OPTIMAL PRIORITY ASSIGNMENT AND FEASIBILITY OF STATIC PRIORITY TASKS WITH ARBITRARY START TIMES

N. C. Audsley

November 1991

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

ABSTRACT

Within the hard real-time community, static priority pre-emptive scheduling is receiving increased attention. Current optimal priority assignment schemes require that at some point in the system lifetime all tasks must be released simultaneously. Two main optimal priority assignment schemes have been proposed: rate-monotonic, where task period equals deadline, and deadline-monotonic where task deadline maybe less than period. When tasks are permitted to have arbitrary start times, a common release time between all tasks in a task set may not occur. In this eventuality, both rate-monotonic and deadline-monotonic priority assignments cease to be optimal. This paper presents an method of determining if the tasks with arbitrary release times will ever share a common release time. This has complexity O($m\ log_e m$) in the longest task period. Also, an optimal priority assignment method is given, of complexity O($n^2 + n$) in the number of tasks. Finally, an efficient feasibility test is presented, for those task sets whose tasks do not share a common release time.

1. INTRODUCTION

Recently, scheduling theory has enjoyed renewed interest within the real-time computing community. In particular, the problem of scheduling hard real-time systems, where missing a single task deadline can have disastrous consequences, has motivated a renaissance of static priority pre-emptive scheduling for periodic task sets. Within this discipline, for every periodic release of each task, sufficient processor resource must be guaranteed to be assigned to a task for it to meet its computational requirement before its deadline.

Much work has been carried out in this area to provide optimal priority assignment to tasks and to check the feasibility of a task set [Liu73, Leh89, Leu80, Leu82]. Much of this work has used a task model which constrains all tasks to have a simultaneous release time or *critical instant* [Liu73]. This simplifies both priority assignment and determination of feasibility. The main focus of this paper is to remove this constraint and to permit periodic tasks to have arbitrary start times. Under these conditions, known priority assignment strategies are, in general, no longer optimal, with known feasibility tests either inefficient or of limited use.

The relaxation towards arbitrary start times for tasks raises a number of key issues:

(i) determining whether tasks with arbitrary start times are ever, within the system lifetime, released simultaneously;

(ii) provision of an optimal priority assignment mechanism;

(iii) determining, in a sufficient and necessary manner, feasibility of a task set with arbitrary start times.

Formally, for a task set of cardinality n, $\Delta = \{\tau_1, \tau_2, ..., \tau_i, ... \tau_n\}$, each task τ_i is assigned priority i (where 1 is the highest priority and n the lowest). Those tasks that have yet to be assigned priority have upper case subscripts, i.e. τ_A, τ_B .. etc. Each τ_i has period T_i, deadline D_i, computation time C_i and offset O_i. The latter represents the time at which the first request for τ_i occurs, assuming that the system commences execution at time 0. Each τ_i makes an initial *request*, or *release*, at O_i, and then periodically every T_i time units. For all task deadlines to be guaranteed, for each request of a τ_i at t, the processor must be allocated to τ_i for C_i units in $[t, t + D_i)$, where the interval is composed of the D_i discrete time units $t, t+1, ..., t+D_i-1$.

Within this general model, we make a number of assumptions:

(i) at any point in time, the highest priority runnable task is allocated the processor;

(ii) the cost of pre-emption is zero;

(iii) pre-emptions occur at the boundaries between discrete time units;

(iv) all tasks are independent: requests for any τ_i are not dependent upon the initiation or completion of any other task;

(v) the computation requirement C_i of task τ_i is constant for each release;

(vi) tasks cannot be blocked;

(vii) tasks cannot voluntarily suspend themselves.

Major contributions in static priority scheduling theory for single processor systems adhering to this model stem from the seminal paper by Liu and Layland [Liu73] describing the rate-monotonic priority assignment for use with a simple execution model: pre-emptive static priority. Rate-monotonic priority assignment is optimal (amongst all possible assignments) for periodic task sets where each τ_i has $O_i = 0$ and $C_i \leq D_i = T_i$. Priorities are assigned inversely proportional to period, i.e. the highest priority is assigned to the

shortest period task. This assignment is of $O(n \log_2 n)$, i.e. that of ordering the tasks in Δ by period.

Leung *et al* [Leu82] describe the deadline monotonic priority assignment for task sets where each τ_i has $C_i \leq D_i \leq T_i$. Again, this priority assignment is optimal, assuming all $O_i = 0$. Priorities are assigned inversely proportional to deadline, with the shortest deadline task given the highest priority. Deadline-monotonic priority assignment is equivalent to rate-monotonic when all $\tau_i \in \Delta$ have $T_i = D_i$.

The problem of determining the feasibility (also termed schedulability) of a task set under a given priority assignment is well studied. For rate-monotonic priority assignment Liu and Layland presented a simple task set utilisation threshold test which is sufficient and not necessary [Liu73]. Hence task sets declared infeasible by the test may at runtime prove feasible. Sufficient and necessary tests, proposed by Joseph *et al* [Jos86] and Lehoczky *et al* [Leh89] are also available.

For deadline-monotonic priority assignment, Leung *et al* propose a feasibility test based upon construction of a schedule until the longest deadline amongst all tasks in Δ. If all deadlines are met in the finite schedule, deadlines will always be met. Other feasibility tests include those by Lehoczky [Leh90], Audsley *et al* [Aud90, Aud91b] and Nassor *et al* [Nas91]. These tests assume all tasks are initially released simultaneously (at time 0) and examine the first deadline of each task. If this deadline is met, all subsequent deadlines for that task will be met, since the computation demand by other higher priority tasks is at a maximum [1].

When the $O_i = 0$ constraint is lifted, and the tasks in Δ never have a simultaneous release at runtime, the rate-monotonic priority assignment is no longer optimal. For example, consider tasks with equal periods. Rate-monotonic priority assignment dictates that the assignment of priorities of such tasks is arbitrary [Liu73]. Consider the priority assignment of the tasks defined in example 1.

Example 1:

$$\tau_A : O_A = 0 ; \; C_A = 3 ; \; D_A = 8 ; \; T_A = 8$$

$$\tau_B : O_B = 10 ; \; C_B = 1 ; \; D_B = 12 ; \; T_B = 12$$

$$\tau_C : O_C = 0 ; \; C_C = 6 ; \; D_C = 12 ; \; T_C = 12$$

■

Task τ_A is given the highest priority. The assignment of priorities for τ_B and τ_C cannot be performed in an arbitrary manner. If τ_B is assigned a higher priority than τ_C, at time 12 τ_C misses a deadline. However, if τ_C is assigned higher priority than τ_B, all tasks meet their deadlines. No general rule regarding the assignment of priorities to tasks with equal periods has been identified in the literature.

If tasks are permitted arbitrary start times, with no simultaneous release of all tasks ever occurring at runtime, deadline-monotonic priority assignment is also no longer optimal. For example, consider the task set in example 2.

---

1. The simultaneous release of all tasks can occur at any time in the systems lifetime, not necessarily at time 0.

Example 2:

$\tau_A : O_A = 2 ; C_A = 2 ; D_A = 3 ; T_A = 4$

$\tau_B : O_B = 0 ; C_B = 3 ; D_B = 4 ; T_B = 8$

■

Deadline-monotonic priority ordering dictates that τ_A is assigned a higher priority than τ_B. However this leads to the deadline of τ_B being missed at time 4 (then successively at 12, 20, 28 ...). When priorities are reversed, (contrary to deadline-monotonic priority assignment) task deadlines are met. Indeed, according to Leung, who describes tasks with arbitrary offsets as asynchronous [Leu82]:

> *"At the present time no priority assignment has been found which is optimal for an arbitrary asynchronous system."*

When considering the feasibility for tasks with arbitrary start times and no simultaneous release, Lehoczky's [Leh90] and Audsley's tests [Aud91b], developed for deadline-monotonic priority assignments, are in general, no longer necessary[2]. For both tests, we can assume, for the purposes of determining feasibility, that all $O_i = 0$ forcing a simultaneous release onto the task set. This creates a maximum computation demand on the processor producing the hardest scenario for the tasks to meet their deadlines. Of course, at runtime a simultaneous release does not occur. Hence, by assuming all $O_i = 0$ the feasibility tests become sufficient and not necessary.

