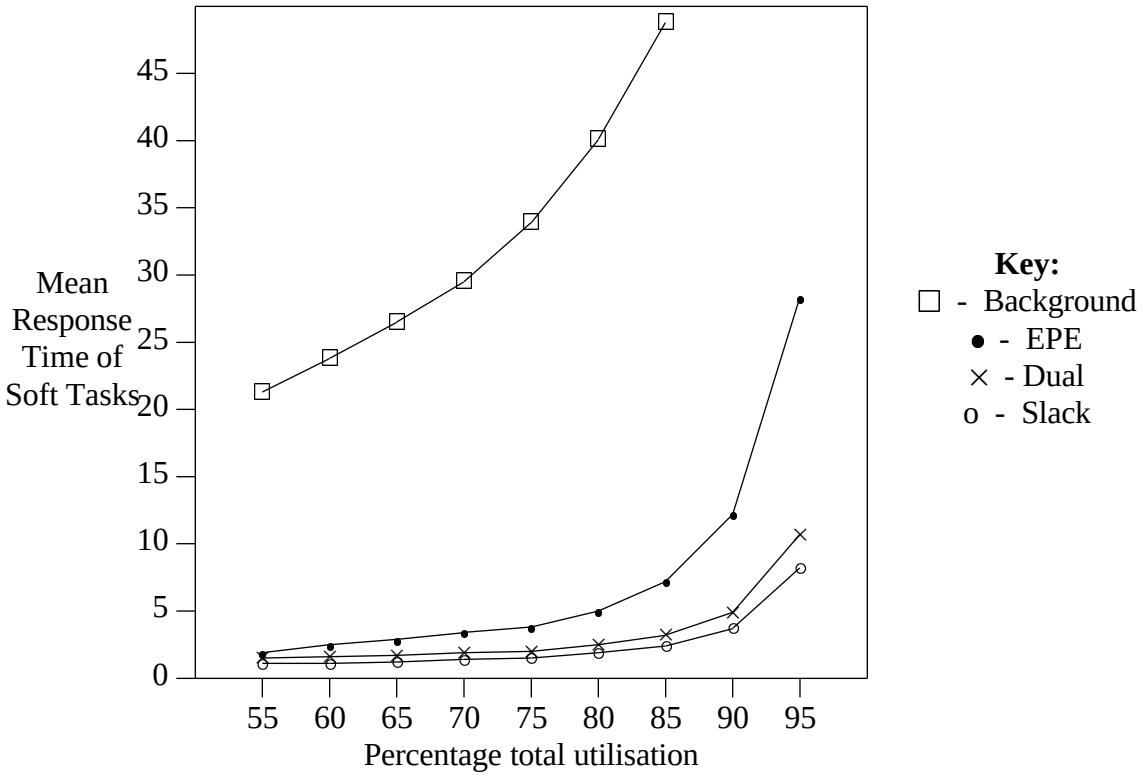
Tests 1, 2 and 3 examined the scheduling of hard task sets with utilisation levels of 50, 70 and 90% respectively. In each case, the soft task load was varied to give the total utilisation levels shown in the graphs. In this series of tests, the performance of the dual priority approach generally exceeded that of the Extended Priority Exchange algorithm. However, at low soft task loads, (e.g. 75% total utilisation for test 2, 91% total utilisation for test 3) the EPE algorithm out-performed the dual priority approach. At such small loads, high priority capacity provided by the EPE algorithm results in highly responsive soft task scheduling. However, as the soft task load increases, this high priority capacity becomes insufficient and the algorithms performance degrades accordingly.

5.1.2. Results for Hard Sporadic / Periodic Task Sets

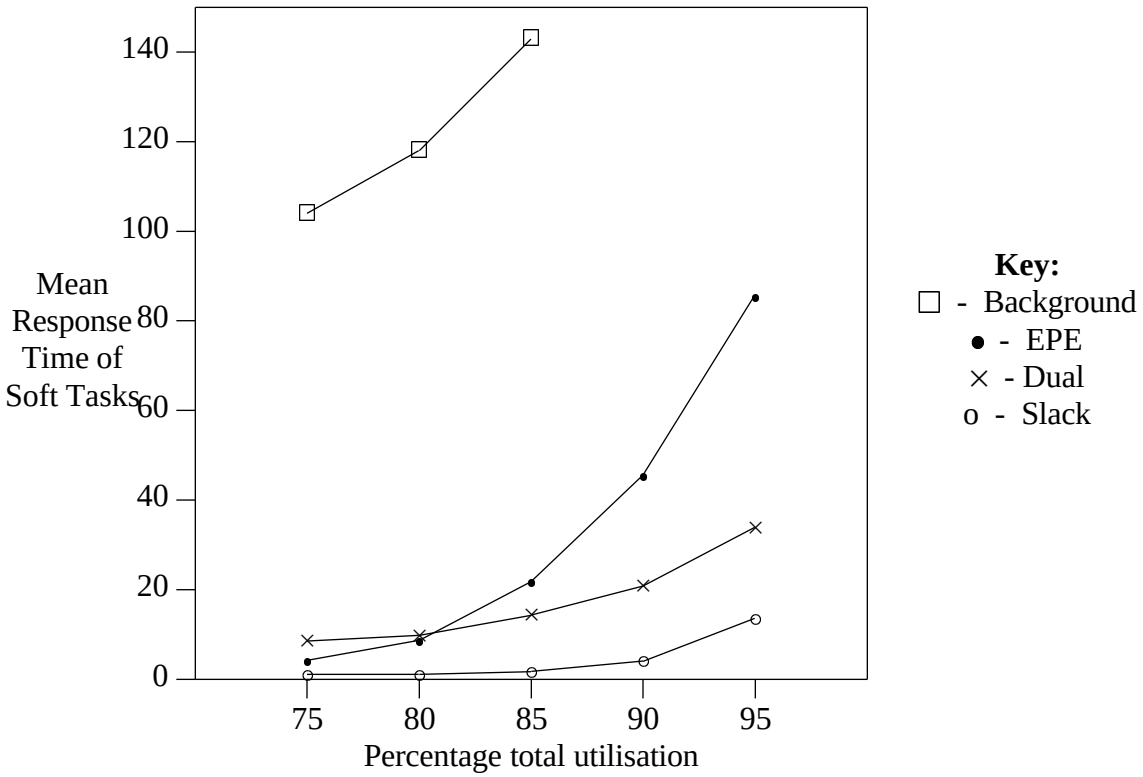
Test 4 examined the scheduling of hard task sets with a worst case utilisation of 90%. In this test, every other task was designated as sporadic and thus the mean hard task utilisation over the test duration was 68%. The soft task load was varied to give the *mean* combined loads shown in the graph. Note, this resulted in a worst case combined load which exceeded 100%.

We recorded the performance of the dynamic slack stealing algorithm assuming that the exact slack at each priority level was re-evaluated every 10 ticks (Slack-10), giving close to optimal performance. (Note, in practice, this would result in very large overheads, although such overheads were ignored in our simulation).

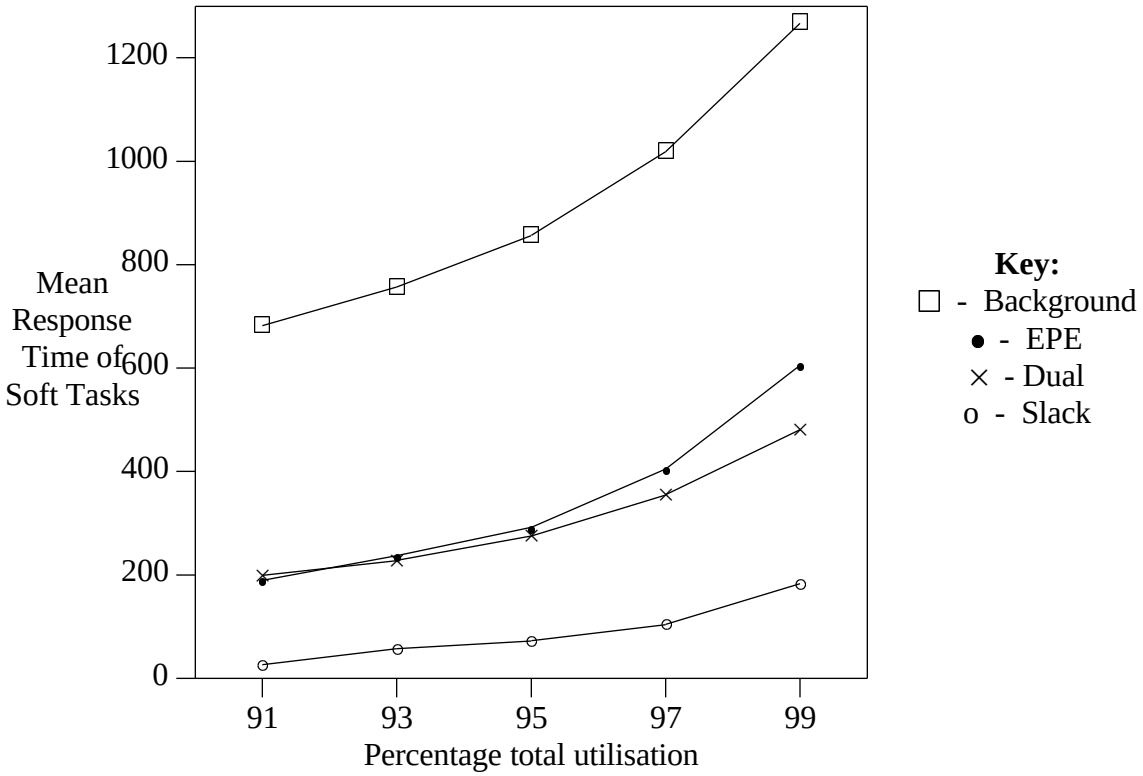
In this test, the performance of the dual priority approach significantly exceeds that of the Extended Priority Exchange and background scheduling methods.



