Scheduling Slack Time in Fixed Priority Pre-emptive Systems

R.I.Davis, K.W.Tindell, A.Burns

Department of Computer Science, University of York, England.

Abstract

This paper addresses the problem of jointly scheduling tasks with both hard and soft time constraints. We present a new analysis which builds upon previous research into slack stealing algorithms. Our analysis determines the maximum processing time which may be stolen from hard deadline periodic or sporadic tasks, without jeopardising their timing constraints. It extends to tasks with characteristics such as synchronisation, release jitter and stochastic execution times, as well as forming the basis for a family of optimal and approximate slack stealing algorithms.

1. Introduction

In a recent paper [7], Lehoczky and Ramos-Thuel presented a new approach to servicing aperiodic requests within the context of a hard real-time system. Their method, known as the Slack Stealer is applicable to systems scheduled using a fixed priority pre-emptive dispatcher, with priorities assigned according to a policy such as the Rate Monotonic algorithm [11]. The Slack Stealer addresses the problem of minimising the response times of soft aperiodic tasks whilst guaranteeing that the deadlines of hard periodics are met. In this paper, we present new analysis which forms the basis of other more generally applicable slack stealing algorithms.

The problem of jointly scheduling both hard and soft deadline tasks is an important issue in many real-time systems. This is due to tension between the scheduling requirements of tasks in the two categories. Typically, soft tasks benefit from being delivered as early as possible, whilst hard tasks need to be guaranteed to meet their deadlines. Further, it is expected that future real-time systems may include optional soft tasks, which can be used to increase the precision, utility or confidence level of hard real-time services [3]. These soft tasks do not constitute background services but are inextricably linked to the behaviour of the set of hard tasks. Clearly if soft tasks of this type are to provide benefit to the system, then they need to be allowed some execution time prior to the deadline of the hard service which they support.

Analysis of fixed priority pre-emptive scheduling has provided a sound theoretical basis for designing predictable hard real-time systems [1, 9]. Within this framework, a number of approaches have been developed for scheduling mixed task sets. The simplest and perhaps least effective of these is to execute soft deadline tasks at a lower priority level than any of those with hard deadlines. This effectively relegates the soft tasks to background processing. Alternatively, soft tasks may be run at a higher priority under the control of a pseudo hard real-time server task, such as a simple polling server.

A polling server is a periodic task with a fixed priority level (usually the highest) and an execution capacity. The capacity of the server is calculated off-line and is normally set to the maximum possible, such that the hard task set, including server, is schedulable. At run-time, the polling server is released periodically and its capacity is used to service soft real-time tasks. Once this capacity has been exhausted, execution is suspended until it can be replenished at the server's next release.

The polling server will usually significantly improve the response times of soft tasks over background processing. However, if the ready soft tasks exceed the capacity of the server, then some of them will have to wait until its next release, leading to potentially long response times. Conversely, no soft tasks may be ready when the server is released, wasting its high priority capacity.

This latter drawback is avoided by the Priority Exchange, Deferrable server [8] and Sporadic server [13] algorithms. These are all based on similar principles to the polling server. However, they are able to preserve capacity if no soft tasks are pending when they are released. Due to this property, they are termed "bandwidth preserving algorithms". The three algorithms differ in the ways in which the capacity of the server is preserved and replenished and in the schedulability analysis needed to determine their maximum capacity.

In general, all three offer improved responsiveness over the polling approach. However, there are still disadvantages with these more complex server algorithms.

They are unable to make use of slack time which may be present due to the often favourable phasing of periodic tasks (i.e. not worst case). Further, they tend to degrade to providing essentially the same performance as the polling server at high loads. The Deferrable and Sporadic servers are also unable to reclaim spare capacity gained, when for example, hard tasks require less than their worst case execution time. This spare capacity, termed *gain time*, can however be reclaimed by the Extended Priority Exchange algorithm [14].

The Slack Stealing algorithm of Lehoczky and Ramos-Thuel [7] suffers from none of these disadvantages. It is optimal in the sense that it minimises the response times of soft aperiodic tasks amongst all algorithms which meet all hard periodic task deadlines. The Slack Stealer services aperiodic requests by making any spare processing time available as soon as possible. In doing so, it effectively steals slack from the hard deadline periodic tasks. A means of determining the maximum amount of slack which may be stolen, without jeopardising the hard timing constraints, is thus key to the operation of the algorithm.

In [7] Lehoczky and Ramos-Thuel describe how the slack available can be found. This is done by mapping out the processor schedule for the hard periodic tasks over their *hyperperiod* (the least common multiple of task periods). The mapping is then inspected to determine the slack present between the deadline on one invocation of a task and the next. The values found are stored in a table.

At run-time, a set of counters are used to keep track of the slack which may be stolen at each priority level. These counters are decremented depending on which tasks, if any, are executing and updated by reference to the table, at the completion of each task. Whenever, the counters indicate that there is slack available at *all* priority levels, then soft tasks may be executed at the highest priority level.

Unfortunately, the need to map out the hyperperiod restricts the applicability of the Slack Stealer: Slack can only be stolen from hard deadline tasks which are strictly periodic and have no release jitter [1] or synchronisation. Realistically, it is also limited to task sets with a manageably short hyperperiod. This is a significant restriction, as even modest task sets (e.g. 10 tasks) may have very long hyperperiods.