Leung's test, that of construction of a schedule, has been extended to cope with arbitrary task start times [Leu82]: a task set is feasible if all deadlines are met in $[s, 2P)$ (where $s = max \{ O_1, O_2, \ldots, O_n \}$ and $P = lcm \{ T_1, T_2, \ldots, T_i \}$ [3]. The feasibility task consists of constructing a schedule for this interval. In practice, this requires the construction of a schedule for the interval $[0, 2P)$. This approach is sufficient and necessary but can be inefficient.

Clearly, an optimal priority assignment and an efficient sufficient and necessary test for periodic task sets with arbitrary start times remain open issues. This paper focuses on these issues, proposing an integrated approach to priority assignment and schedulability. This is in contrast to previous work cited above, which separately addresses the issues of priority assignment and feasibility testing. The integrated approach uses feasibility testing as part of task priority assignment. Thus, when a valid priority assignment is found, the task set is known to be feasible. Additionally, the problem of determining if all tasks in a task set will ever have a common release time is addressed.

The remainder of this paper is arranged as follows. The next section provides a method for determining if and when tasks share a common release time. Section 3 discusses optimal priority assignment. Section 4 defines a minimum interval that must be considered when determining the feasibility of a task. Section 5 proposes a sufficient and necessary test that combines priority assignment and feasibility to provide an integrated approach. Concluding remarks are offered in section 6.

---

2. Both these tests are applicable to any fixed priority assignment rule. They are not limited to task priorities assigned by the deadline-monotonic method.

3. *lcm* - least common multiple of a set of integers.

2. CRITICAL INSTANTS

The feasibility tests for static priority periodic task systems cited in section 1 are founded on the assumption that at some point of time during the lifetime of the system all tasks share a common release time. This is termed the *critical instant* [Liu73] [4]. When a critical instant occurs, the worst-case load on the processor becomes apparent. In a static priority pre-emptive system, this forms the hardest time for all task deadlines to be guaranteed: Theorem 1 in [Liu73] shows that if deadlines can be guaranteed for releases starting at a critical instant, they can be guaranteed, by implication, for the lifetime of the system.

When tasks have offsets, it is difficult, by inspection of task timing characteristics alone, to determine if a critical instant will occur during system execution. If a critical instant does not exist, different priority assignments and feasibility tests need to be employed for optimality to be achieved.

The following discussion details how to determine if a task set contains a critical instant. Firstly, we constrain the problem to that of determining if two tasks share a common release time. This result is developed in the subsequent section to show how to decide if a task set of arbitrary cardinality has a critical instant.

2.1. Two-Tasks

Consider $\Delta = \{\tau_1, \tau_2\}$. We note that for any $\tau_i \in \Delta$, without loss of generality, if $O_i = T_i$, O_i can be set to 0. If $O_i > T_i$, we can reduce the offset to be $O_i - \lfloor O_i/T_i \rfloor T_i$. Hence, we can state that $O_1 < T_1$ and $O_2 < T_2$. Thus, a critical instant will occur if

$$O_1 + aT_1 = O_2 + bT_2 \qquad [a, b \in Z]$$

Without loss of generality, we can reduce the smallest offset to be 0. Let $O_1 < O_2$. Thus the critical instant condition becomes:

$$aT_1 = O_2' + bT_2 \qquad [a, b \in Z \; ; \; O_2' = O_2 - O_1] \tag{1}$$

This is a linear diophantine equation in two variables. Equation (1) holds if and only if the greatest common divisor (*gcd*) of T_1 and T_2 divides exactly into O_2' [Jac75]. Thus, a critical instant exists if and only if

$$O_2' = h \; gcd(T_1, T_2) \quad [h \in Z] \tag{2}$$

We note that the *gcd* can be found using Euclid's Algorithm (algorithm 1.1.E in [Knu73]). This has complexity $O(log_e max\{T_1, T_2\})$.

Consider the task set given in example 3.

Example 3:

$\tau_A : \; O_A = 3 \; ; \; T_A = 42$

$\tau_B : \; O_B = 66 \; ; \; T_B = 147$

■

We let $O_A' = 0$ and $O_B' = 63$. Since $gcd(42, 147) = 21$ and $O_B' = 3 \times 21$, a common release time exists between the tasks. Indeed, releases of τ_A occur at 0, 42, 84, 126, 168, 210,.. Releases of τ_B occur at 63, 210,.. Common release times are 210, 594, 798,...

---

4. A *critical instant* refers to a simultaneous release of all tasks in Δ. A simultaneous release of a subset of tasks in Δ is termed a *common release time.*

2.2. Many Tasks

We extend the problem to $\Delta = \{\tau_1, \tau_2 \ldots \tau_n\}$. Two approachs are identified.

Naive Approach

Compare every release of τ_i $(1 \le i \le n)$ with every τ_j $(j = 1 \ to \ i-1, i+1 \ to \ n)$. This requires a total number of comparisons given by:

$$\sum_{i=1}^{n} \sum_{j=i+1}^{n} \left\lceil \frac{lcm(T_i, T_j)}{T_i} \right\rceil \tag{3}$$

In the worst case $lcm(T_i, T_j) = T_i T_j$, giving:

$$\left\lceil \frac{lcm(T_i, T_j)}{T_i} \right\rceil = \left\lceil \frac{T_i T_j}{T_i} \right\rceil = T_j$$

Therefore, equation (3) can be restated:

$$\sum_{i=1}^{n} \sum_{j=i+1}^{n} T_j = \sum_{i=2}^{n} (i-1) T_i$$

The complexity of this approach is $O(\sum_{i=2}^{n} (i-1) T_i)$. Hence, in general the approach is exponential.

Efficient Approach

We use the results of section 2.1 to establish a more efficient approach. Consider two tasks, τ_A and τ_B, that share a critical instant (i.e. use the method in section 2.1). Assume that $O_A = 0$ and $0 < O_B < T_B$. We can form a hybrid task, τ_{AB} to represent the critical instant of the two tasks throughout the system lifetime. The period of τ_{AB} will be the time between successive critical instants, with O_{AB} being the time from 0 to the first critical instant (hence $O_{AB} \ge O_B$). Thus we replace two tasks with one.

We now identify the period and offset of τ_{AB}. The period, T_{AB}, is the *lcm* of T_A and T_B [5]:

$$T_{AB} = \frac{T_A \, T_B}{gcd(T_A, T_B)} \tag{4}$$

Since the denominator has already been found (when determining if a critical instant exists between τ_A and τ_B) the calculation of T_{AB} is trivial.

The offset can be calculated using the Euclidean algorithm for finding the *gcd* of two integers. This can be adapted to find two integers x and y such that (algorithm 1.2.1.E in[Knu73]):

$$xT_A + yT_B = gcd(T_A, T_B)$$

We note that x and y represent one of an infinite set of solutions for this expression. Multiplying through by h:

---

5. The *lcm* of two integers a, b, is related to the *gcd* of a, b, by $ab/gcd(a,b) = lcm(a,b)$. For further details see theorem 2.5 in [Ros85].

$$(hx)T_A + (hy)T_B = h\ gcd(T_A, T_B)$$

Since by equation (2) $O_B = h\ gcd(T_A, T_B)$:

$$(hx)T_A + (hy)T_B = O_B \tag{5}$$

We note that one of x and y will be negative, the other positive. If the multiplier of T_A is negative, we have effectively extrapolated a common release time of the tasks prior to time 0. To find a common release time after 0 we need to make the multiplier of T_A positive and that of T_B negative. This is achieved by noting that common release times occur with period T_{AB}. Thus, we add kT_{AB} to the first term of equation (5) subtract the same quantity from the second term:

$$(hxT_A + kT_{AB}) + (hyT_B - kT_{AB}) = O_B \tag{6}$$

where

$$k = \left\lceil \frac{|x|hT_A}{T_{AB}} \right\rceil \tag{7}$$

Since the first term of equation (6) is positive and the second negative, we may rearrange:

$$(hx)T_A + kT_{AB} = (hy)T_B + kT_{AB} + O_B \tag{8}$$

Either side of equation (8) defines a simultaneous release of τ_A and τ_B after 0. Let t equate to one side of equation (8):

$$t = (hx)T_A + kT_{AB} \tag{9}$$

It may occur that $t > T_{AB}$ or that $t > lcm(T_A, T_B)$. Therefore we form O_{AB} by subtracting from t a multiple of $lcm(T_A, T_B)$, the resulting quantity representing the first release of τ_{AB}, that is O_{AB}:

$$O_{AB} = t - \left\lfloor \frac{t}{T_{AB}} \right\rfloor T_{AB} \tag{10}$$

where k and t are defined by equations (7) and (9) respectively. We note that $O_{AB} \geq O_B$ as $T_{AB} \geq T_B$ and $O_B < T_B$.