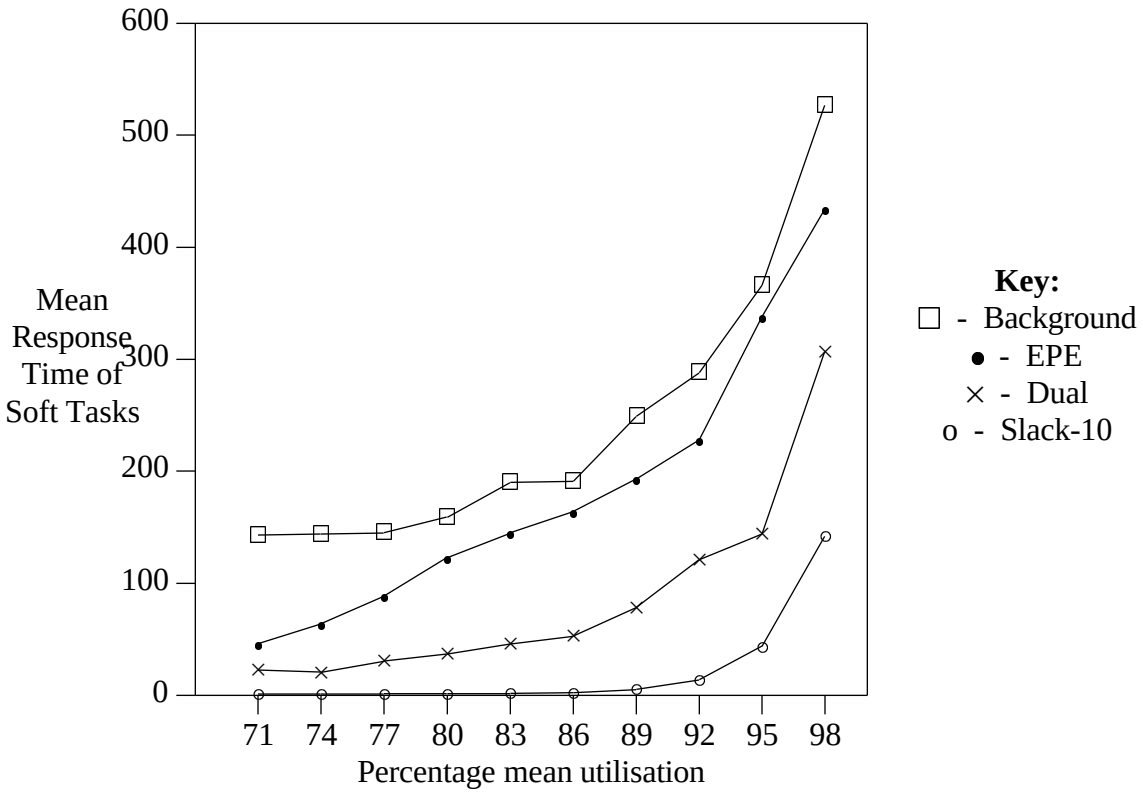
Test 1: Exponential hard task sets, 12 tasks, 50% utilisation



Test 2: Exponential hard task sets, 12 tasks, 70% utilisation



Test 3: Exponential hard task sets, 12 tasks, 90% utilisation



Test 4: Exponential hard task sets, 12 tasks, 90% utilisation
Tasks 1,3,5,7,9,11 sporadic.

5.2. Simulation 2: Firm Task Acceptance

To evaluate the performance of the dual priority approach against our second metric, we simulated the scheduling of hard and firm tasks. The sufficient acceptance test given in section 4 was used to determine if each firm task could be guaranteed. Further, only firm tasks which had been given a guarantee were allowed to execute.

For comparison purposes, we also simulated scheduling and acceptance tests for background, Extended Priority Exchange and Slack Stealing algorithms. The acceptance tests provided for these methods were exact in that only firm tasks which would miss their deadlines were rejected by the tests. Note, these exact tests were of pseudo-polynominal complexity. For further comparison, we also provided an exact test for the dual priority approach. In the case of 50 and 70% utilisation hard task sets, the results for this exact test differed little from those for the $O(n)$ test. For reasons of clarity, only results for the approximate acceptance test were plotted in these cases.

The hard task sets used in this series of simulations were the same as those used in tests 1, 2 and 3. That is, they had an approximately exponential distribution of periods between 10 and 10000 ticks and a worst case utilisation of 50, 70 or 90%.

The firm task load was simulated by a queue of firm tasks each with a given relative deadline and requiring some execution time, corresponding to a fixed proportion of that deadline. The arrival times of these tasks followed a uniform random distribution over the test duration (100000 ticks). The execution requirement and relative deadline of the firm tasks were varied to examine the effect on the percentage of tasks accepted. In each case the number of firm tasks in the queue was set such that the combined utilisation of all hard and firm tasks was 100%. The queue of firm tasks was ordered by arrival time prior to commencing the simulation. During each simulation run, once the arrival time of the firm task at the head of the queue had been reached, it was removed from the pending queue and subject to an acceptance test. If accepted, the task was placed on the firm task run-queue and executed under the appropriate scheduling algorithm. (Note, the firm task run-queue was ordered by absolute deadline: earliest deadline first).

Our initial simulations examined the acceptance and scheduling of firm tasks with relative deadlines of 10, 100 and 1000 ticks and execution requirements corresponding to 10 - 90% of their deadlines (plotted on the x-axis of the graphs).

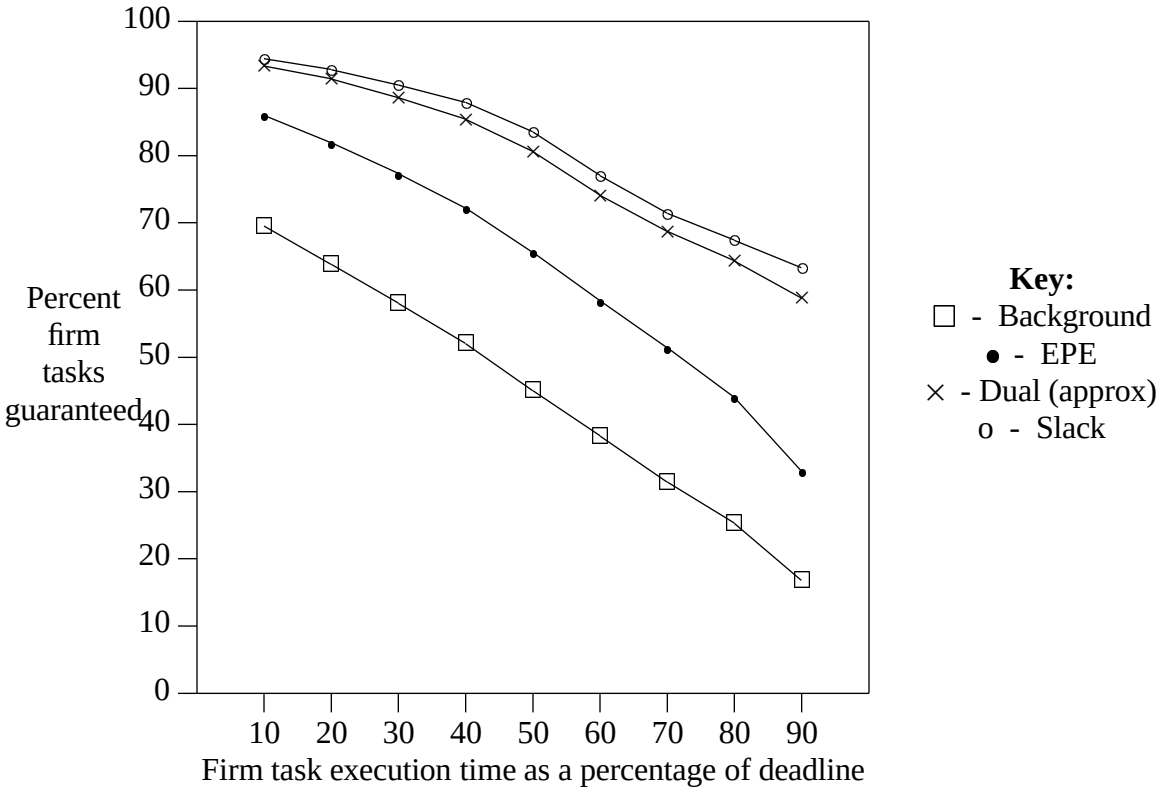
Finally, we conducted a further series of simulations using sets of firm tasks with exponentially distributed deadlines. Again the execution requirements of each firm task corresponded to a fixed proportion of its deadline. In all of these tests, scheduling and acceptance test overheads were assumed to be zero.