In this paper, we extend the deadline monotonic analysis given by Audsley *et al* [2] to determine the slack which may be stolen from both hard deadline sporadic and hard deadline periodic tasks. This new analysis forms the basis for a dynamic slack stealing algorithm, which we prove to be optimal. The dynamic slack stealer is equivalent to the static Slack Stealer of Lehoczky and Ramos-Thuel, in the limited case of independent periodic tasks. However, by virtue of computing the slack at run-time, the dynamic algorithm is applicable to a more general class of scheduling problems including hard deadline sporadics and tasks which exhibit release jitter and synchronisation. Further, the dynamic algorithm is able to improve the response times of soft tasks by exploiting run-time information about hard task execution requirements, extended inter-arrival times and deadlines. It can also be readily extended to cover a broad spectrum of other task characteristics. Unfortunately, the execution time overhead of the optimal dynamic algorithm is such that it is infeasible in practice. We address this problem by presenting a family of approximate algorithms which provide close to optimal performance with practical utility.

In section 2, we outline the computational model and assumptions used. Section 3 presents analysis of the maximum amount of slack which may be stolen at each priority level. This is used as the basis for the dynamic algorithm. In section 4, we show how our analysis can be extended to tasks which exhibit blocking and release jitter. Further, we examine how gain time can be reclaimed from tasks which require less than their worst case execution time or extend their current period or deadline. We also briefly discuss resource sharing between hard and soft tasks and give an outline of how tasks with arbitrary deadlines may be incorporated into our model. In section 5, we consider the implementation of approximate slack stealing algorithms and compare their performance to that of the optimal algorithm. Finally, we briefly discuss the overheads involved in calculating slack dynamically. Section 6 offers our conclusions.

2. Computational model and assumptions

In this paper, we consider the scheduling of n hard deadline tasks on a single processor. The analysis given, is however, equally applicable to multiprocessor systems with a static allocation of tasks. Each task has a base priority i where $1 \le i \le n$, thus 1 is the highest priority level and n the lowest. We use $hp(i)$ to denote the set of tasks with a higher base priority than i and $lp(i)$ to denote the tasks with base priority i or lower. Each task gives rise to an infinite sequence of invocation requests, separated by a minimal inter-arrival time T_i. 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. Further, each task may lock and unlock semaphores according to the Priority Ceiling Protocol [12]. The worst case blocking time which an invocation of task i can experience due to the operation of this protocol is denoted by B_i. Although the tasks are assigned unique static priorities, they may have their priorities temporarily increased due to priority inheritance, as part of the operation of the Priority Ceiling Protocol. Finally, tasks can arrive at any time after their minimum inter-arrival interval, but be delayed for a variable but bounded amount of time (termed the *release jitter*, J_i) before being placed on a notional run-queue by the dispatcher, they are then said to be released.

The analysis presented in section 3 uses the concept of *busy* and *idle periods* [6]. These are defined as follows: A level i busy period is a continuous time interval during which the notional run-queue contains one or more tasks of priority level i or higher. Similarly, a level i idle period is a time interval during which the run-queue is free of level i or higher priority tasks. (We note that the run queue may become momentarily free of level i tasks, when one tasks completes and another is released. This appears in our formulation as an idle period of zero length.)

In subsequent sections, the following assumptions apply:

- The hard deadline task set is assumed to be schedulable using fixed priority pre-emptive dispatching with a priority ordering determined by some means such as deadline monotonic priority assignment [10].
- All overheads for context switching, scheduling etc. are subsumed into the worst case execution times and blocking factors.
- Tasks cannot voluntarily suspend themselves.

3. Schedulability Analysis

In this section, we determine the maximum amount of processing time which may be stolen from an invocation of a hard deadline task without causing its deadline to be missed.

For clarity, we initially assume that the task set exhibits no synchronisation or release jitter and that each invocation takes its worst case execution time. Further, we assume that the deadline of each task is less than or equal to its minimum inter-arrival time. In section 4, we relax these assumptions.

Our formulation stems from considering the schedulability of each hard real-time task at some arbitrary time t. We assume that at time t, the following data is available via the operating system, (typically derived from data stored in a task control block):

$l_{i,t}$	The time at which task i was last released.
$x_{i,t}$	The earliest possible next release of task i. Typically $x_{i,t} = l_{i,t} + T_i$.
$d_{i,t}$	The next deadline on an invocation of task i. (Note, if the current invocation of task i is complete, then $d_{i,t} = x_{i,t} + D_i$, i.e. $d_{i,t}$ is the deadline following the next release).
$c_{i,t}$	The remaining execution time budget for the current invocation of task i. Typically found by subtracting the execution time used from the worst case execution time, C_i. (Note, if at time t task i is complete and thus awaiting release, then $c_{i,t} = 0$).

Note, $l_{i,t}$, $x_{i,t}$ and $d_{i,t}$ are all measured relative to time t.

Initially, we focus on finding the maximum amount of slack time, $S_{i,t}^{\max}$, which may be stolen at priority level i, during the interval $[t\ ,\ t + d_{i,t})$, whilst guaranteeing that task i meets its deadline. (Note, $S_{i,t}^{\max}$ may not actually be available for soft task processing due to the constraints on hard deadline tasks with priorities lower than i. We return to this point in section 3.1.) To guarantee that task i will meet its deadline, we need to analyse the worst case scenario from time t onwards. We therefore assume that all tasks j are re-invoked at their earliest possible next release $x_{j,t}$ and subsequently with a period of T_j.