We have shown that if two tasks τ_A and τ_B have a critical instant, we can form the hybrid task τ_{AB} with period T_{AB} and offset O_{AB} (given by equations (4) and (10) respectively). τ_{AB} defines the common release times of τ_A and τ_B.

Successively, we can apply this technique to the remaining tasks in Δ: we compare τ_{AB} and $\tau_C \in \Delta - \{\tau_A, \tau_B\}$. If these two tasks share a critical instant (by section 2.1), we formulate τ_{ABC}. Therefore, we can determine whether the tasks in Δ share a critical instant. If this is so, the resulting hybrid task, by its offset and period, will characterise the common release times of the tasks in Δ.

The complexity of this method is now considered. For each combination of two tasks (or one hybrid and one task) the cost involves:

(i) deciding whether a critical instant exists for the two tasks;

(ii) formulating the hybrid task: this is trivial by equations (4) and (10).

Since we may create a maximum of $n - 1$ hybrid tasks for a task set containing n tasks, the complexity of finding if the tasks in Δ share a critical instant is, as $n \to \infty$, $O(n\ log_e max\{T_A, T_B, T_C, ..\})$. This is clearly more efficient (and less exponential) than the "Naive Approach".

We return to example 3 noting that $O_A = 0$ and $O_B = 63$. In section 2.1 it was shown that the tasks had a common release time. By evaluation of the schedule the first such time is 210. We now derive the hybrid task τ_{AB}. Evaluating T_{AB} according to equation (4), noting that $gcd(T_A, T_B) = 21$:

$$T_{AB} = \frac{T_A T_B}{gcd(T_A, T_B)} = \frac{6174}{21} = 294$$

Evaluating k in equation (7), noting that $h = 3$:

$$k = \left\lceil \frac{|x| hT_A}{T_{AB}} \right\rceil = \left\lceil \frac{3 \times 3 \times 42}{294} \right\rceil = 2$$

Evaluating t according to equation (9):

$$t = (hx)T_A + kT_{AB} = (-9 \times 42) + (2 \times 294) = 210$$

Evaluating O_{AB} according to equation (10):

$$O_{AB} = t - \left\lfloor \frac{t}{T_{AB}} \right\rfloor T_{AB} = 210 - \left\lfloor \frac{210}{294} \right\rfloor 210 = 210$$

Thus the hybrid task has period $T_{AB} = 294$ and offset $O_{AB} = 210$. We note that O_{AB} corresponds to the first common release of τ_A and τ_B.

3. OPTIMAL PRIORITY ASSIGNMENT

In the previous section it was shown how to check if the tasks in Δ will ever be released simultaneously. Tasks sets whose tasks will never undergo such a release are termed *non-critical-instant tasks sets*, Δ^* (the set of all Δ^* is a subset of the set of all Δ). For any Δ^*, section 1 indicated that neither rate-monotonic or deadline-monotonic priority assignments were optimal. We now develop an optimal priority assignment for tasks with arbitrary start times.

Consider $\Delta^* = \{\tau_A, \tau_B, \tau_C, ...\}$ of cardinality n. A priority assignment function maps each task onto a different priority level. For Δ^* there are $n!$ distinct priority assignments over the task set, hence the set of distinct priority assignment functions has cardinality $n!$. This is denoted by $\Phi = \{\Phi_1, \Phi_2, \ldots, \Phi_{n!}\}$. We denote the mapping of a task onto a priority level by

$$\Phi_i(\tau_A) = j$$

where the i^{th} priority assignment function maps τ_A onto priority level j. The inverse mapping, of priority level to task, is denoted:

$$\Phi_i^{-1}(j) = \tau_A$$

When the priority ordering over Δ^* specified by a priority ordering function is feasible, we term that function a *feasible priority assignment function*.

In general, τ_A is feasible (schedulable) if and only if [Aud90, Aud91b]:

$$C_A + I_A \le D_A$$

where I_A represents the execution requirement (interference) of higher priority tasks in the interval defined by the release and deadline of τ_A.

If τ_A is not feasible, and the task timing characteristics cannot be changed (i.e. C_A

cannot be decreased and D_A cannot be increased), the only way to make τ_A feasible is to decrease I_A. This is achieved by changing the priority ordering over Δ^* by using a priority assignment function that reduces the priority of a higher priority task to be lower than τ_A. (Note that we could then promote a lower priority task to be higher than τ_A as long as the new I_A is less than the original.)

Let us now consider the effect on the feasibility of Δ^* (cardinality n) for $\Phi_x \in \Phi$ such that $\Phi_x(\tau_A) = n$. We present two theorems regarding such an assignment.

Theorem 1:

If τ_A is assigned the lowest priority, n, and is infeasible, no priority assignment function that assigns τ_A priority level n produces a feasible assignment.

Proof:

Amongst the $n!$ distinct priority assignment functions, $(n-1)!$ produce an assignment with τ_A at priority level n. For all such assignments, the interference due to tasks of higher priority than τ_A is equal, as the same set of tasks is of higher priority than τ_A in each ordering. Thus if τ_A is infeasible as the lowest priority task by one assignment function, it will be infeasible under the priority ordering of any other function assigning it the lowest priority.

■

Theorem 2

If τ_A is assigned the lowest priority, n, and is feasible, then if a feasible priority ordering for Δ^* exists, an ordering with τ_A assigned the lowest priority exists.

Proof:

Let us assume that an assignment function Φ_y produces the feasible assignment:

$$\Phi_y(\tau_B) = 1, \Phi_y(\tau_C) = 2, , \ldots , \Phi_y(\tau_A) = i , \Phi_y(\tau_D) = i + 1, , \ldots , \Phi_y(\tau_E) = n$$

We note that τ_A is feasible at priority level $i < n$. A second priority assignment function Φ_x defines:

$$\Phi_x(\tau_B) = 1, \Phi_x(\tau_C) = 2, , \ldots , \Phi_x(\tau_D) = i , , \ldots , \Phi_x(\tau_E) = n - 1, \Phi_x(\tau_A) = n$$

Since τ_A is feasible if assigned priority level n (by the theorem), we can assign τ_A to level n. The tasks originally assigned priority levels $i+1 \cdots n$ in Φ_x are promoted 1 place (i.e. the task at priority level $i+1$ is now assigned priority i). Clearly, the tasks assigned levels $1 \cdots i+1$ remain feasible as nothing has changed to affect their feasibility. The tasks originally assigned levels $i+1 \cdots n$ also remain feasible as the interference on them has decreased with τ_A now being of the lowest priority. Since τ_A is feasible at the lowest priority level, at least one feasible priority assignment exists with τ_A as the lowest priority task. The theorem is proved.

■

The above theorems limit considerations to priority level n. We now extend the above two theorems, to consider assignment of arbitrary priorities to tasks, rather than merely priority n.

Theorem 3:

Let the tasks assigned priority levels $i, i+1,.., n$ by assignment function Φ_x be feasible under that priority ordering. If there exists a feasible priority ordering for Δ^*, there exists a feasible priority ordering that assigns the same tasks to levels $i..n$ as Φ_x.

Proof:

We prove the theorem by showing that a feasible priority assignment function Φ_y can be transformed to assign the same tasks to priority levels $i, i+1,.., n$ as Φ_x, whilst preserving the feasibility of Φ_y. The proof is by induction: Φ_y is transformed successively moving tasks $\Phi_x^{-1}(n), \Phi_x^{-1}(n-1),.., \Phi_x^{-1}(i)$ to priority levels $n, n-1,..,i$ under Φ_y.