5.2.1. Results for Firm Tasks with Short Deadlines

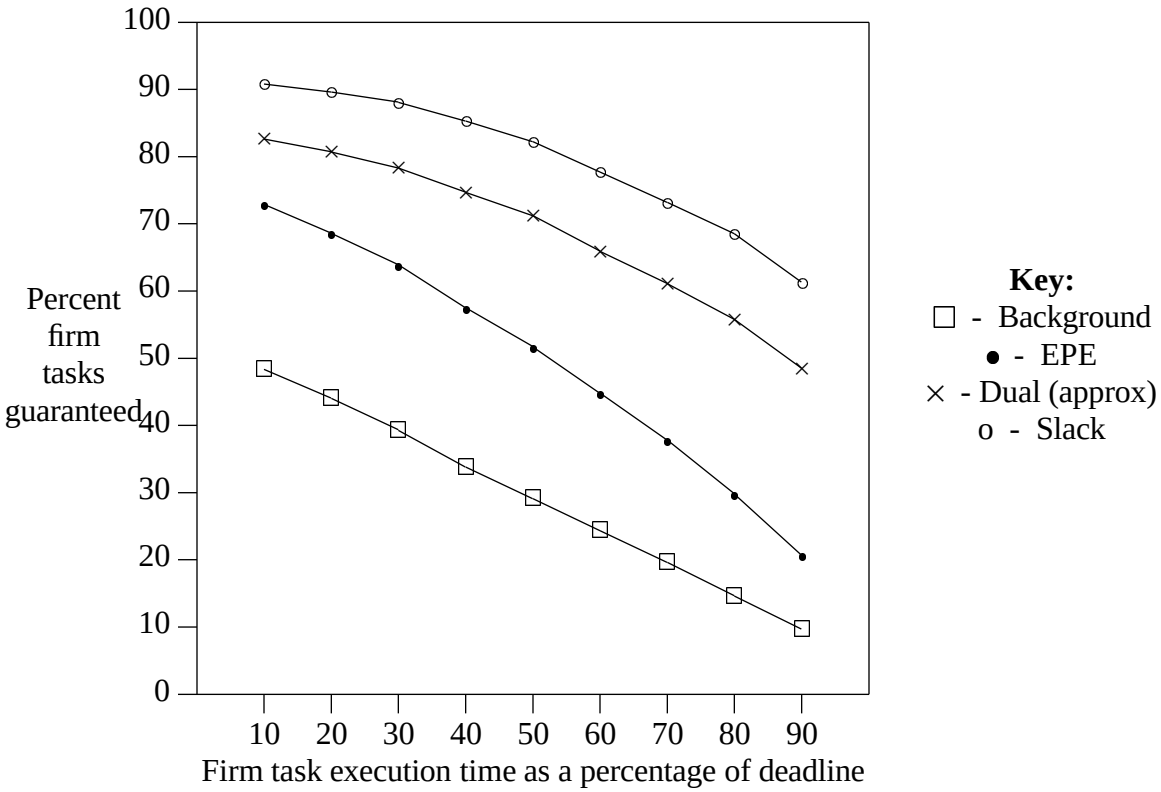
In tests 5,6 and 7, we examined the percentage of firm tasks with a deadline of 10 ticks which were accepted and scheduled by the dual priority, background, Extended Priority Exchange and Slack Stealing algorithms.

In this series of tests, the deadline on each firm task was less than the period of any of the hard tasks. We therefore expected the results to reflect the ability of each scheduling method to immediately provide spare capacity for firm tasks.

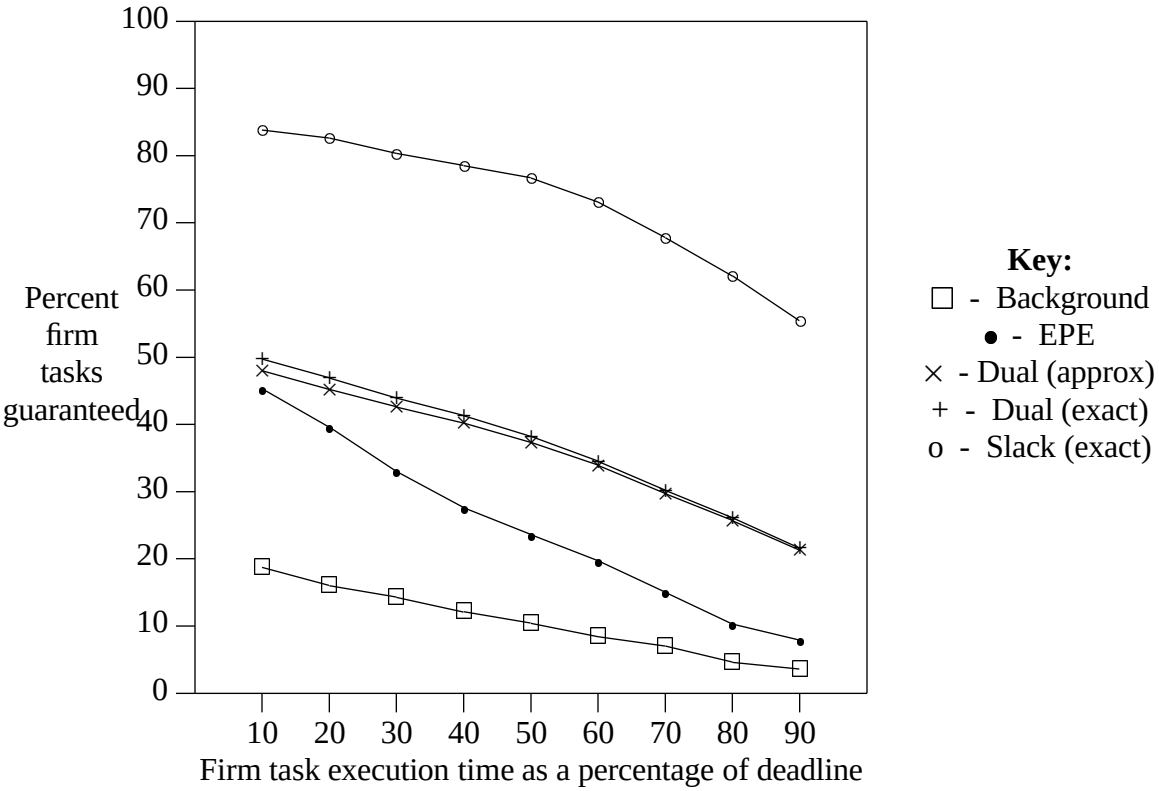
In each of tests 5,6 and 7, the slack stealing algorithm combined with its associated exact test resulted in the highest percentage of firm tasks being guaranteed. The dual priority approach using the approximate test derived in section 4 provided consistently better performance than the Extended Priority Exchange algorithm and its associated exact test.



Test 5: Exponential hard task sets, 12 tasks, 50% utilisation
Firm task deadline = 10.



Test 6: Exponential hard task sets, 12 tasks, 70% utilisation
Firm task deadline = 10.



Test 7: Exponential hard task sets, 12 tasks, 90% utilisation
Firm task deadline = 10.

5.2.2. Results for Firm Tasks with Medium Deadlines

In tests 8,9 and 10, we examined the percentage of firm tasks with a deadline of 100 ticks which were accepted and scheduled by the various methods. Recall that 33% of the hard tasks had periods in the range 10 to 100.

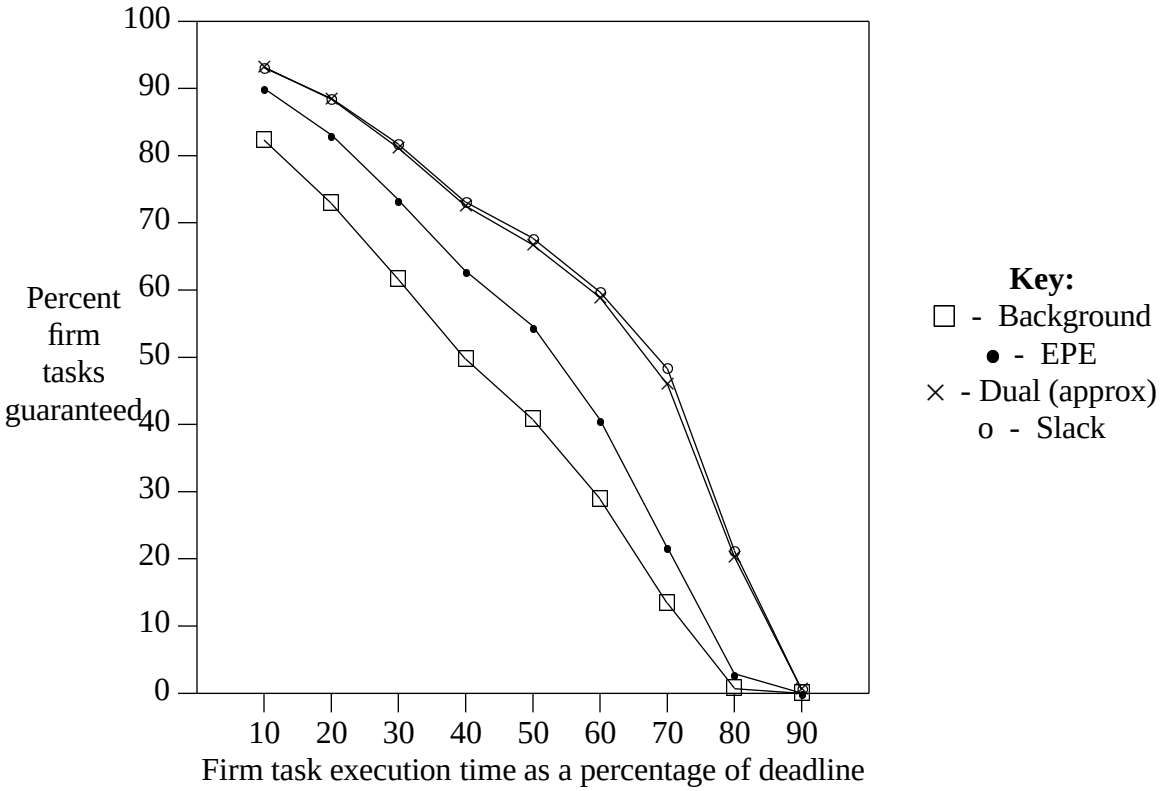
In each of these tests, the slack stealing algorithm combined with the appropriate exact test again resulted in the highest percentage of firm tasks being guaranteed. The dual priority approach provided a similar level of performance for hard task sets with a utilisation of 50% and significantly better performance than the Extended Priority Exchange algorithm (and its associated exact test) at all hard and firm task utilisation levels.

It is noticeable that the graphs for this series of tests differ in shape from those for firm tasks with a short deadline. This is because in the latter case there are hard tasks with periods less than the firm task deadline. This means that there is always some hard task execution which cannot be deferred until after the deadline of the firm task. Hence the percentage of firm tasks which can be guaranteed by even the most efficient methods drops to zero when each firm task has a high utilisation.