In attempting to determine the maximum guaranteed slack, $S_{i,t}^{\max}$, it is instructive to view the interval $[t\ ,\ t + d_{i,t})$ as comprising a number of level i busy and idle periods. Any level i idle time between the completion of task i and its deadline could be swapped for task i computation without causing the deadline to be missed. Hence the maximum slack which may be stolen is equal to the total level i idle time in the interval. We use this result to calculate $S_{i,t}^{\max}$.

Our method for finding the level i idle time relies on two equations: equation (1), determines $w_{i,t}$, the length of a level i busy period which starts at time t. Equation (2) determines the length of a level i idle period given its start time. By combining these two equations, we are able to iterate over the interval $[t\ ,\ t + d_{i,t})$, totaling up all the idle time, and hence find $S_{i,t}^{\max}$.

We first derive equation (1) using techniques given in [1]; two components determine the extent of the busy period:

1. The level i or higher priority processing outstanding at time t.
2. The level i or higher priority processing released during the busy period.

The second component implies a recursive definition. As the processing released increases monotonically with the length of the busy period, a recurrence relation can be used to find $w_{i,t}$: †

† Note, $(x)_0$ is notational shorthand for $\max(x\ ,0)$, i.e. the minimum value of $(x)_0$ is zero. $\lceil y \rceil$ is notation for the ceiling function.

$$w_{i,t}^{m+1} = S_{i,t} + \sum_{\forall j \in hp(i) \cup i} \left(c_{j,t} + \left\lceil \frac{(w_{i,t}^m - x_{j,t})_0}{T_j} \right\rceil C_j \right) \qquad (1)$$

The term $S_{i,t}$ represents level *i* slack processing released at time *t*. We return to this shortly.

The recurrence relation begins with $w_{i,t}^0 = 0$ and ends when $w_{i,t}^{m+1} = w_i^n$ or $w_{i,t}^{m+1} > d_{i,t}$. Proof of convergence follows from analysis of similar recurrence relations by Joseph and Pandya [5] and Audsley *et al* [1]. The final value of $w_{i,t}$ defines the length of the busy period. Alternatively, we may view $t + w_{i,t}$ as defining the start of a level *i* idle period.

Given the start of a level *i* idle period, within the interval $[t\ ,\ t + d_{i,t})$, the end of the idle time, which may be converted to slack, occurs either at the next release of a task of priority *i* or higher or at the end of the interval. Equation (2) gives the length, $v_i(w_{i,t})$, of the level *i* idle window.

$$v_{i,t}(w_{i,t}) = \min \left(\begin{array}{l} (d_{i,t} - w_{i,t})_0\ , \\ \min\limits_{\forall j \in hp(i) \cup i} \left(\left\lceil \dfrac{w_{i,t} - x_{j,t}}{T_j} \right\rceil_0 T_j + x_{j,t} - w_{i,t} \right) \end{array} \right) \qquad (2)$$

Combining equations (1) and (2), our method for determining the maximum slack, $S_{i,t}^{\max}$, proceeds as follows:

1. The slack which may be stolen, $S_{i,t}$, is initially set to zero.
2. Equation (1) is used to compute the end of a busy period in the interval $[t\ ,\ t + d_{i,t})$.
3. The end of the busy period is used as the start of an idle period by equation (2) which returns the length of contiguous idle time.
4. The slack processing, $S_{i,t}$ is incremented by the amount of idle time found in step 3.
5. If the deadline on task *i* has been reached, then the maximum slack which can be stolen is given by $S_{i,t}$. Otherwise, we repeat steps 2 to 5.

This method can be implemented as detailed in algorithm (3).

The complexity of the above approach is O(*mn*) where *m* is the number of iterations and *n* is the number of hard deadline tasks. Hence, computing the exact slack available at all *n* priority levels can be done in O(mn^2) time. We note that the number of iterations, *m*, depends on the periods and deadlines of the hard tasks, thus the complexity of algorithm (3) is pseudo-polynomial.

Algorithm for determining the level *i* slack

```
$S_{i,t} = 0$
$w_{i,t}^{m+1} = 0$
do   while $w_{i,t}^{m+1} \le d_{i,t}$
    $w_{i,t}^m = w_{i,t}^{m+1}$
    $w_{i,t}^{m+1} = S_{i,t} + \sum_{\forall j \in hp(i) \cup i} \left( c_{j,t} + \left\lceil \frac{(w_{i,t}^m - x_{j,t})_0}{T_j} \right\rceil C_j \right)$
    if   $w_{i,t}^m = w_{i,t}^{m+1}$
    then $S_{i,t} = S_{i,t} + v_{i,t}(w_{i,t}^m) + \varepsilon$
         $w_{i,t}^{m+1} = w_{i,t}^{m+1} + v_{i,t}(w_{i,t}^m) + \varepsilon$
    endif
$S_{i,t}^{\max} = S_{i,t} - \varepsilon$                                    (3)
```

(Note: ε, set to the granularity of time, is a mathematical device used to force the recurrence relation to continue.)

If the schedulability of the hard task set has been guaranteed off-line, then $S_{i,t}^{\max}$ will always be greater than or equal to zero. However, if the hard task set is not guaranteed, then algorithm (3) provides early error detection as $S_{i,t}^{\max} < 0$ indicates that task *i* could miss its next deadline.

It is important to note that the above formulation is applicable regardless of whether each hard deadline task is periodic or sporadic. Further, it can be readily extended to handle more complex task characteristics. (See section 4.)