Base

Let $\Phi_x^{-1}(n) = \tau_A$ and $\Phi_y(\tau_A) = m$, where $m \leq n$. By theorem 2 we can move τ_A to the assigned level (n) under Φ_k without altering the feasibility of Δ^*.

Inductive Hypothesis

We assume that the tasks assigned to priority levels $n-1, n-2, .., i+1$ under Φ_x are moved to levels $n-1, n-2, .., i+1$ under Φ_y. Δ^* is assumed to remain feasible.

Inductive Step

Let $\Phi_x^{-1}(i) = \tau_B$ and $\Phi_y(\tau_B) = m$, where $m \leq i$ (since the reassignment of priority levels $n, n-1, .., i+1$ has promoted τ_B to have a priority of between 1 and i). Under both orderings, the tasks assigned to priority levels $i+1, .., n$ are identical. Task τ_B is reassigned in Φ_y to level i. We know (by Φ_x) that τ_B is feasible at this level (assuming that tasks assigned to levels $i+1..n$ are identical under Φ_x and Φ_y).

After the reassignment, tasks at levels $1..i-1$ remain feasible, as their respective interferences are no greater than before the reassignment and therefore must remain feasible. This proves the theorem.

■

We now use the above theorems in developing an optimal static priority assignment scheme which assigns tasks to priority levels $n, n-1,.., 1$ in order. Only if a feasible assignment can be made to priority level i do we proceed to priority level $i-1$. A generic approach for assigning to level i $(1 < i \leq n)$ is now given.

Task Assignment to Priority Level i

We assume that priority levels $n..i+1$ have been assigned such that the tasks assigned to those levels are feasible. Let the task assigned to priority level j be given by $\Psi(j)$. We note that $\Psi(j)$ is only defined for $i < j \leq n$. Let the set Δ^{i+1} be composed of those tasks in Δ^* that have been assigned priority levels $n, n-1,..,i+1$ (cardinality of Δ^{i+1} is $n-i$). For each task τ_A in Δ^*-Δ^{i+1} (i.e. the set of unassigned tasks) we select a single $\Phi_k \in \Phi$ such that

$$\forall l : i < l \leq n : \Phi_k^{-1}(l) = \Psi(l)$$

The set of such Φ_k for priority level i is termed Φ^i. Set Φ^i contains i elements of Φ, each of which assigns priority levels n to $i+1$ identically, differing in their assignment of priority level i (each Φ_k assigns a different $\tau_r \in \Delta^*$-Δ^{i+1} to level i).

For each $\Phi_k \in \Phi^i$ we check the feasibility of the task assigned priority level i (by virtue of reaching the assignment of priority level i we know that tasks assigned to levels $n, n-1,..,i+1$ are feasible). Two cases are identified:

(i) all tasks are infeasible when assigned priority level i;

(ii) one or more tasks are feasible when assigned priority level i.

In the first case, we know by theorem 1 that no feasible priority assignment exists for Δ^* and so the task set is infeasible. In the second case, we may arbitrarily select one of the feasible tasks noting by theorem 3 that if a feasible priority assignment for Δ^* exists, one will exist with the selected task assigned priority level i. Thus, $\Psi(i)$ is defined. We

proceed to the assignment of priority level $i-1$.

Eventually, we reach the assignment for level 1. This is trivial since at this stage onl one task remains to be assigned (i.e. the cardinality of Δ^*-Δ^2 is 1) leaving no choice for priority level 1. The next section gives an efficient implementation for this method of priority assignment.

Algorithmic Implementation

An algorithm implementing the priority assignment method detailed in the previous sections is now given.

Algorithm 1: Optimal Priority Assignment:

```
PriorityAssignment ()
  begin
    Δ = Δ*  -- copy the non-critical instant task set
    for j in (n..1)  -- priority level j
        unassigned = TRUE
        for τA in Δ
            if (feasible(τA, j)) then -- if τA is fesaible
                                      -- for priority level j
                ψ(j) = τA   -- assign τA to priority level j
                Δ = Δ - τA
                unassigned = FALSE
            endif
            if (unassigned)
                exit  -- no feasible priority assignment exists
            endif
        endfor
    endfor
  end
```

■

Firstly, we attempt to find a task τ_A that is feasible at priority level $j = n$. If one is found, then by theorem 2 if a feasible priority assignment function exists, one also exists with τ_A assigned priority level n, i.e. $\Psi(j) = \tau_n$. Next, priority level $j = n - 1$ is now considered. If a task can be found (amongst the $n - 1$ tasks that have not yet been assigned a priority level) that is feasible at priority level $n - 1$, then by theorem 1 we know that if a feasible priority assignment function exists, a feasible priority one also exists with this task assigned priority level $n - 1$. Successively, tasks are found that are feasible at priority levels n to 1. If, for any priority level j a feasible task cannot be found, no feasible priority assignment function exists.

Discussion

This priority assignment scheme is optimal in the sense that if a feasible priority ordering exists for a task set, it will be found by this method. The proof of this assertion lies in theorems 1, 2 and 3.

The complexity of the priority assignment scheme lies in the number of priority

assignment functions examined. This is more readily described by examining the behaviour of algorithm 1. To find a task that is feasible at priority level n involves testing the feasibility of a maximum of n tasks. In general, to find a task feasible when assigned priority level $i \leq n$ requires testing the feasibility of a maximum of i tasks (that is the i tasks that have yet to be assigned a priority). Therefore, across all priority levels, the number of tests required, B, is given by:

$$B = n + (n-1) + (n-2) + ... + (n-(n-2)) + (n-(n-1))$$

Since there are n terms

$$B = n^2 - 1 - 2 - \cdots - (n-1) = n^2 - n - \sum_{i=1}^{n-1} i$$

Since the sum of all integers between 1 and m is $m(m+1)/2$

$$B = n^2 - \frac{(n-1)\,n}{2} = \frac{1}{2}(n^2 + n)$$

Therefore, finding a feasible priority assignment or showing that no such assignment exists requires that a maximum of $n^2 + n$ priority assignment functions be examined. In effect, the feasibility of a maximum of $n^2 + n$ tasks needs to be determined. This is polynomial in n and as such is exponentially more efficient than examining all possible $n!$ priority orderings.

The priority assignment technique detailed above relies upon a sufficient and necessary feasibility test being available. Such a test is developed in the following sections.

4. FEASIBILITY INTERVAL

Feasibility testing of a task set requires the definition of an interval over which that testing needs to occur. We term this the *feasibility interval*. For tasks which share a critical instant, feasibility can be determined by examining the first deadline of each task after a critical instant. This approach is used in many of the feasibility tests cited in section 1 for static priority task sets. Hence the feasibility interval for each τ_i is $[t, t + D_i)$ where t corresponds to a simultaneous release of all tasks. When arbitrary task start times are permitted, or more specifically when the tasks form a non-critical-instant task set, this approach is not appropriate.

Leung *et al* [Leu80, Leu82, Leu89] showed that an interval of $[O_{\max}, 2P)$ is a sufficient interval (where $O_{\max}$ represents the maximum process offset and P is defined by the *lcm* of all task periods). This interval was established by considering dynamic priority scheduling schemes, such as Earliest Deadline. For static priority schemes we now show that, in general, a smaller interval is sufficient.

We make the initial observation that the feasibility of an individual task in a static priority pre-emptive scheduling scheme depends only upon itself and other tasks of higher priority. Therefore, when determining the lower and upper endpoints for the feasibility interval of $\tau_i \in \Delta^*$ we consider τ_i and those tasks in Δ^* of higher priority. Considering the optimal priority assignment method introduced in section 3, all priority levels $1..i-1$ are unassigned. Therefore, without loss of generality, we arbitrarily assign currently unassigned tasks to those levels. When determining the feasibility interval, the specific assignments of levels $1..i-1$ is not important, only that all unassigned tasks have priority greater than τ_i.

To aid the following discussion, we refine the offsets of the tasks in Δ^* without

altering their relative phasing. The following theorem expresses this refinement.