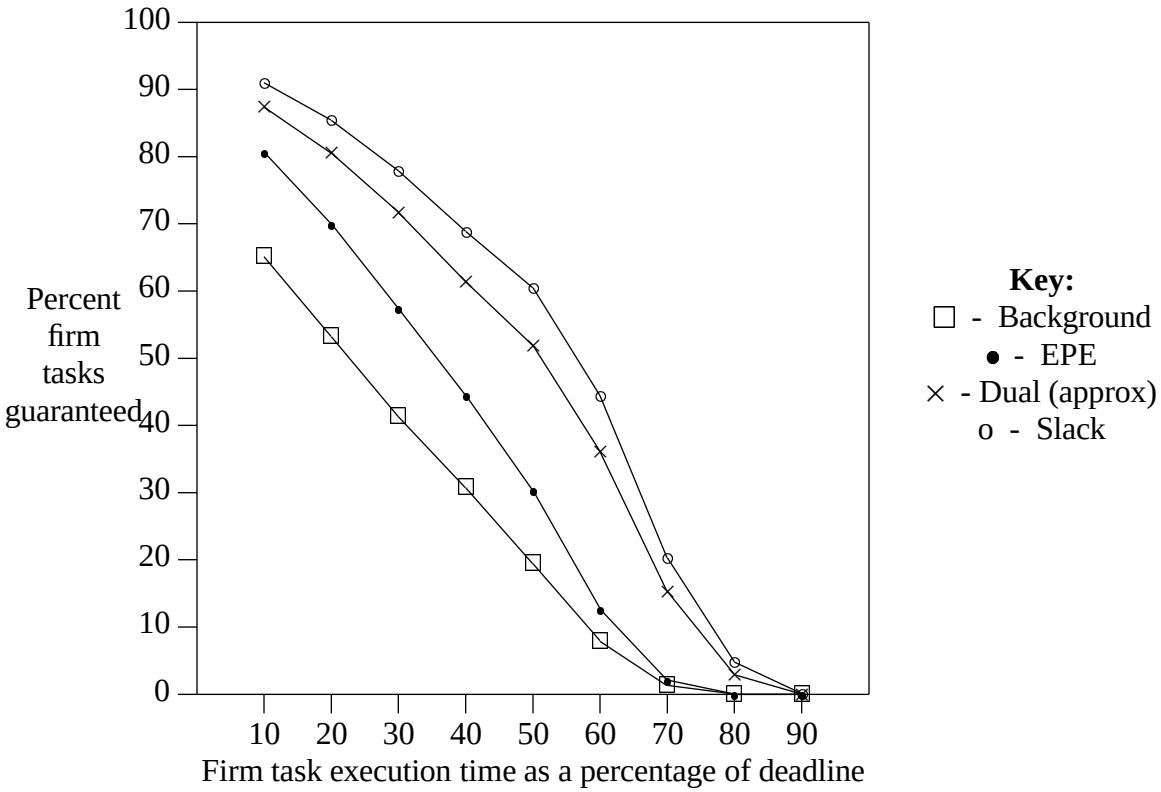
5.2.3. Results for Firm Tasks with Long Deadlines

Tests 11,12 and 13 examined the percentage of firm tasks with a deadline of 1000 ticks which were accepted and scheduled by the various approaches.

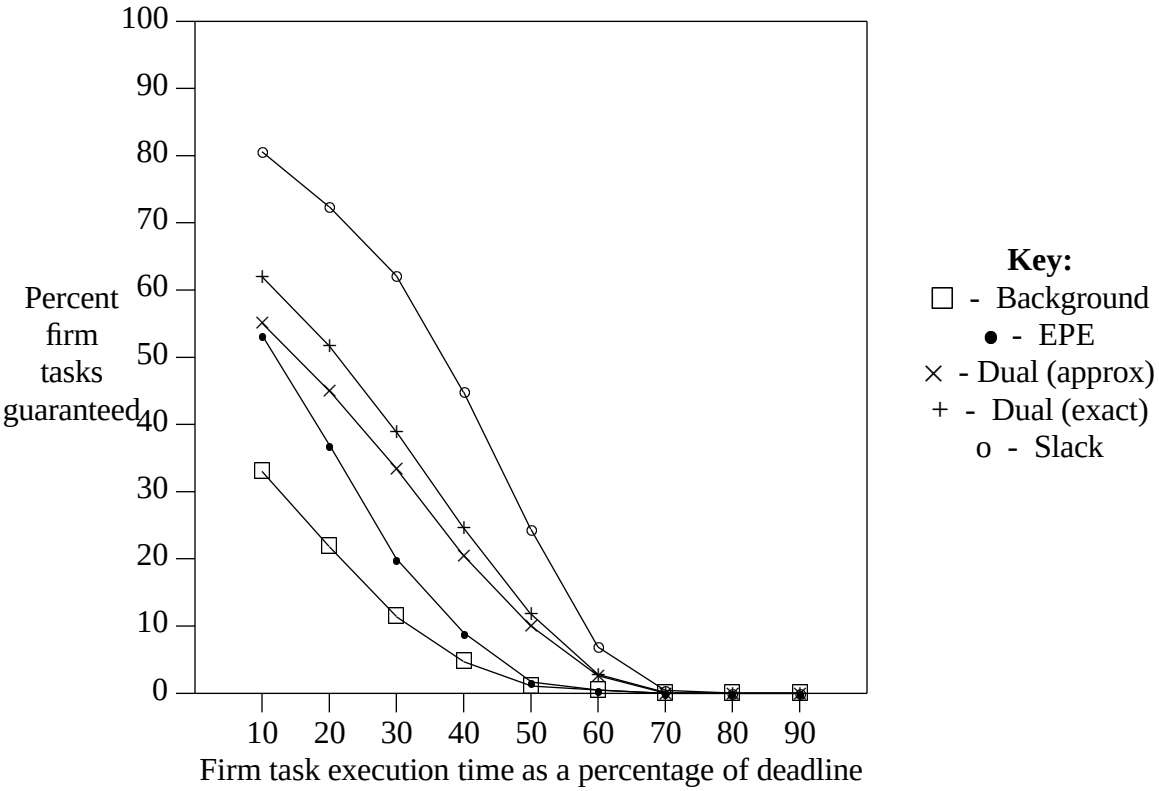
With firm task deadlines greater than many of the hard task periods, and an order of magnitude greater than some, there is always a large proportion of hard task execution which cannot be deferred until after the deadline of the firm task. Hence there is a distinct cut-off point in terms of firm task utilisation after which none of the methods can



Test 8: Exponential hard task sets, 12 tasks, 50% utilisation
Firm task deadline = 100.



Test 9: Exponential hard task sets, 12 tasks, 70% utilisation
Firm task deadline = 100.



Test 10: Exponential hard task sets, 12 tasks, 90% utilisation
Firm task deadline = 100.

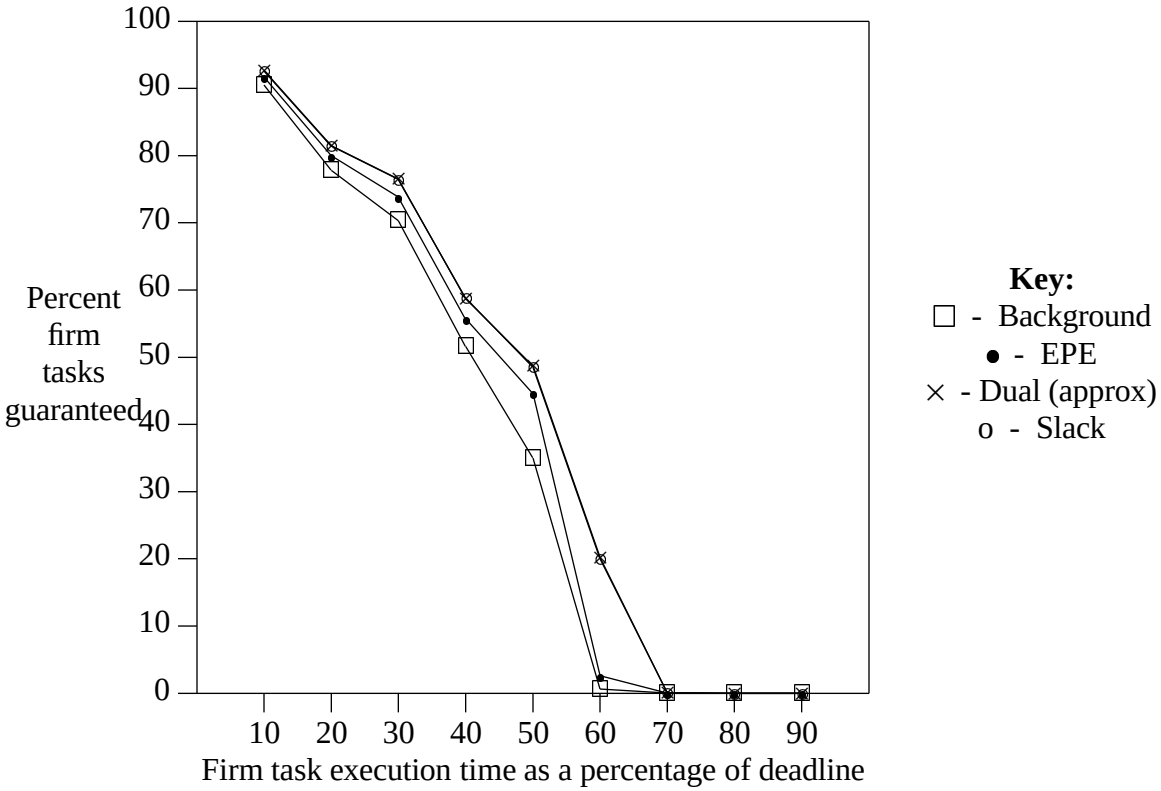
guarantee any firm tasks. However before this point, the slack stealing and dual priority approaches still noticably outperform both the Extended Priority Exchange and background scheduling methods.