3.1. Dynamic slack stealing algorithm

In this section, we use our previous analysis as the basis for a dynamic slack stealing algorithm. We require that soft tasks are executed as soon as possible, such that the deadlines on all hard real-time tasks are still guaranteed to be met. In the case of strictly periodic tasks discussed by Lehoczky and Ramos-Thuel [7], this can be achieved by servicing soft tasks at the highest priority, when there is slack available at *all* priority levels. However, when hard sporadic tasks are considered, there are problems with this approach.

Suppose, at time *t* there are soft tasks pending and the highest priority hard task in the run-queue is task *k*. Further, suppose a hard sporadic task with priority higher than *k* has zero slack (say D = C for this task) and could arrive at any time. This sporadic task may never arrive, preventing slack from ever being available at *all* priority levels.

To avoid the above problem, we use a different criteria for determining when soft tasks may execute. Our analysis guarantees that, provided level *k* soft task processing is limited to $S_{k,t}^{\max}$ in the interval $[t\ ,\ t + d_{k,t})$ then hard tasks at priority levels k and higher will meet their deadlines. As we require that the deadlines of *all* hard real-time tasks are met, soft task processing is only permissible at priority k whilst there is slack present at priority level k and *all* lower levels:

$$\min_{\forall j \in lp(k)} S_{j,t}^{\max} > 0 \tag{4}$$

(Note, for completeness, when there are no hard tasks runnable, we regard there as being infinite slack available at priority level n+1). Provided inequality (4) is true, soft tasks can execute at priority k, (in preference to hard task k) even though higher priority sporadics apparently have zero slack.

Using the above result, the dynamic slack stealing algorithm is formulated as follows: Whenever there are soft tasks pending, algorithm (3) is used to find the slack available at each priority level lower than or equal to k. Where k is the priority level of the highest priority runnable hard task. Inequality (4) is then used to determine if soft task processing can proceed immediately in preference to task k.

We note that the dynamic algorithm, described above, potentially requires the slack at each priority level to be re-computed at each time increment. We explore this problem further in section 3.4.

3.2. Optimality of the dynamic algorithm

In this section, we prove that the dynamic slack stealing algorithm described above is optimal.

Theorem: *For any hard deadline task set scheduled according to a fixed priority scheme and a stream of soft deadline tasks processed in FIFO order, the dynamic slack stealing algorithm, minimises the response time of every soft task, amongst all algorithms which are guaranteed to meet all hard task deadlines. (Note, we exclude clairvoyant algorithms which may use prior knowledge of sporadic arrivals.)*

Proof: We prove the theorem by showing that any alternative algorithm, A, which results in a shorter response time for any soft task cannot guarantee that the deadlines of all the hard tasks will be met.

Let f be the first soft task which has a shorter response time when scheduled by algorithm A. As f is the first such task, the response times of all previously serviced soft tasks must be the same as, or longer than when scheduled by the dynamic slack stealer. Thus f reaches the head of the queue no earlier when scheduled by algorithm A. Once at the head of the queue, f is serviced by the dynamic slack stealer so long as inequality (4) remains true. For algorithm A to result in a lower response time, it must process f for at least one clock tick when the dynamic slack stealer is unable to do so. We denote the time at which this occurs by t. Further, we assume that the highest priority runnable task at time t is task k. The dynamic slack stealer uses the exact upper bound on the slack which may be stolen, this is zero at time t. Hence at least one of the hard deadline tasks with priority equal to or lower than k has zero slack at time t, let this be task i. In servicing soft task f at time t, algorithm A has therefore lengthened the worst case level i busy period culminating in the completion of task i beyond its deadline. Algorithm A cannot therefore guarantee that the deadline of task i will be met. A simple induction argument proves that the dynamic slack stealer minimises the response times of all the soft tasks.

□

We note that the dynamic slack stealer is optimal for stealing slack from both periodic and sporadic tasks with hard deadlines. In the case of task sets comprising only periodic tasks, its behaviour is equivalent to the static, table driven Slack Stealing algorithm of Lehoczky and Ramos-Thuel. We return to this point in section 5.

3.3. Feasibility of the dynamic algorithm

We now seek to reduce the run-time overheads of the dynamic algorithm. To do this, we examine the feasibility of deriving the slack available at some later time t' from the values computed at time t. First, we assume that no hard tasks complete during the interval $[t\ ,\ t')$ and that each task i is released at $t + x_{i,t}$ and subsequently with a period of T_i. By considering the level i idle time in the intervals $[t\ ,\ t + d_{i,t})$ and $[t'\ ,\ t + d_{i,t})$, it can be seen that, if the processor serviced soft tasks or was idle between t and t' then slack is consumed at all priority levels:

$$\forall j \in hp(i) \cup lp(i): \quad S_{j,t'}^{\max} = S_{j,t}^{\max} - (t' - t) \tag{5}$$

Whereas, if the processor was busy with hard deadline task i, then slack is consumed at all priority levels higher than i:

$$\forall j \in hp(i): \quad S_{j,t'}^{\max} = S_{j,t}^{\max} - (t' - t) \tag{6}$$

Next, we consider the effect of task i completing on the level i slack. The next deadline which task i could miss increases by at least T_i when it completes. As any level i idle time in the interval $[t\ ,\ t + d_{i,t})$ must also be present in $[t\ ,\ t + d_{i,t} + T_i)$, the level i slack cannot be reduced when task i completes. On the contrary, it may well be increased. Hence, for stealing slack from strictly periodic hard deadline tasks we need only re-compute the level i slack each time task i completes. (Note, the level i slack present immediately prior to completion provides a lower bound which can be used as an initial value for $S_{i,t}$ in algorithm (3), reducing the computation required.)