Theorem 4:

When determining the feasibility interval for $\tau_i \in \Delta^*$, the timing characteristics of tasks $\tau_1, \tau_2, ..., \tau_{i-1}$ can be rearranged, without altering the relative phasing of those task releases, such that:

(a) $min\,(O_1 \cdots O_i) = O_i = 0$

(b) $\forall\ \tau_j \in \{\tau_1, .. \tau_i\} : O_j \in [O_i, O_i + P_i)$

Proof:

Let the offsets of tasks $\tau_1, \tau_2, .. \tau_i$ be represented by $(O_1, O_2, \ldots, O_i)$. We can form condition (a) by adding an amount l_j to O_j for any task $\tau_j \in \{\tau_1, \tau_2, .., \tau_{i-1}\}$ where $O_j < O_i$. To preserve the relative phasing of the tasks, l_j is a multiple of T_j. Also, l_j must be at least $O_i - O_j$. Therefore, we define

$$l_j = \left\lceil \frac{O_i - O_j}{T_j} \right\rceil \tag{11}$$

Thus

$$l_j T_j \geq O_i - O_j$$

Therefore we have

$$O_i \leq O_j + l_j T_j$$

where l_j is defined by equation (11). Since $T_j \leq P_i$ and $O_j < O_i$,

$$O_i \leq O_j + l_j T_j < O_i + P_i$$

Hence we can transform $(O_i \cdots O_i)$ into $(O_1' \; O_2' \cdots O_i')$ by the function:

$$O_j' = \begin{cases} \text{if } O_i > O_j \quad O_j' = \left\lceil \dfrac{O_i - O_j}{T_j} \right\rceil T_j + O_j \\ \text{if } O_i \leq O_j \quad O_j' = O_j \end{cases}$$

We note that the transformation of O_i (to O_i) is defined by the function, although not necessary. Once transformed to meet (a), the task set also meets (b), since all $O_j < T_j$. Without loss of generality, we can now assign $O_{\min} = 0$ with all other offsets reduced by $O_{\min}$. We note task phasing is not affected.

■

With task timing characteristics rearranged according to Theorem 4, we can now establish the feasibility interval for τ_i. Firstly, we define the initial stabilisation time at the start of the task set execution.

Definition 1:

The initial stabilisation time, S_j, of task τ_j, is the time after which the execution of the task set repeats exactly every P_j with respect to tasks $\tau_1, \tau_2, ..., \tau_j$.

■

For a task set with a critical instant, that is where all $O_j = 0$ for $1 \leq j \leq i$, the initial stabilisation time is 0 [Liu73]. In general, the initial stabilisation time is given by the following lemma.

Lemma 1:
The initial stabilisation time, S_j for τ_j is given by:

$$S_j = \left\lceil \frac{O_{\max} - O_j}{T_j} \right\rceil T_j = \left\lceil \frac{O_{\max}}{T_j} \right\rceil T_j$$

$$\textit{where} \quad O_{\max} = max\,(O_1, O_2, \ldots, O_{j-1})$$

$$\textit{assuming} \quad O_j = 0$$

■

The above lemma equates the initial stabilisation time for a task to be the first release of τ_j at or after $O_{\max}$.

Now we show a fundamental characteristic of the run-time behaviour of periodic static priority task sets, namely that schedules repeat exactly at intervals equal to the *lcm* of the task periods, after the initial stabilisation phase. More specifically, for any τ_j $(1 \le j \le n)$ the schedule repeats at intervals defined by P_j. Hence the schedule repeats every T_1 units with respect to τ_1; every $lcm(T_1, T_2)$ units with respect to τ_2 etc. This is formalised in the following theorem.

Theorem 5:
For all tasks τ_j, $1 \le j \le n$, the execution of τ_j at time t, denoted $exec(\tau_j, t)$, where $t \ge S_j$, implies $exec(\tau_j, t + kP_j)$ for $0 \le k \le \infty$.

Proof: (including proof of lemma 1)
Consider the behaviour of the highest priority task τ_1. It executes for the first C_1 units in any interval $\left[O_1 + kT_1, O_1 + kT_1 + D_1\right)$ for $0 \le k \le \infty$. Therefore, the behaviour of τ_1 is static, in that for every time t_1 that the task executes, it will also execute at $t_1 + P_1$. Hence,

$$exec(\tau_1, t_1) \Rightarrow exec(\tau_1, t_1 + kP_1) \quad 0 \le k \le \infty \quad t_1 \ge S_1$$

We note that P_1 is exactly T_1.
The behaviour of τ_2 (the second highest priority) can be expressed in a similar manner. After the initial stabilisation time, τ_2 executes in the first C_2 time units in every interval $[O_2 + kT_2, O_2 + kT_2 + D_2)$ for $0 \le k \le \infty$, which will not be used by any higher priority task, namely τ_1. Therefore, since the times that τ_1 executes are already determined, and τ_1 has been released, we can assert:

$$exec(\tau_2, t_2) \Rightarrow exec(\tau_2, t_2 + kP_2) \quad 0 \le k \le \infty \quad t_2 \ge S_2$$

The argument can be continued until τ_i is reached. This task will reserve the first C_i units of computation time that are not required by any higher priority task. Thus,

$$exec(\tau_i, t_i) \Rightarrow exec(\tau_i, t_i + kP_i) \quad 0 \le k \le \infty \quad t_i \ge S_i \qquad \text{(a)}$$

Therefore, we have built up the static requirements of all tasks, assuming all higher priority tasks have been released.
The only time that the assertion (a) does not hold is if at time t_i, there exists a higher priority task τ_j that has not yet been initially released, (i.e. $t_i < O_j$). In this case, it may occur that $exec(\tau_i, t)$, but that $exec(\tau_j, t_i + kT_i)$ as $t_i < O_j \le t_i + kT_i$.
This can only occur if τ_j has not been released, which contradicts Theorem 5

and Lemma 1. Hence Theorem 5 holds given Lemma 1.

We now proceed to prove Lemma 1.
All tasks are initially released before or at $O_{\max}$ (assuming offset refinement by Theorem 4). However, this is not a sufficient value for S_i. Consider a release of τ_i before $O_{\max}$ with a deadline after $O_{\max}$. That is:

$$O_i + nT_i < O_{\max} < O_i + nT_i + D_i \qquad \text{(b)}$$

Since τ_i is released before some higher priority tasks, this could lead to a time t such that

$$O_{\max} < t < O_i + nT_i + D_i$$

which is idle but with the following condition holding:

$$exec(\tau_i, t + mT_i) \quad m \geq 1$$

That is, due to τ_i running before all higher priority tasks had not been released, τ_i completed its execution defined by (b) early in comparison to corresponding executions in subsequent P_i periods. This cannot occur in releases of τ_i beginning at or after $O_{\max}$. Therefore S_i corresponds to the first release of τ_i after $O_{\max}$, given in Lemma 1.
Thus, Theorem 5 and Lemma 1 are proved.

■

Theorem 5 and Lemma 1 can now be restated to give the feasibility interval for τ_i.

Theorem 6:
Task τ_i is feasible if and only if the deadlines corresponding to releases of the task in $[S_i, S_i + P_i)$ are met.

Proof: By Theorem 5 any $\tau_i \in$ c that executes at time $t \in [S_i, S_i + P_i)$ will also execute at $t + P_i$. Therefore, the schedules in the following intervals will be identical (with respect to $\tau_1, \tau_2, .. \tau_i$):

$$[S_i, S_i + P_i)$$

$$[S_i + P_i, S_i + 2P_i)$$

$$\ldots$$

$$[S_i + mP_i, S_i + (m + 1)P_i)$$

If a task deadline is missed at d from the beginning of one interval, it will be missed at d from the beginning of all such intervals. Therefore it is sufficient to check the deadlines of one interval only, so proving Theorem 6.

■

Discussion

The feasibility intervals defined in the above sections are shorter than those proposed by Leung *et al* [Leu80, Leu82]. The latter interval is designed for testing earliest deadline feasibility. Therefore, determining static feasibility requires less evaluation than for dynamic in two main ways. Firstly, the latter requires twice the *lcm* to be examined. Secondly, dynamic feasibility requires the *lcm* of all task periods, whilst Theorem 6 only requires the *lcm* to consider the periods of tasks with greater or equal priority to the task whose feasibility is being considered.