5.2.4. Results for Firm Tasks Sets with an Exponential Distribution of Deadlines

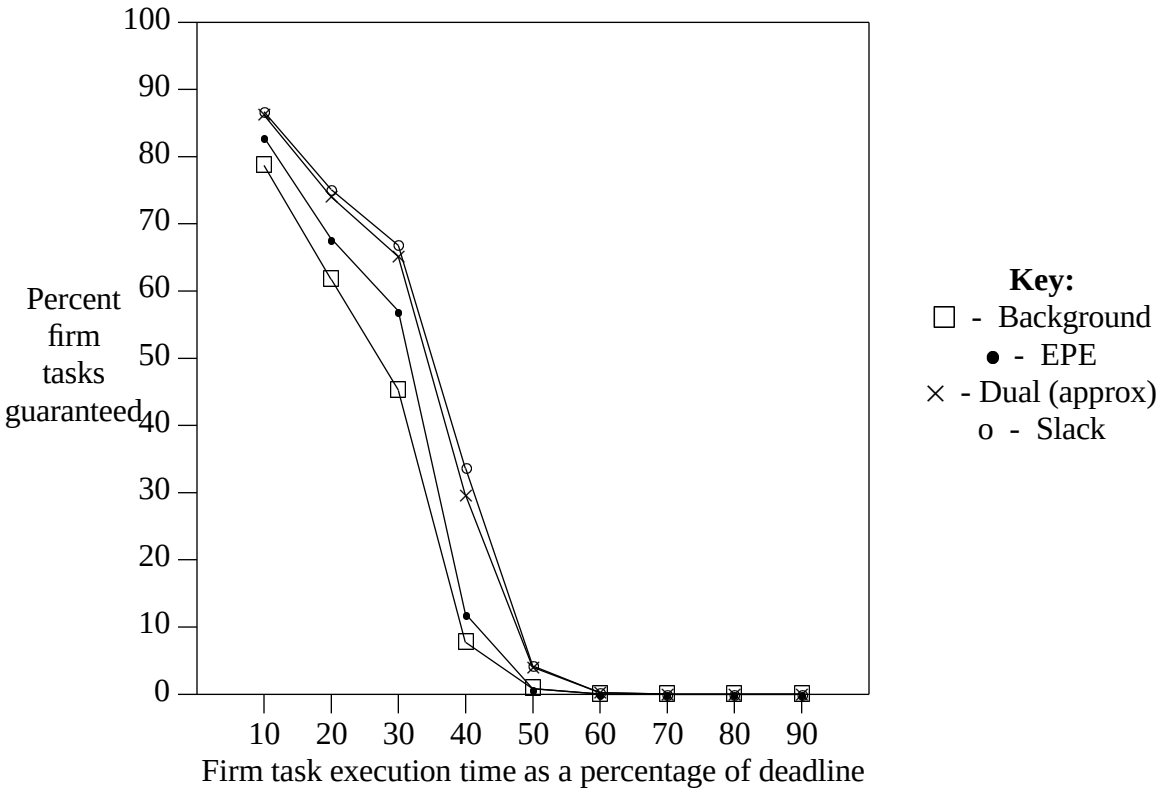
Tests 14,15 and 16 examined the percentage of firm tasks with an exponential distribution of deadlines in the range 10 to 1280 ticks (mean = 60) which were accepted and scheduled by the various approaches. Again, the execution time of each firm task corresponded to a fixed proportion of its deadline. This proportion was varied to examine the effect on the number of firm tasks guaranteed and is plotted on the *x*-axis of the graphs. The percentage of total firm task execution time guaranteed is plotted on the *y*-axis.

In these tests, the mean firm task deadline was 60 ticks, thus for low firm task utilisation levels (< 100% - hard task set utilisation) the graphs resemble those for earlier tests with medium firm task deadlines (100 ticks). However, as firm task execution time is increased some of the short deadline (e.g. 10 ticks) firm tasks can still be guaranteed. Thus the percentage of firm task execution guaranteed tails off more gradually than was the case in tests 8, 9 and 10.

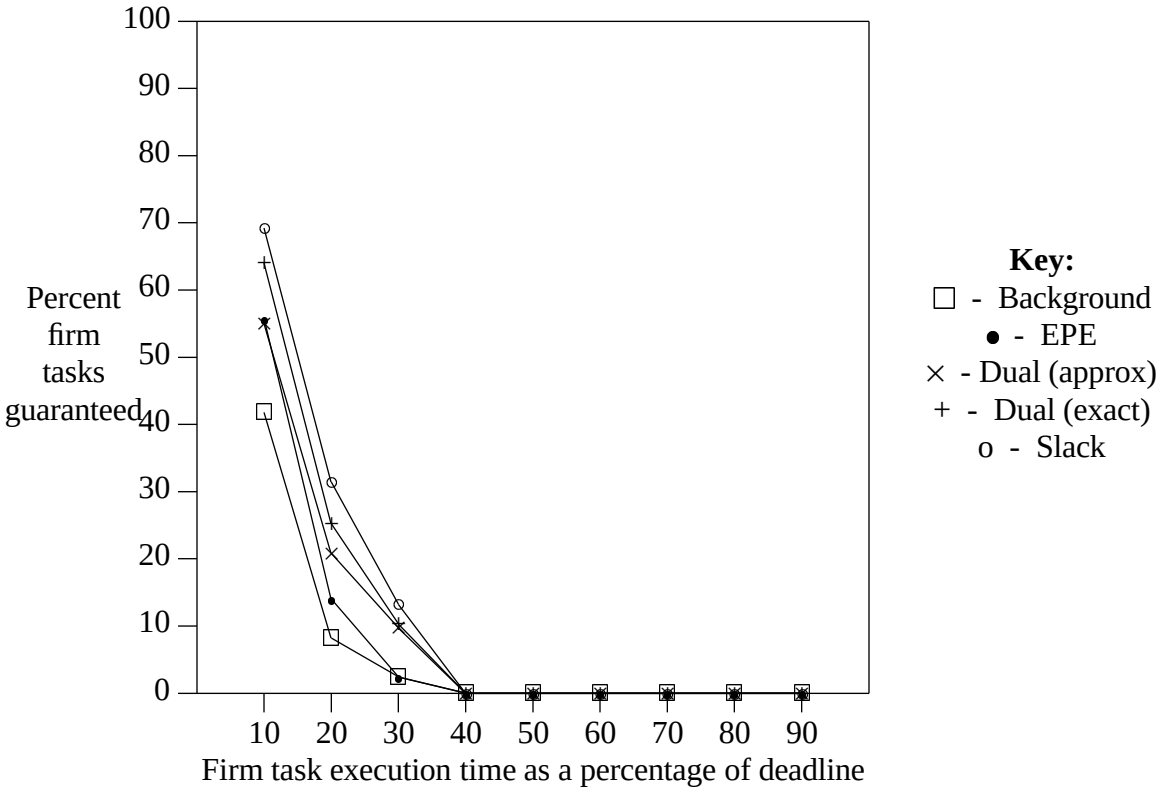
Again, the Slack Stealing algorithm supported by an exact test gives the best theoretical performance over the entire range of hard and firm task loads examined. The Extended Priority Exchange algorithm (and associated exact test) performs well when firm task execution time is a small proportion of deadline. However, its performance tends to that of background at high firm task utilisation levels. This can be attributed to the diverse priorities at which EPE makes spare capacity available, with only a small amount generally available at high priority levels. The dual priority approach generally provides improved performance over the EPE approach.



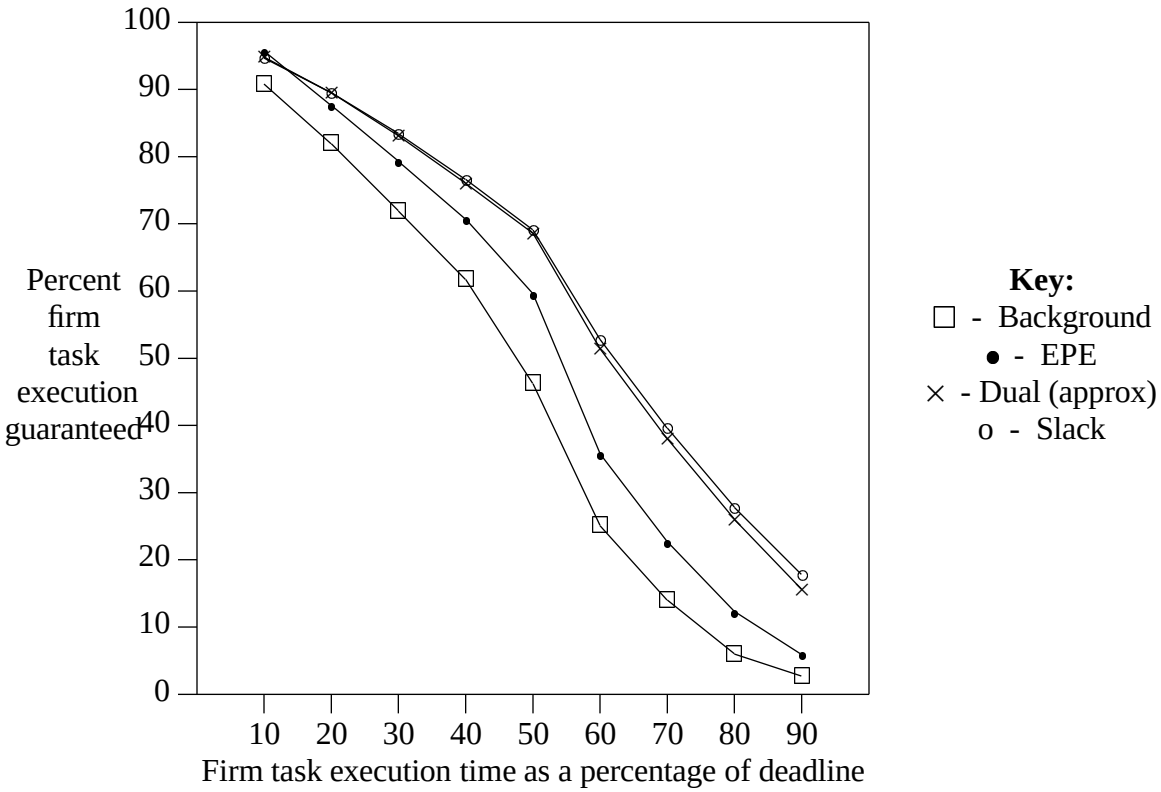
Test 11: Exponential hard task sets, 12 tasks, 50% utilisation
Firm task deadline = 1000.