Finally, we consider the effect of sporadic tasks not arriving at their maximum rate. Suppose a sporadic task i is not released at its earliest possible next release time, t^1, but at some later time t^2. Equations (5) and (6) provide a lower bound on the level i slack at time t^2. However, the deadline on the request at t^2 is later than it would have been had the request occurred at time t^1, potentially increasing the slack. Therefore to ensure that we have the exact upper bound on the slack available, we are compelled to re-evaluate $S_i^{\max}$ at each time increment in the interval $[t^1, t^2]$. Again, this may be done using the lower bound given by equations (5) and (6) as an initial value. In addition to the possible increase in level i slack, the interference on lower priority tasks may be reduced. (This can be seen by examining the effect of increasing the values of $x_{j,t}$ in algorithm (3)). To retain optimality, we are therefore compelled to also re-evaluate the slack available at all priority levels lower than i.

With a large number of sporadic tasks, the exact slack available at each priority level can potentially change on each processor clock tick. Thus the slack at each priority level may still need to be re-evaluated at each clock tick. Clearly, this is infeasible in practice. It does however provide us with an optimal algorithm for stealing slack from both hard deadline periodic and sporadic tasks. Moreover, it forms the basis for approximate, but more efficient, slack stealing algorithms which we discuss in section 5.

4. Extensions

The analysis given in section 3 extends the scope of slack stealing algorithms to task sets comprising hard deadline sporadic as well as periodic tasks. In this section, we augment this analysis further, to handle tasks which exhibit stochastic execution times, release jitter and blocking. We also discuss resource sharing between hard and soft tasks. Finally, we briefly outline how tasks with arbitrary deadlines may be incorporated into our model.

4.1. Reclaiming ‘Gain Time’

‘Gain time’ is generated when a guaranteed hard deadline task requires less than its worst case execution time. Gain time may be identified at various points during the execution of a task: for example, at the start by inspecting the task’s input parameters. (These can have a critical influence on the execution path, by determining the number of times around a loop etc.) Using this run-time information, a less pessimistic, ‘specific’ worst case execution time can be calculated [15]. Alternatively, gain time may be identified part way through task execution via milestones [4]. The milestones separate the task into sections each of which has a worst case execution time. This enables gain time to be identified when a section completes in less than its worst case time. Finally, gain time can be identified when the entire task completes in less than its worst case time.

The analysis given in section 3 requires no modification to reclaim gain time. The worst case execution time remaining, $c_{i,t}$, is normally found by subtracting the execution time used from the worst case execution time, C_i. However, the three methods of identifying gain time described above may be used to provide less pessimistic values for $c_{i,t}$, enabling algorithm (3) to reclaim gain time as slack. In fact, gain time may be directly added to the slack available without recourse to algorithm (3): Suppose gain time g_i is identified at level i, then the slack available at level i and all lower priority levels is increased by g_i [7].

Additional slack may also be generated when it is known that the next release of a sporadic task will occur after a time greater than its minimum inter-arrival time. An example of this is a task which carries out some operation when a measured value reaches a certain level. It is assumed that the maximum rate of change of the value is known. Once a measurement has been taken, it is often possible to determine when the task should be next released, so that it is guaranteed not to miss the value reaching its specified level. Thus the task operates with a guaranteed minimum period when the value is close to its critical level and less frequently otherwise.

In the above scenario, an extended earliest possible release, $x_{i,t}$, of an invocation may result in increased slack time. Similarly if the deadline on a hard real-time service is dependent on the external environment, it will be guaranteed at a minimum acceptable level, however, at run-time, the deadline on specific invocations may be increased. Again this is potentially a source of gain time.

It is interesting to note the effect of executing an independent hard deadline task i in slack time (i.e. effectively at a raised priority). This results in gain time being generated at level i, whilst slack time is reduced at all priority levels. The gain time produced may be transformed into slack at level i and below, illustrating the equivalence between slack and gain time.

4.2. Release jitter

When a task is subject to a bounded delay between its arrival and release, it is said to exhibit release jitter [1]. To incorporate release jitter into our analysis, we need only consider the effect on the earliest possible next release of each task. We also assume that there may only be one invocation of each task present at any given time, hence $D_i + J_i \le T_i$. For each task i with release jitter, we modify the calculation of $x_{i,t}$ as follows:

$$x_{i,t} = (l_{i,t} + T_i - J_i)_0 \tag{7}$$

4.3. Blocking

We now relax the assumption that the hard real-time tasks are independent. We assume that each invocation of a task i may lock and unlock semaphores according to the Priority Ceiling Protocol. Each invocation of task i may therefore be blocked for at most B_i, the worst case blocking time. Where B_i is equal to the longest critical section of any lower priority task which accesses a semaphore with a ceiling priority of i or higher [12]. For the moment, we assume that for Sha *et al's* analysis of the Priority Ceiling Protocol to hold, we require that no semaphores, used by hard deadline tasks, may be accessed by any task executing in slack time, (we return to this point in section 4.4).