The interval derived in this section will be exponentially less than that proposed by Leung for most task sets. Consider 5 tasks with relatively co-prime periods, i.e. $T_1 = 9, T_2 = 11, T_3 = 13, T_4 = 17, T_5 = 29$. Under Leung's method, each task is examined for twice the *lcm* of all periods [6], hence the feasibility interval for each task is 1,268,982 long, 6,344,910 in total (for the 5 tasks). The interval derived in the previous section has total length T_1 for τ_1, T_1T_2 for τ_2 etc. This evaluates to a total of

$$9 + (9 \times 11) + (9 \times 11 \times 13) + (9 \times 11 \times 13 \times 17) + (9 \times 11 \times 13 \times 17 \times 29) = 657705$$

This is approximately 10% of the total length of Leung's interval. In general, the length of either interval is at a maximum when all task periods are co-prime. In this case, Leung's interval for n tasks becomes

$$Leung\_Interval\_Total \;=\; 2n(T_1T_2...T_n) \tag{12}$$

The interval derived above becomes

$$New\_Interval\_Total \;=\; T_1 + T_1T_2 + ... + T_1T_2 \cdots T_n \tag{13}$$

If all task periods are unitary, we have $Leung\_Interval\_Total = 2n$ and $New\_Interval\_Total = n$. Let all task periods be mutually co-prime, with $T_i < T_{i+1}$ $(1 \le i < n)$. We can state that:

$$Leung\_Interval\_Total \;>\; New\_Interval\_Total \tag{14}$$

Substituting (12) and (13):

$$2n(T_1T_2 \cdots T_n) \;>\; T_1 + T_1T_2 + ... + T_1T_2 \cdots T_n$$

Taking $T_1T_2 \cdots T_n$ from each side:

$$2(n-1)(T_1T_2 \cdots T_n) \;>\; T_1 + T_1T_2 + ... + T_1T_2 \cdots T_{n-1}$$

The left hand side can be expanded to $n-1$ terms of $2(T_1T_2 \cdots T_n)$, giving $n-1$ terms on each side. Comparing terms, each $2(T_1T_2 \cdots T_n)$ is clearly greater than corresponding terms on the right hand side. Thus, equation (14) holds. The difference between left and right hand sides is given by:

$$Leung\_Interval\_Total - New\_Interval\_Total \;=$$

$$(2nT_2 \cdots T_n - 1)T_1 + (2nT_3 \cdots T_n - 1)T_1T_2 + ... + (2nT_n - 1)T_1T_2 \cdots T_{n-1}$$

Thus, in general, *New_Interval_Total* is exponentially less than *Leung_Interval_Total*.

5. FEASIBILITY AND PRIORITY ASSIGNMENT

Optimal priority assignment requires a sufficient and necessary feasibility test. The previous section showed that such a test need only check deadlines of τ_i in $[S_i, S_i + P_i)$ to establish feasibility. This section provides a method for examining feasibility and shows how the combined optimal priority assignment and feasibility approach work in practice.

---

6. Leung's approach incorporates the construction of a schedule over an interval equal to twice the *lcm* of all task periods. In the worst case, when inserting any task into the schedule, each slot in that schedule has to be examined. Hence, each task requires a check of the entire interval.

5.1. Feasibility Testing

The primary result of section 4 was that feasibility of a task set can be determined by examining the executions of each task over its feasibility interval (defined by theorem 6). This could be achieved by the construction of a schedule over each feasibility interval. This requires that for $\tau_1,\tau_2..\tau_i$, we assign sufficient slots in a schedule for each task to meet its deadlines. However, since we need to know exactly how much outstanding computation needs to be honoured at S_i, the schedule has to be constructed for $[0, S_i + P_i)$. This is an increase of S_i over the exact feasibility interval required. S_i can have a maximum value given by:

$$S_i^{\max} = max_{1 \le j \le i}\{\tau_j\} + T_i - 1$$

Therefore, as n increases, the additional interval length becomes less intrusive. This approach is inefficient in that, in the worst case, the entire schedule has to be examined for each task.

We now introduce a more efficient approach. For the purposes of the following discussion, we assume that task offsets have been adjusted according to theorem 4. For a release of τ_i at t, we can state that it will meet its deadline if and only if the computational demands of higher priority tasks and C_i are no greater than D_i. That is:

$$I_i^t + C_i \le D_i \tag{15}$$

Since the feasibility of an individual task in a static priority pre-emptive scheduling scheme depends only upon itself and other tasks of higher priority, when determining the feasibility interval of $\tau_i \in \Delta^*$ we consider τ_i and those tasks in Δ^* of higher priority. Considering the optimal priority assignment method introduced in section 3, all priority levels $1..i-1$ are unassigned. Therefore, without loss of generality, we arbitrarily assign currently unassigned tasks to those levels. When determining feasibility, the specific assignments of levels $1..i-1$ is not important, only that all unassigned tasks have priority greater than τ_i.

The term I_i^t is defined as follows:

Definition 2:

The interference that is suffered by τ_i due to higher priority tasks wishing to execute during the release of τ_i starting at t is defined as I_i^t.

■

The interference I_i^t is made up of two parts: the execution demands of higher priority tasks that have been released before t and have deadlines after t; and the executions of higher priority tasks released in $[t, t + D_i)$. We now define these two terms.

Definition 3:

The *remaining interference* on a release of τ_i at time t, due to higher priority tasks that have not completed their execution at t, is defined as R_i^t.

■

Definition 4:

The *created interference* on a release of τ_i at time t, due to higher priority tasks released in the interval $[t, t + D_i)$, is defined as K_i^t.

■

Formally, we can state that the interference on τ_i during the release starting at t is

$$I_i^t = R_i^t + K_i^t$$

Hence, at each release of τ_i at t, if the following condition holds, τ_i is feasible for that release.

$$R_i^t + K_i^t + C_i \leq D_i$$

Formally, for the entire task set, the feasibility condition becomes:

$$\forall i : 1 \leq i \leq n \tag{16}$$

$$\forall t \in B_i : R_i^t + K_i^t + C_i \leq D_i$$

$$B_i = \left\{ t \mid t \in (S_i, S_i + T_i, S_i + 2T_i, ..., S_i + P_i) \right\}$$

Any test based upon the feasibility intervals defined by theorem 6 using the feasibility condition given by equation (16) will be sufficient and necessary if and only if the values of R_i^t and K_i^t are exact.

We now move to define R_i^t and K_i^t together with a sufficient and necessary feasibility test.

Exact Calculation of R_i^t

The easiest method for determining R_i^t is to construct and examine a schedule for the interval $[0, t)$, for each release of τ_i at t. This is clearly inefficient. A better solution would be to adopt Leung's approach [Leu80], and construct a schedule for the entire task set for the interval $[0, t_{\max})$, where $t_{\max}$ is defined by the maximum endpoint of the n feasibility intervals defined for the tasks in Δ^*. This again is inefficient.

Another approach can be derived by noting, when we consider the feasibility of τ_i, that its first release occurs at time 0 [7]. Therefore, since the interference due to higher priority tasks released at time 0 will be included in K_t^i, we have $R_i^0 = 0$.

Definition 5:

L_i^t represents the outstanding computation requirement by tasks $\tau_1..\tau_i$ at time t.
L_i^t is only defined for $t = 0, D_i, T_i + D_i, 2T_i + D_i \cdots$ [8]

■

Considering the second release of τ_i at T_i, we use $L_i^{D_i}$ as a basis for calculating $R_i^{T_i}$ (releases of τ_i occur at $0, T_i, 2T_i, ...$). Knowing at D_i that $L_i^{D_i}$ computation of $\tau_1..\tau_i$ is outstanding, we can step through the execution of tasks $\tau_1..\tau_{i-1}$ for $[D_i, T_i)$ noting the remaining computation at T_i. This forms the remaining interference value $R_i^{T_i}$. In general, we utilise $L_i^{(m-1)T_i + D_i}$ as the basis to determine $R_i^{mT_i}$, $m \in Z^+$: all releases of $\tau_1..\tau_{i-1}$ in $[(m-1)T_i + D_i, mT_i)$, together with $L_i^{(m-1)T_i + D_i}$, can contribute to $R_i^{mT_i}$.