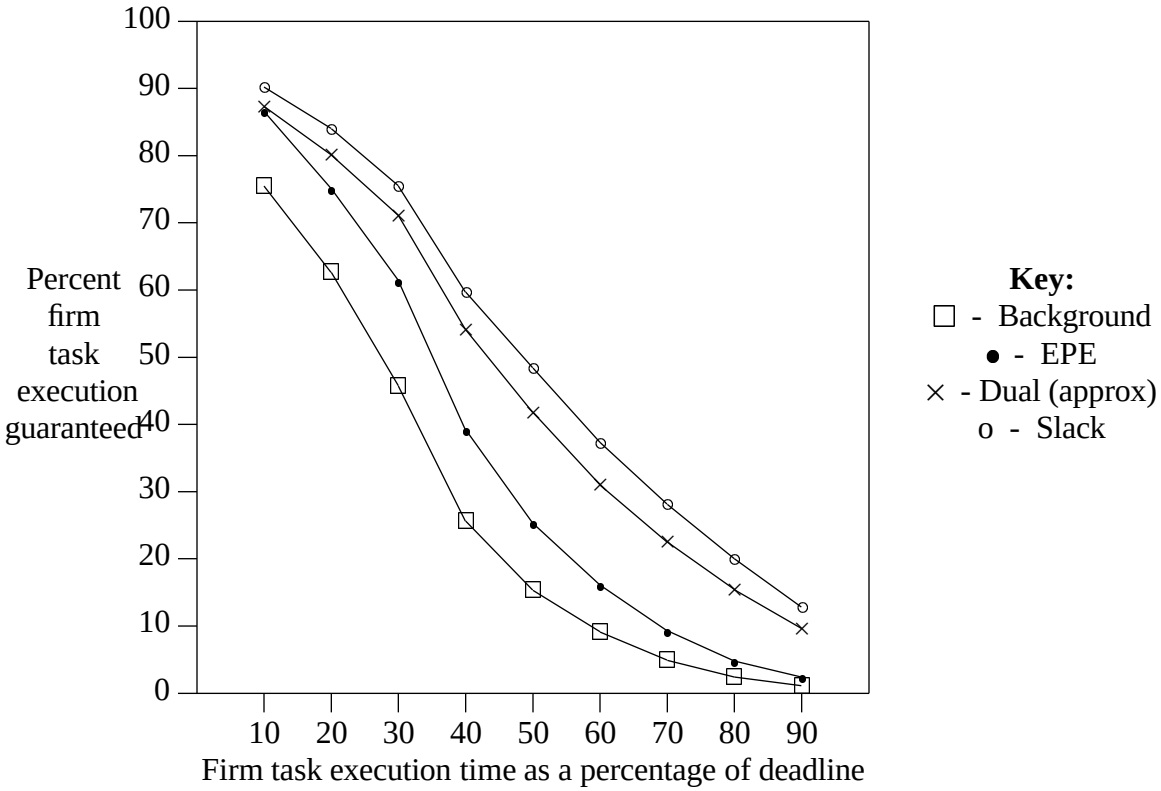
Test 12: Exponential hard task sets, 12 tasks, 70% utilisation
Firm task deadline = 1000.



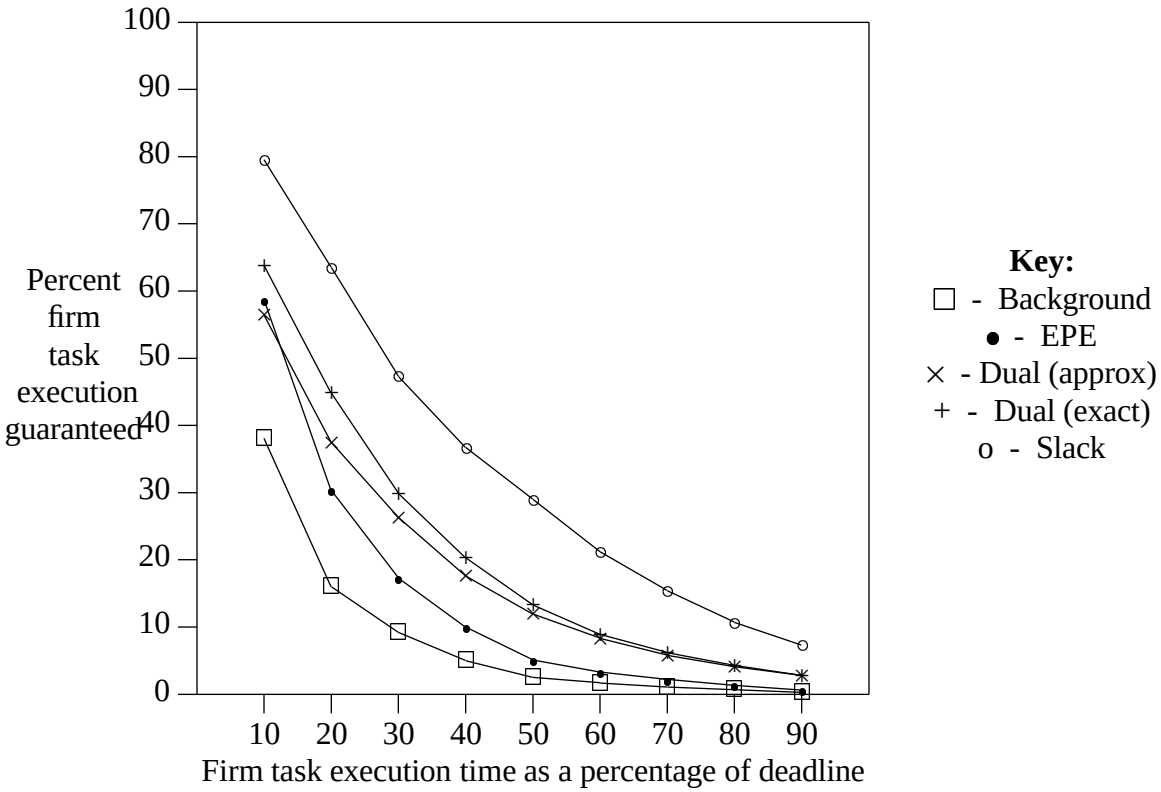
Test 13: Exponential hard task sets, 12 tasks, 90% utilisation
Firm task deadline = 1000.



Test 14: Exponential hard task sets, 12 tasks, 50% utilisation
Exponential distribution of firm task deadlines.



Test 15: Exponential hard task sets, 12 tasks, 70% utilisation
Exponential distribution of firm task deadlines.



Test 16: Exponential hard task sets, 12 tasks, 90% utilisation
Exponential distribution of firm task deadlines.

5.3. Summary

From the results of our simulations, it is clear that the dynamic slack stealing algorithm in conjunction with its associated exact acceptance test, can theoretically guarantee the largest percentage of firm tasks execution. This is perhaps not unexpected, as the slack stealing algorithm has been shown to be optimal in terms of minimising the response times of soft tasks [20, 13] and is thus in some senses an optimal method of deferring hard task execution.

The performance of the dual priority approach allied with the $O(n)$ acceptance test given in section 4, was seen to exceed that of the Extended Priority Exchange algorithm supported by an appropriate exact test. Further, performance was close to that of the slack stealer for hard task sets with low (e.g. 50%) utilisation levels.

Recall that in our simulations it was assumed that all scheduling overheads, including the performance of acceptance tests and the calculation and maintenance of data describing extra capacity or slack, were zero. However in practice these overheads may have an impact on the effectiveness of a particular technique. These issues are discussed in the next section.

6. Practical Considerations

In considering the practical application of a particular scheduling approach and its associated acceptance test, it is necessary to examine the overheads involved and any restrictions which the approach places on hard tasks.

Dual priority scheduling places few restrictions on hard tasks. Indeed, in section 3, we showed how the feasibility of sporadic and periodic hard tasks which exhibit blocking, release jitter and have arbitrary deadlines could be determined using the technique of response time analysis. In terms of overheads, dual priority scheduling results in an extra release type event at the promotion point of each invocation of a task. This event requires that the priority of the task is increased and hence its position in the run-queue altered accordingly. In the worst case, this operation takes $O(N)$ time, where N is the number of tasks in the run-queue. The acceptance testing of a task also incurs an overhead which is $O(n + m)$, where n is the number of hard tasks and m the number previously guaranteed but as yet uncompleted firm tasks. Thus the run-time overheads involved in dual priority scheduling are relatively low, making the approach suitable for many practical systems.

Although offering higher theoretical levels of performance, the slack stealing approach suffers from a number of problems in terms of its practical application. There are two distinct methods of implementing slack stealing: static and dynamic. The former relies on a table of values recording the slack on each invocation of a hard task over the LCM. Unfortunately, this restricts its applicability as hard tasks are not allowed to arrive sporadically, nor to exhibit release jitter. Further, the LCM must be sufficiently small that the memory requirements of the table are not excessive. The dynamic method overcomes these restrictions by virtue of calculating slack online. However, this increases overheads. The approximate dynamic slack stealing algorithm described in [12] periodically requires $O(n^2)$ time to calculate the slack. In addition to this, both static and dynamic methods require $O(n)$ time at each context switch to maintain slack counters.