In section 3, we derived the slack available from a consideration of busy and idle periods. Taking account of semaphore access, task i may be blocked for at most B_i, extending the busy period given by equation (1). As the busy period includes precisely one invocation of task i, we augment our analysis by including the blocking factor in equation (1). (Similar changes are needed to algorithm (3).)

$$w_{i,t}^{m+1} = B_i + S_{i,t} + \sum_{\forall j \in hp(i) \cup i} \left(c_{j,t} + \left\lceil \frac{(w_{i,t}^m - x_{j,t})_0}{T_j} \right\rceil C_j \right) \tag{8}$$

We note that this represents the pessimistic view that task i will always be subject to blocking. However, at run-time, it is possible to determine a less pessimistic bound on the time for which the current invocation of a task i could be blocked. The dynamic slack stealing algorithm can be augmented to use this information and thus reclaim unused blocking time. Unfortunately, space considerations prevent us from describing our research in this area.

4.4. Resource sharing between hard and soft tasks

We now briefly consider the situation where a soft task shares one or more semaphores with the hard deadline tasks. Suppose that the soft task wishes to lock a semaphore with ceiling priority j. The deadlines of tasks at priority levels j or lower could be jeopardised by being blocked by the soft task. To ensure that this cannot happen, the soft task must be guaranteed sufficient slack processing time to complete its critical section, prior to the continued execution of any tasks of priority j or lower. Only then can it be allowed to lock the semaphore. If the length of the critical section is bounded by c, the ceiling priority of the semaphore is j and the time of the request is t, then the soft task is only allowed to succeed in locking the semaphore if:

$$\min_{\forall i \in lp(j)} S_{i,t}^{\max} \geq c \tag{9}$$

Provided the above condition holds, only hard tasks with priority higher than j can pre-empt execution of the soft task's critical section. From the definition of the priority ceiling, none of these higher priority tasks use the semaphore and hence none of them can be blocked by the soft task. Further, the soft task is guaranteed execution time of at least c in preference to hard tasks of priority j and lower. It is therefore able to complete its critical section before being pre-empted by any task of priority j or lower. Thus using the criteria given in (9) resources may be shared between hard and soft tasks without increasing the blocking on any hard task.

4.5. Arbitrary deadlines

Tasks can be permitted arbitrary deadlines (i.e. $D_i > T_i$) [6] by combining the analysis of Tindell *et al* [16, 17] with that given in section 3. To find the worst-case response time of a task i the arbitrary deadline analysis examines a number of level i busy periods and takes the largest response time corresponding to each of these.

To apply the arbitrary deadline analysis to the algorithm for determining slack (3), we must check when evaluating a level i busy period to see if the end of the busy period exceeds the subsequent release of task i, denoted by $y_{i,t} = d_{i,t} - D_i + T_i$ (i.e. if $w_{i,t} > y_{i,t}$). If it does then we must determine if the level i slack is limited by the invocation of task i released at $y_{i,t}$. This is done by examining another level i busy period starting at time t but including additional computation time C_i. If this new busy period exceeds the subsequent invocation of task i (i.e. $w_{i,t} > y_{i,t} + T_i$), then we must check if the slack is limited by the invocation of task i released at $y_{i,t} + T_i$. To do this, we examine a further busy period, again starting at time t, but including additional computation time $2C_i$. In general, if the $(q-1)$th busy period exceeds $y_{i,t} + qT_i$ we need to check if the level i slack is limited by the invocation of task i released at $y_{i,t} + qT_i$. The slack available at level i is then the minimum of the slack on each of the q invocations examined. This value is used as before, in inequality (4) to determine whether soft task processing may take place.

The algorithm for determining slack extends the busy period to $d_{i,t} + qT_i$ which given $D_i > T_i$, always exceeds the next release of task i. The sequence of busy periods which require examination is therefore potentially infinite. The invocation of task i with the least slack must however, be one of the first h_i examined, (where h_i is the number of invocations of task i in H_i, the hyperperiod of the subset of tasks with priority i and higher). This can be seen by considering the slack on the qth and $(q + h_i)$th invocations of task i. All the level i idle time in H_i lies between these two invocations, hence the slack on the $(q + h_i)$th invocation exceeds that on the qth by the level i

idle time in the level *i* hyperperiod. Thus the invocation with the least slack must be one of the first h_i. As noted earlier, the hyperperiod of task sets is often very large, hence examining h_i invocations is infeasible in practice. The authors have therefore developed an imprecise algorithm which provides a lower bound on the available slack and converges to the exact value. The detailed operation of this algorithm is however beyond the scope of this paper.

5. Approximate slack stealing algorithms

In this section, we discuss approximations to the optimal dynamic algorithm. Here, our aim is to produce slack stealing algorithms which are efficient enough for run-time usage. First we consider stealing slack from purely periodic task sets. We indicate how our analysis can be used in an algorithm which replicates the behaviour of the optimal Slack Stealer of Lehoczky and Ramos-Thuel. We then present an approximate algorithm which can be used to steal slack from both hard deadline periodic and sporadic tasks. We compare the performance of this algorithm to the optimal and background processing methods. Finally, we discuss the overheads of the approximate algorithm and their justification in terms of improved soft task response times.

For hard deadline periodic task sets, the slack available at priority level *i* only increases when task *i* completes. We may therefore replicate the behaviour of optimal Slack Stealer of Lehoczky and Ramos-Thuel by using algorithm (3) to calculate $S_{i,t}^{\max}$ at each completion of task *i*. Equations (5) and (6) are then used to keep track of the slack available at other times. The overhead of computing the slack can be reduced by *a priori* calculation of the least additional slack, S_i^{add}, which becomes available at *every* completion of task *i*. This enables a less pessimistic initial value for $S_{i,t}$ to be used in algorithm(3), thus reducing computation.