A set of tuples β can be found, where each tuple (C_j, t) represents a demand by $\tau_j \in \{\tau_1, \tau_2, .., \tau_{i-1}\}$ for C_j units of computation time for a release of τ_j at any time $t \in [(m-1)T_i + D_i, mT_i)$. If $L_i^{(m-1)T_i + D_i} > 0$ we introduce an extra tuple $(L_i^{(m-1)T_i + D_i}, (m-1)T_i + D_i)$ into β to represent the outstanding computation at $(m-1)T_i + D_i$. The tuple set β is ordered in non-decreasing t values. R_i^t can now be determined by stepping through the

---

7. By rearrangement of task timing characteristics according to theorem (2) $\min(O_1..O_i) = O_i$ with $O_i = 0$.

8. We note that if τ_i has met its deadline at t then no part of L_i^t will be due to τ_i.

computation demands defined by the tuples in β. The following algorithm encapsulates this approach.

Algorithm 2: Exact Remaining Interference.

```
RemainingInterference ()
  begin
    time  =  t - T_i + D_i
    R_i^t  =  0
    -- create and order tuple set β
    -- for this release of τ_i
    for (C, t_r) in β
        if (t_r > time + R_i^t) then
            R_i^t = 0
        endif
        time = t_r
        R_i^t = R_i^t + C
    endfor
    R_i^t = R_i^t - (t - t_r)
    if (R_i^t < 0) then
        R_i^t = 0
    endif
  end
```

■

The result of the algorithm is that rem = R_i^t. We note that β is empty for R_i^0, hence on termination R_i^0 = rem = 0.

The complexity of the algorithm is due to the ordering of the tuple set. This can be achieved in O($N_i \log_2 N_i$) where N_i gives the cardinality of β for τ_i. In the worst case N_i is given by:

$$N_i = 1 + \sum_{j=1}^{i-1} \left\lceil \frac{T_i - D_i}{T_j} \right\rceil \qquad (17)$$

This approach is also sufficient and necessary in that any value of R_i^t for a release of τ_i at t ($1 \le i \le n$) is exact. The proof of this is trivial.

Exact Calculation of K_i^t

One approach to solve is to define a set of tuples η, in the same manner identified in section 5.1, with one tuple (C_j, t_j) per release of $\tau_j \in \{\tau_1..\tau_{i-1}\}$ at $t_j \in [t, t + D_i)$. Each tuple is used to step along the interval $[t, t + D_i)$ to calculate the demands of higher priority tasks. Allowance is made for the outstanding computation at t, namely R_i^t, by stepping through $[t + R_i^t, t + D_i)$. The tuple set is ordered by non-decreasing t. The following algorithm illustrates the approach by calculating K_i^t.

Algorithm 3: Exact Created Interference.

```
CreatedInterference ()
  begin
    next_free = R_i^t + t
    K_i^t = 0
    total_created = R_i^t
    -- create and order tuple set η
    -- for this release of τ_i
    for (C, t_r) in η
        total_created = total_created + C
        if (next_free < t_r) then
            next_free = t_r
        endif
        K_i^t = K_i^t + min (t + D_i - next_free, C)
        next_free = min (t + D_i, next_free + C)
    endfor
    L_i^(t+Ti) = total_created - create - max(D_i, R_i^t)
  end
```

■

Variable `next_free` is used to keep track of the next free slot in $[t, t + D_i)$. For tuple (C, t), this is always at least t as a task cannot execute before it is released. The initialisation of `next_release` indicates the first free slot occurs at $t + R_i^t$. Variable `create` tracks the part of the computation demanded by $\tau_1..\tau_{i-1}$ in $[t, t + D_i)$ which is actually met in the interval. Hence on termination, $K_i^t =$ `create`. Variable `total_created` contains the total computation demand during the interval. It is initialised to R_i^t to allow for the outstanding computation at t. The value of $L_i^{t+T_i}$, required for calculating $R_i^{t+T_i}$ can be formed from the termination value of `total_created` by subtracting `create` and $max(R_i^t, D_i)$. Note that if $R_i^t \geq D_i$ the release of τ_i at t cannot be feasible.

The complexity of the approach lies in ordering the tuples, i.e. $O(N_i \log_2 N_i)$ where N_i gives the number of tuples to be ordered. In the worst case, N_i is given by:

$$N_i = \sum_{j=1}^{i-1} \left\lceil \frac{D_i}{T_j} \right\rceil \tag{18}$$

The approach is sufficient and necessary in that the value of K_i^t on termination is exact. The proof lies in observing that only the computation demand that is honoured within $[t, t + D_i)$ contributes toward K_i^t.

Sufficient and Necessary Feasibility Test

The following algorithm defines a sufficient and necessary schedulability test. Its framework is based upon the feasibility condition, equation (16), and algorithms 2 and 3 to provide calculations for R_i^t and K_i^t respectively.

Algorithm 4: Sufficient and Necessary Feasibility.

```
FeasibilityTest ()
  begin
    for τi in Δ*    -- taken in order τ1, τ2,...
        t = 0;
        Lit = 0
        while (t < Si + Pi)

            -- Calculate Rit - create and order β
            RemainingInterference ()

            -- Calculate Kit - create and order η
            CreatedInterference ()

            if (Ci + Rit + Kit > Di)
                exit    -- τi not feasible so quit
            endif

            t = t + Ti   -- go to next release of τi
        endwhile
    endfor
  end
```

■

The algorithm assumes task offsets are rearranged according to theorem 4 for each loop iteration of τ_i.

The complexity of the algorithm is held in the number of releases of each task examined together with the complexity of determining R_i^t and K_i^t. In general, for τ_n, we examine $(P_n/T_n) - 1$ releases. The worst-case for R_i^t and K_i^t are given by equations (17) and (18) respectively. Hence, the complexity is given by:

$$O\left(\frac{P_n}{T_n}\left(\sum_{j=1}^{n-1}\left(\left\lceil\frac{T_n - D_n}{T_j}\right\rceil + \left\lceil\frac{D_n}{T_j}\right\rceil\right)\right)\right)$$

This approach is sufficient and necessary as the values calculated for R_i^t and K^{i^t} are exact for each iteration of every $\tau_i \in \Delta^*$.

Theorem 7:
The schedulability test defined by equation (16) using algorithms 3 and 2 for R_i^t and K_i^t respectively is sufficient and necessary.

Proof:

Consider $\tau_i \in \Delta^*$. By theorem 6, if each release of τ_i in $[S_i, S_i + P_i)$ meets its deadline, τ_i will always meet its deadline.

Consider the release of τ_i at t. In a static priority system, the only tasks that can prevent τ_i from meeting its deadline at $t + D_i$ are those higher priority tasks that need to execute in $[t, t + D_i)$. This is quantified as interference: $R_i^t + K_i^t$. Since these values are exact, the test is sufficient and necessary.

■

5.2. Combining Priority Assignment and Feasibility Testing

The optimal priority assignment approach outlined in section 3 finds tasks feasible for priority levels $n, n-1, ..., 1$, in that order. The overall complexity of the combined approach is bounded by n times the complexity for determining the feasibility of τ_n, that is:

$$O\left(n \frac{P_n}{T_n} \left(\sum_{j=1}^{n-1} \left(\left\lceil \frac{T_n - D_n}{T_j} \right\rceil + \left\lceil \frac{D_n}{T_j} \right\rceil \right) \right) \right)$$

Consider the following task set.

Example 4:

$\tau_A: C_A = 2 \quad D_A = 3 \quad O_A = 2 \quad T_A = 4$

$\tau_B: C_B = 3 \quad D_B = 4 \quad O_B = 0 \quad T_B = 8$

$\tau_C: C_C = 1 \quad D_C = 5 \quad O_C = 1 \quad T_C = 8$

■

Trivially the above is a non-critical instant task set. Firstly, we attempt to assign a task to priority level 3. Both τ_A and τ_B are infeasible at this level (for brevity we omit the calculation). Consider τ_C at level 3 (arbitrarily we assign τ_A to 1 and τ_B to 2). Task offsets are refined according to theorem 4: $O_1 = 1$, $O_2 = 7$ and $O_3 = 0$. For τ_3, $P_i = 8$ with $S_i = 8$. Therefore, the feasibility interval is [8, 16) implying we must check the deadline of the release of τ_3 at 8:

Release of τ_3 at 8:

Calculate R_3^8. Since $L_3^0 = 0$, tuple set $\beta = \{(2,2), (2,6), (3,7)\}$ Stepping through time, we derive $R_3^8 = 2$.