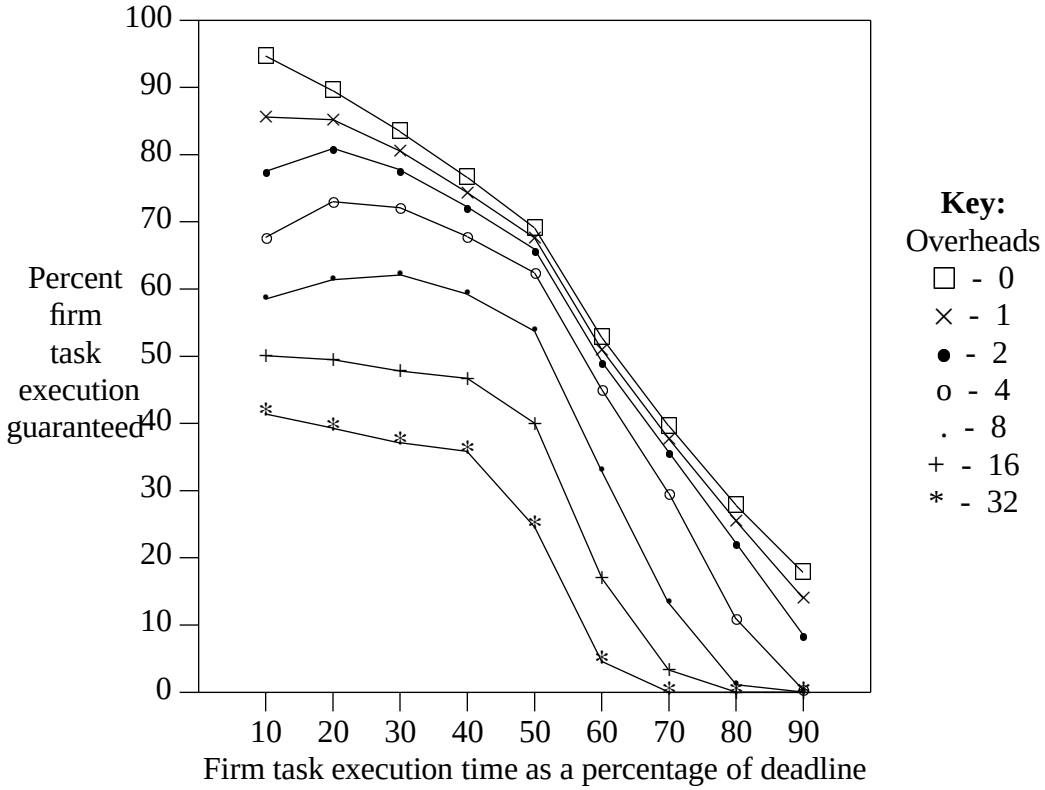
Analysis of the Extended Priority Exchange algorithm [32] can easily be extended using response time analysis [1, 34] thus catering for hard task characteristics such as sporadic arrival, release jitter, blocking and arbitrary deadlines. Hence, in common with the dual priority approach, the Extended Priority Exchange algorithm places few restriction on hard tasks. Further, in common with slack stealing, EPE requires in the worst case $O(n)$ time at each context switch to maintain extra capacity counters. The performance of

the EPE algorithm was however, generally inferior to that of the dual priority approach, both in terms of responsively scheduling soft tasks and in providing online guarantees, even with the advantage of an exact rather than approximate acceptance test.

6.1. Overheads: Simulation

To determine the effect of overheads on the percentage of firm task execution guaranteed, we carried out further simulations using only the slack stealing approach but including various levels of overhead for the acceptance test. Note, these overheads were only incurred in accepting or rejecting firm tasks which, at the time of acceptance testing, could have any chance of being accepted. Thus firm tasks with less than their execution time to go before their deadline were assumed to be rejected immediately without incurring any overhead.

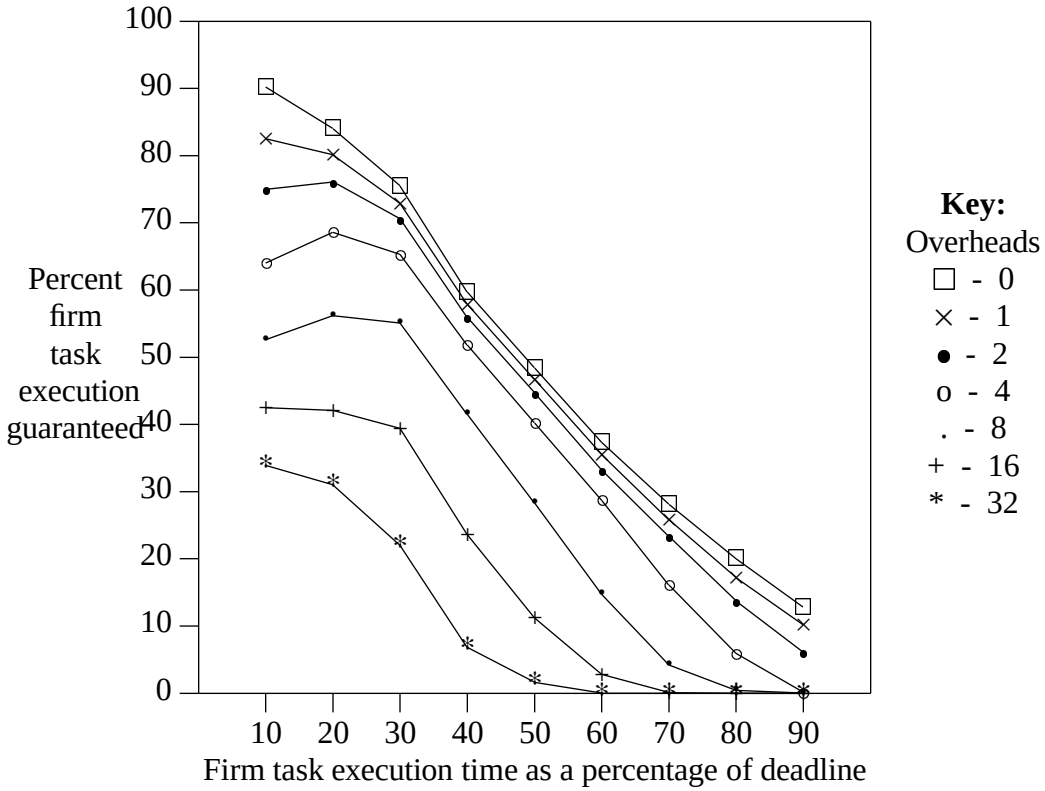
Tests 17, 18 and 19 illustrate the effects of overheads which were varied from 0 to 32 ticks per acceptance test. In these simulations, the same exponential distributions of hard and firm tasks were used as for tests 14, 15 and 16. In each case, the mean firm task deadline was 60 ticks.



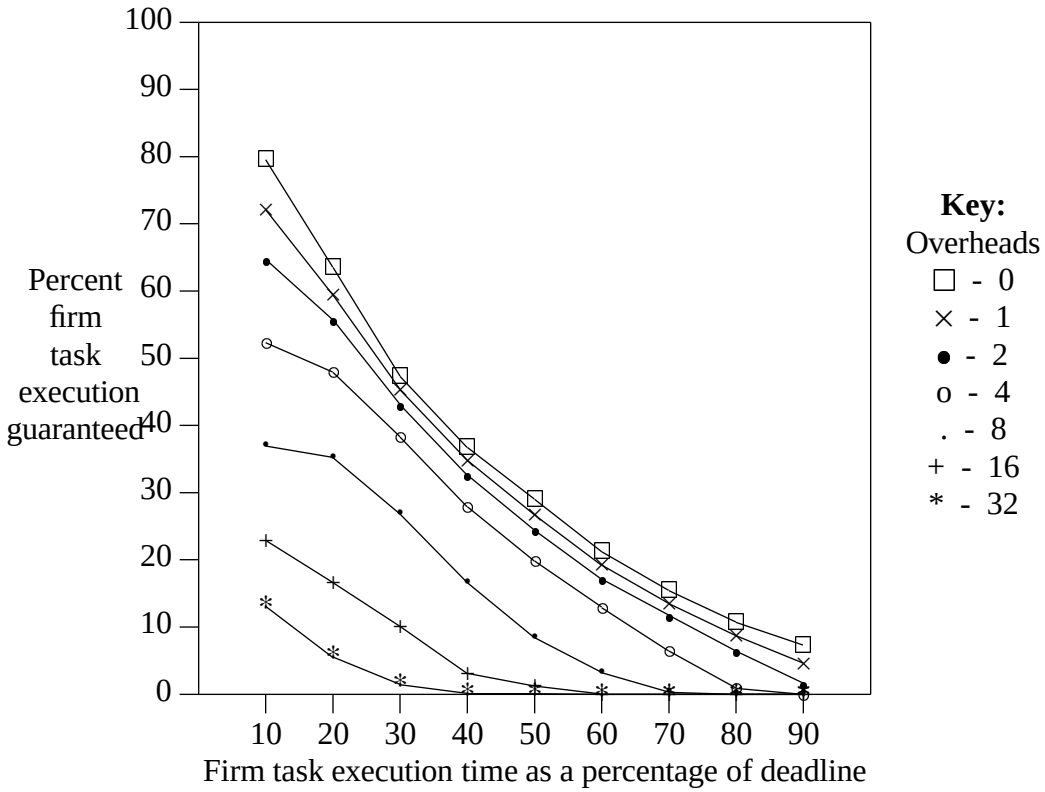
Test 17: Exponential hard task sets, 12 tasks, 50% utilisation
Exponential distribution of firm task deadlines.

From the graphs, it is clear that increasing overheads rapidly results in degraded performance. Typically, an acceptance test requiring execution time equivalent to 5 - 10% of the mean firm task deadline degrades the theoretical performance of the slack stealing approach to that of dual priority. For many real-time task sets, the additional overheads of performing a pseudo-polynomial acceptance test will exceed this level.

It is therefore our contention that the dual priority approach combined with the approximate acceptance test given in section 4 is an appropriate and practical method of responsively scheduling soft tasks and providing online guarantees for firm tasks.



Test 18: Exponential hard task sets, 12 tasks, 70% utilisation
Exponential distribution of firm task deadlines.



Test 19: Exponential hard task sets, 12 tasks, 90% utilisation
Exponential distribution of firm task deadlines.

7. Conclusions

In this paper, we have presented a new approach to providing flexibility in hard real-time systems. This approach is based upon the idea that critical hard tasks have two priorities: an initial lower band priority which is assumed upon release and an upper band priority which the task is promoted to a fixed time interval after its release. We have shown that this dual priority paradigm facilitates:

(1) The *a priori* guarantee of mandatory hard tasks.

(2) The efficient and effective provision of online guarantees for alternate / optional tasks with hard or firm deadlines.

(3) The responsive scheduling of soft tasks.

We outlined how analysis of fixed priority scheduling could be adapted to guarantee hard tasks with dual priorities. A sufficient online acceptance test was then derived enabling run-time guarantees to be given to alternate / optional tasks. The time complexity of this test is $O(n + m)$, where n is the number of hard tasks and m the number of runnable alternate / optional tasks already guaranteed.