The least additional level *i* slack is generated when task *i* completes as late as possible (i.e. at its deadline) and the subsequent invocation is subject to the maximum interference from higher priority tasks. Maximum interference occurs when all higher priority tasks are released at the above deadline. This is effectively a critical instant [11] for task *i*. Thus S_i^{add} is equivalent to the level *i* idle time in the interval $[0\ ,\ T_i)$ where time 0 is a critical instant. Algorithm (3) may be used to calculate the value of S_i^{add} off-line.

We may implement a simple slack stealing algorithm by incrementing the level *i* slack available by S_i^{add} at each completion of task *i*, whilst again using equations (5) and (6) to keep track of the slack at other times. Our approximate slack stealing algorithm comprises this simple algorithm augmented by periodically evaluating the exact slack available at *all* priority levels. Varying the period at which this is done enables the overhead of the dynamic slack stealer to be traded off against a decrease in the responsiveness of soft tasks. A very short period minimises the response times of soft tasks at the expense of a large overhead (c.f. the optimal dynamic algorithm). Whilst a very long period minimises the overhead, but increases the response times. In the next section, we examine this performance trade-off.

5.1. Performance comparison

For evaluation purposes, we used sets of hard deadline periodic tasks, enabling comparisons to be made between the approximate algorithm described above and the optimal Slack Stealer of Lehoczky and Ramos-Thuel. Our results are averaged over 10 task sets, each comprising 10 hard deadline periodic tasks. The periods of the hard deadline tasks were chosen at random in the range 2 to 1000 ticks. Deadlines were randomly chosen, but constrained to be less than or equal to the period. Finally computation times were adjusted randomly until the total processor utilisation of the hard deadline task set was approximately 50%. Each task set was then subject to a sufficient and necessary schedulability test. Unschedulable task sets were discarded.

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 distribution over the test duration (100000 ticks). The number of soft tasks was varied to produce a simulated total processor loading of 80%, 85%, 90% and 95%. For each utilisation level, we recorded the response time of each soft task scheduled by background, optimal and approximate slack stealing algorithms. In the case of the approximate algorithm, the simulation was repeated using periods of 250, 500, 1000, 2000, 4000, 8000, 16000 and 32000 ticks between calculations of the exact slack available.

Figure 1 shows how the mean response time of the soft tasks varied with the period of the approximate slack stealing algorithm for the four values of processor loading. (Note, for comparison purposes, the mean response times of soft tasks scheduled by the optimal algorithm are plotted on the left, whilst corresponding values for background processing are plotted on the right). The graph shows that for all loadings the mean response time of soft tasks, scheduled by the approximate algorithm, was very close to the optimal provided the period of the approximate algorithm was less than the mean task period (500 ticks). Increasing the period of the approximate algorithm resulted in decreased responsiveness, until with a large period (32000), it was close to that of background.

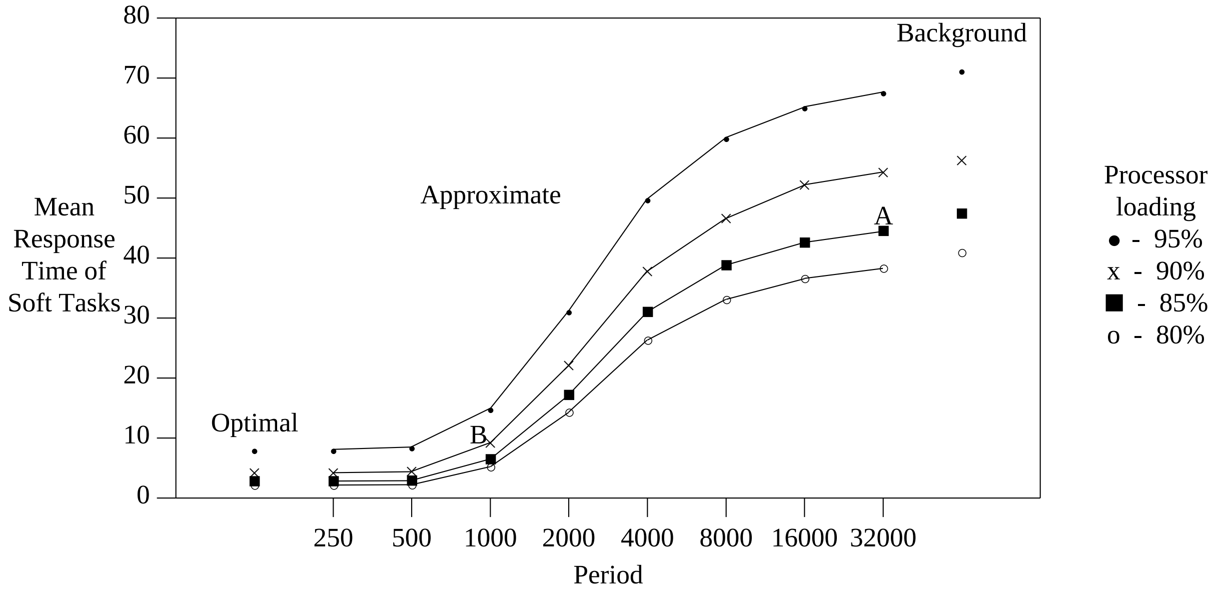

Figure 1. Performance of the approximate slack stealing algorithm

(We note that all the hard task sets used in the test were close to the schedulability bound. This represents a severe test of slack scheduling algorithms. Under these conditions, algorithms which cannot take advantage of favourable task phasing fair little better than background processing.)

Figure 1 also shows that the computation of exact slack does not need to be co-ordinated with hard task completion in order to ensure highly responsive soft task processing. This is an important result as it allows flexibility in scheduling the computation of slack, (for example as a soft task scheduled in slack time).

5.2. Overheads

The memory requirements of the approximate algorithm are generally much lower than those of the static Slack Stealer of Lehoczky and Ramos-Thuel. The static Slack Stealer uses a table of values for each invocation of each task over the hyperperiod. By comparison, the dynamic algorithm requires memory for the task which computes the exact slack and storage for the *n* values of c_i, x_i and d_i. Both static and approximate algorithms also require a set of *n* counters to record the slack available each priority level.

The execution time overhead of the approximate algorithm is much higher than that of the static Slack Stealer due to the run-time computation of slack. This overhead increases the processor loading, reducing the responsiveness of soft tasks below the theoretical levels shown in figure 1. We now examine to what degree, this overhead can be tolerated and still improve response times. Suppose we have a real-time system comprising one of the 85% loading task sets described above. Using an algorithm which does not exploit run-time information on task phasings, we can expect the mean response time of the soft tasks to be approximately 45 ticks (point A, figure 1). We next assume that the computation of slack can be implemented as a task with period and deadline of 1000 ticks and a processor loading of 5%. Using the approximate algorithm and including this overhead, we can expect the mean response time of soft tasks to lie in the vicinity of point B on the 90% load curve of figure 1. This represents a response time of 9 ticks, a factor of five improvement.

We measured the mean time to compute the exact slack for *all* priority levels for sets of 10,15 and 20 hard deadline tasks. (The task sets were generated as described in section 5.1). Timings were recorded using a Sun Sparc workstation. The mean computation time is given in the table below:

Timings	
No. Tasks	Time (ms)
10	1.2
15	4.7
20	9.8

Assuming that the task sets are typical of real task sets with 1 tick = 1ms, the above figures represent a processor loading of between 0.1% and 1%. This is well below the level at which computation of exact slack can be expected to improve soft task response times.

These results show that although algorithm (3) is pseudo-polynomial, there are many hard tasks sets for which the overheads of dynamically computing slack are justifiable.

6. Summary and Conclusions

This paper extended previous research on slack stealing algorithms. New analysis was presented which allows the slack available on hard deadline periodic and hard deadline sporadic tasks to be calculated. This new analysis forms the basis of a dynamic slack stealing algorithm which we proved to be optimal. We augmented our analysis to cater for tasks which have release jitter, synchronisation, stochastic execution times and arbitrary deadlines. We then constructed an approximate algorithm and compared its performance to the optimum. Simulation studies indicated that the approximate algorithm can offer significant improvements over methods which do not exploit run-time phasing information. Furthermore, for the task sets studied, the overhead of dynamically computing slack was shown to be justified.

In light of these results, we intend to carry out further research into approximate slack stealing algorithms using the Distributed Real-time Execution Environment (DrTEE) 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 authors would also like to thank the anonymous reviewers for their comments on an earlier draft of this paper.

References

1. N. Audsley, A. Burns, M. Richardson, K. Tindell and A.J. Wellings, ‘‘Applying New Scheduling Theory to Static Priority Pre-emptive Scheduling’’, *Software Enginnering Journal* (to appear).
2. 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).
3. N. C. Audsley, A. Burns, M. F. Richardson and A. J. Wellings, ‘‘Incorporating Unbounded Algorithms Into Predictable Real-Time Systems’’, *Computer Systems Science and Engineering* **8**(3), pp. 80-89 (April 1993).
4. A. Dix, R. F. Stone and H. S. M. Zedan, ‘‘Design Issues for Reliable Time-Critical Systems’’, *Proceedings of Workshop on Real-Time Systems*, University of York (September 1989).
5. 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).
6. 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).
7. 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).
8. 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).
9. J. Lehoczky, L. Sha and Y. Ding, ‘‘The Rate-Monotonic Scheduling Algorithm: Exact Characterization and Average Case Behaviour’’, *Proceedings IEEE Real-Time Systems Symposium*, Santa Monica, California, pp. 166-171, IEEE Computer Society Press (5-7 December 1989).
10. J. Y. T. Leung and J. Whitehead, ‘‘On the Complexity of Fixed-Priority Scheduling of Periodic, Real-Time Tasks’’, *Performance Evaluation (Netherlands)* **2**(4), pp. 237-250 (December 1982).
11. 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).
12. 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).
13. 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).
14. 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).
15. J. A. Stankovic and K. Ramamritham, ‘‘The Spring Kernel: A New Paradigm for Real-Time Operating Systems’’, COINS Technical Report 88-97, Department of Computer and Information Science, University of Massachusetts at Amherst (November 7, 1988).
16. K. Tindell, ‘‘An Extendible Approach for Analysing Fixed Priority Hard Real-Time Tasks’’, YCS189, Department of Computer Science, University of York (December 1992).
17. K. Tindell, A. Burns and A.J. Wellings, ‘‘An Extendible Approach for Analysing Fixed Priority Hard Real-Time Tasks’’, *Journal of Real Time Systems* (to appear).