We reported the results of a number of simulations which investigated the comparative theoretical performance of dual priority, Slack Stealing, Extended Priority Exchange and background scheduling methods. Two metrics were examined: the mean response time of soft tasks and the percentage of optional firm tasks guaranteed. In each case, the slack stealing algorithm (combined with an exact acceptance test) gave the best results. Further, the dual priority approach combined with a sufficient acceptance test exceeded the performance of the Extended Priority Exchange algorithm supported by an exact test.

Finally, we looked at practical considerations such as overheads and restrictions placed on the hard task set. We concluded that although theoretically outperformed by the slack stealing approach, dual priority scheduling has two key advantages: Its applicability to a wide range of scheduling problems and the low overheads involved in both scheduling and acceptance tests.

The dual priority approach, supported by the analysis, acceptance tests and mechanisms given in this paper, provides a practical basis for combining the benefits of guaranteeing hard requirements with the flexibility inherent in the best-effort paradigm. It therefore represents a significant step towards meeting one of the key challenges presented by the next generation of real-time systems: To provide support for dynamic, adaptive and intelligent behaviour whilst retaining the 100% guarantees needed by crucial tasks.

With this objective in mind, we intend to continue research into the use of adaptive scheduling policies within the framework of the dual priority paradigm. Further, we intend to extend the approach to disk access and communications scheduling, with the goal of providing support for multimedia systems. Finally, we also expect to implement the mechanisms required for dual priority scheduling in the Distributed Real-time Execution Environment (DrTEE) test-bed currently being developed at the University of York.

Acknowledgements

This work is supported, in part, by the UK Science and Engineering Research Council, Grant GR/H39611. The author would also like to thank Neil Audsley, Alan Burns, Andy Wellings and Ken Tindell for comments on an earlier draft of this paper.

References

1. N. Audsley, A. Burns, M. Richardson, K. Tindell and A. Wellings, ‘‘Applying New Scheduling Theory to Static Priority Pre-emptive Scheduling’’, *Software Engineering Journal* **8**(5), pp. 284-292 (September 1993).
2. N. C. Audsley, ‘‘Deadline Monotonic Scheduling’’, YCS146, Department of Computer Science, University of York (October 1990).
3. N. C. Audsley, ‘‘Flexible Scheduling in Hard Real-Time Systems’’, D.Phil. Thesis, Department of Computer Science, University of York, UK (August 1993).
4. N. C. Audsley, A. Burns, M. F. Richardson and A. J. Wellings, ‘‘Hard Real-Time Scheduling: The Deadline Monotonic Approach’’, *Proceedings 8th IEEE Workshop on Real-Time Operating Systems and Software*, Atlanta, GA, USA (15-17 May 1991).
5. N.C. Audsley, A. Burns, M.F. Richardson and A.J. Wellings, ‘‘Deadline Monotonic Scheduling Theory’’, pp. 55-60 in *Proceedings IFAC/IFIP International Workshop on Real-time Programming, WRTP’92, Bruges* (June 1992).
6. N. C. Audsley, A. Burns and A. J. Wellings, ‘‘Incorporating Unbounded Algorithms Into Predictable Real-Time Systems’’, *Proceedings of IEEE Workshop on Imprecise and Approximate Computation*, pp. 11-15 (December 1992).
7. N. C. Audsley, K. W. Tindell and A. Burns, ‘‘The End of the Road for Static Cyclic Scheduling’’, *Proceedings of 5th Euromicro Workshop on Real-Time Systems*, Oulu, Finland, pp. 36-41 (1993).
8. A. Burns and A.J. Wellings, ‘‘Criticality and Utility in the Next Generation’’, *Real-Time Systems* (1991).
9. A. Burns and A.J. Wellings, ‘‘Dual Priority Assignment: A Practical Method for Increasing Processor Utilization’’, pp. 48-55 in *Proceedings of 5th Euromicro Workshop on Real-Time Systems, Oulu*, IEEE Computer Soc. Press (1993).
10. A. Burns, A.J. Wellings and A.D. Hutcheon, ‘‘The Impact of an Ada Run-time System’s Performance Characteristics on Scheduling Models’’, pp. 240-248 in *Ada sans frontieres Proceedings of the 12th Ada-Europe Conference, Lecture Notes in Computer Science 688*, Springer-Verlag (1993).
11. H. Chetto and M. Chetto, ‘‘Some Results of the Earliest Deadline Scheduling Algorithm’’, *IEEE Transactions Software Engineering* **15**(10), pp. 1261-1269 (October 1989).
12. R. I. Davis, ‘‘Approximate Slack Stealing Algorithms for Fixed Priority Pre-emptive Systems’’, YCS217, Dept. Computer Science, University of York (1993).
13. R. I. Davis, K. W. Tindell and A. Burns, ‘‘Scheduling Slack Time in Fixed Prioroity Pre-emptive Systems’’, *Proceedings IEEE Real-Time Systems Syposium*, pp. 222-231 (December 1993).
14. K. S. Decker, V. R. Lesser and R. C. Whitehair, ‘‘Extending a Blackboard Architecture for Approximate Processing’’, *Real-Time Systems* **2**(1/2), pp. 47-80 (May 1990).
15. A. Garvey and V. Lesser, ‘‘Scheduling Satisficing Tasks with a Focus on Design-to-time Scheduling. ’’, *Proceedings of IEEE Workshop on Imprecise and Approximate Computation.*, pp. 25-29 (December 1992).
16. M. G. Harbour, M. H. Klein and J. P. Lehoczky, ‘‘Fixed Priority Scheduling of Periodic Tasks with Varying Execution Priority’’, *Proceedings 12th IEEE Real-*

Time Systems Symposium, San Antonio, TX, USA, pp. 116-128 (3-6 December 1991).

17. M. Joseph and P. Pandya, ''Finding Response Times in a Real-Time System'', *The Computer Journal (British Computer Society)* **29**(5), pp. 390-395, Cambridge University Press (October 1986).
18. B. W. Lampson and D. D. Redell, ''Experience with Processes and Monitors in Mesa'', *CACM* **23**(2), pp. 105-117 (February 1980).
19. J. P. Lehoczky, ''Fixed Priority Scheduling of Periodic Task Sets With Arbitrary Deadlines'', *Proceedings 11th IEEE Real-Time Systems Symposium*, Lake Buena Vista, FL, USA, pp. 201-209 (5-7 December 1990).
20. J. P. Lehoczky and S. Ramos-Thuel, ''An Optimal Algorithm for Scheduling Soft-Aperiodic Tasks Fixed-Priority Preemptive systems '', *Proceedings Real-Time Systems Symposium*, pp. 110-123 (December 1992).
21. J. P. Lehoczky, L. Sha and J. K. Strosnider, ''Enhanced Aperiodic Responsiveness in Hard Real-Time Environments'', *Proceedings IEEE Real-Time System Symposium*, San Jose, California, pp. 261-270 (1987).
22. K. J. Lin, S. Natarajan and J. W. S. Liu, ''Imprecise Results: Utilizing Partial Computations in Real-Time Systems'', *Proceedings 8th IEEE Real-Time Systems Symposium*, Fairmont Hotel, San Jose, California, pp. 210-217 (1-3 December 1987).
23. C. L. Liu and J. W. Layland, ''Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment'', *Journal of the ACM* **20**(1), pp. 40-61 (1973).
24. C. D. Locke, ''Best-Effort Decision Making for Real-Time Scheduling'', CMU-CS-86-134 (PhD Thesis), Computer Science Department, CMU (May 10, 1986).
25. R. Rajkumar, L. Sha and J. P. Lehoczky, ''An Experimental Investigation of Synchronisation Protocols'', *Proceedings 6th IEEE Workshop on Real-Time Operating Systems and Software*, pp. 11-17 (May 1989).
26. S. Ramos-Thuel and J. P. Lehoczky, ''On-Line Scheduling of Hard Deadline Aperiodic Tasks in Fixed Priority Systems'', *Proceedings Real-Time Systems Symposium* (December 1993).
27. K. Schwan and H. Zhou, ''Dynamic Scheduling of Hard Real-Time Tasks and Real-Time Threads'', *IEEE Transactions on Software Engineering* **18**(8), pp. 736-748 (August 1992).
28. O. Serlin, ''Scheduling of Time Critical Processes'', *Proceedings AFIPS Spring Computing Conference*, pp. 925-932 (1972).
29. L. Sha, J.P. Lehoczky and R. Rajkumar, ''Solutions For Some Practical Problems in Prioritizing Preemptive Scheduling'', *Proceedings IEEE Real-Time Sytems Symposium* (1986).
30. L. Sha, R. Rajkumar and J. P. Lehoczky, ''Priority Inheritance Protocols: An Approach to Real-Time Synchronisation'', *IEEE Transactions on Computers* **39**(9), pp. 1175-1185 (September 1990).
31. L. Sha, B. Sprunt and J. P. Lehoczky, ''Aperiodic Task Scheduling for Hard Real-Time Systems'', *The Journal of Real-Time Systems* **1**, pp. 27-69 (1989).
32. B. Sprunt, J. Lehoczky and L. Sha, ''Exploiting Unused Periodic Time For Aperiodic Service Using the Extended Priority Exchange Algorithm '', *Proceedings IEEE Real-Time Systems Symposium*, pp. 251-258 (December 1988).

33. J. A. Stankovic and K. Ramamritham, ‘‘What is Predictability for Real-Time Systems?’’, *The Journal of Real Time Systems* **2**(4), pp. 247-254 (1990).

34. K. Tindell, A. Burns and A.J. Wellings, ‘‘An Extendible Approach for Analysing Fixed Priority Hard Real-Time Tasks’’, *Journal of Real Time Systems* **6**(2), Department of Computer Science, University of York (March 1994).

35. K.W. Tindell, A. Burns and A.J. Wellings, ‘‘Mode Changes in Priority Pre-emptively Scheduled Systems’’, *Proceedings IEEE Real-Time Systems Symposium*, pp. 100-109 (December 1992).