Calculate K_3^8. Tuple set $\eta = \{(2,10)\}$. Stepping through time, we derive $K_3^8 = 2$.

Giving $R_3^8 + K_3^8 + C_3 = 5 = D_3$

Hence, if a feasible priority assignments exist for the task set, (at least) one will assign τ_C to priority level 3. The process of assignment and feasibility testing continues for levels 2 and 1, although is omitted for brevity. Tasks τ_B and τ_A are assigned levels 1 and 2 respectively and are feasible. An extended example is given in the appendix.

6. CONCLUSIONS

This paper has considered and addressed several outstanding issues in static priority scheduling theory for task sets containing tasks with arbitrary start times. Presented in the paper have been efficient methods for

(i) determining if a set of tasks each with an arbitrary start time share a critical instant;

(ii) determining an optimal priority assignment;

(iii) determining feasibility.

Whilst no previous work is known to the authors regarding (i) and (ii), a comparison can be drawn between (iii) and a feasibility test proposed by Leung, based upon the construction of a schedule for a pre-determined interval. We have reduced Leung's interval, so making the feasibility problem easier, as well as offering a more efficient feasibility test for the reduced interval.

Whilst this paper presents a complete piece of work, further consideration of the feasibility test may yield more efficiency. The theory provided in this paper, provides a springboard for efficient solution of many problems that utilise task sets with arbitrary start times. Also, the theory in this paper is extensible. For example, it is the authors contention that the priority ceiling protocol theory [Sha90] can be incorporated trivially, thus permitting tasks to block on resource access. Overall, the theory presented in this paper provides a suitable vehicle for future research into the feasibility of even more generalised and flexible static priority task sets containing tasks with arbitrary timing constraints.

REFERENCES

Aud90. N. C. Audsley, ''Deadline Monotonic Scheduling'', YCS 146, Department of Computer Science, University of York (October 1990).

Aud91a. N. C. Audsley, A. Burns, M. F. Richardson and A. J. Wellings , ''STRESS: A Simulator For Hard Real-Time System'', RTRG/91/106, Real-Time Research Group, Department of Computer Science, University of York (October 1991).

Aud91b. 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).

Jac75. T. H. Jackson, *Number Theory*, Routledge and Kegan Paul (1975).

Jos86. 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).

Knu73. D. E. Knuth, *The Art of Computer Programming: Vol 1 Fundamental Algorithms*, Addison-Wesley (2nd Edition 1973).

Leh90. 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 Decmeber 1990).

Leh89. 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).

Leu89. J. Y. T. Leung, ''A New Algorithm for Scheduling Periodic, Real-Time Tasks'', *Algorithmica* **4**, pp. 209-219 (1989).

Leu80. J. Y. T. Leung and M. L. Merrill, ''A Note on Preemptive Scheduling of Periodic, Real-Time Tasks'', *Information Processing Letters* **11**(3) (November 1980).

Leu82. 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).

Liu73. 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).

Nas91. E. Nassor and G. Bres, ''Hard Real-Time Sporadic Task Scheduling for Fixed Priority Schedulers'', *Proceedings International Workshop on Responsive Systems*, Golfe-Juan, France, pp. 44-47, INRIA (Institut National de Recherche en Informatique et en Automatique) (3-4 October 1991).

Ros85. K. H. Rosen, *Elementary Number Theory and its Applications*, Addison-Wesley (1985).

Sha90. 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).

APPENDIX: Extended Example

Consider the task set, Δ^*, given in example 5.

Example 5:

$\tau_A : O_A = 4 ; C_A = 1 ; D_A = 1 ; T_A = 10$

$\tau_B : O_B = 5 ; C_B = 1 ; D_B = 2 ; T_B = 10$

$\tau_C : O_C = 0 ; C_C = 5 ; D_C = 6 ; T_C = 20$

$\tau_D : O_D = 7 ; C_D = 8 ; D_D = 9 ; T_D = 40$

$\tau_E : O_E = 27 ; C_E = 8 ; D_E = 14 ; T_E = 40$

$\tau_f : O_F = 0 ; C_F = 6 ; D_F = 30 ; T_F = 40$

■

We note that the tasks $\tau_A..\tau_F$ are arranged in both rate-monotonic and deadline-monotonic order, that is in order of increasing periods and deadlines. The utilisation of Δ^* is 100%.

Common Release Time

We determine if a common release time exists for the tasks in Δ^* using the method outlined in section 2.2. By inspection, τ_C and τ_F both have offset 0, hence share a common release time at 0. We form the hybrid task τ_{CF}. The *gcd* is calculated by Euclid's algorithm, returning $gcd(T_C, T_F) = 20$, with $x = 1$ and $y = 0$. The period T_{CF} is given by equation 4:

$$T_{CF} = T_C \frac{T_F}{gcd(T_C, T_F)} = \frac{20 \times 40}{20} = 40$$

We now evaluate O_{CF}. Evaluating k in equation (7), where $h = 0$, leaves $k = 0$. Evaluating t according to equation (9) leaves $t = 0$. The offset, O_{CF}, given by equation 10 is $O_{CF} = 0$,

coninciding with the first common release of τ_C and τ_F.

Next, we consider if the hybrid task, τ_{CF}, shares a common release time with τ_A. By equation 1 in section 2.1 we note that τ_{CF} and τ_A have a common release time if:

$$O_A - O_{CF} = h\ gcd(T_A, T_{CF}) \quad [h \in Z]$$

Since, by Euclid's algorithm, $gcd(T_A, T_{CF}) = 40$, $h = 20/7$ and so is not an integer. Thus τ_{CF} and τ_A, and therefore all the tasks in Δ^* do not share a common release time.

Now, we move to consider the priority assignment and feasbility of the task set, according to the method established in sections 3 and 5: each priority level from 6 to 1 is assigned to a task in Δ^*.

Priority Level 6

Let $\Psi(6) = \tau_F$ i.e. we wish to see if τ_F is feasible when assigned the lowest priority. Now we test the feasibility of τ_F at priority level 6. We have no need to refine offsets as O_F is the minimum offset of Δ^*. By theorem 6, the deadlines of τ_F need to be checked for releases occuring in $[S_F, S_F + P_F)$ where $P_F = 40$ and $S_F = \lceil 27/40 \rceil 40 = 40$. Hence the feasibility interval is [40, 80), requiring the examination of the release of τ_F at 40, assuming all other tasks have a higher priority:

Release of τ_F at 40:

Calculate R_6^{40}. Tuple set β is

$$\beta = \{(1,4), (1,14),(1,24), (1,34), (1,5), (1,15), (1,25), (1,35),$$
$$(5,0), (5,20), (8,7), (8,27)\}$$

Stepping through time, we derive $R_6^{40} = 0$ by algorithm 2.

Calculate K_6^{40}. Tuple set η is:

$$\eta = \{(1,44), (1,54), (1,64), (1,45), (1,55), (1,65), (5,40), (5,60), (8,47), (8,67)\}$$

Stepping through time, we derive $K_6^{40} = 27$ by algorithm 3.

Giving $R_6^{40} + K_6^{40} + C_F = 33 > D_F$. Hence τ_F is not feasible at priority level 6.

Let $\Psi(6) = \tau_E$. We test the feasibility of τ_E at priority level 6. According to theorem 4 we refine task offsets: $O_A = 7, O_B = 8, O_C = 13, O_D = 20, O_F = 13, O_E = 0$. The feasibility interval for τ_E is $[S_E, S_E + P_E)$ where $S_E = \lceil 27/40 \rceil 40 = 40$ and $P_E = 40$, giving [40, 80). We check the release of τ_E at 40 assuming all other tasks have higher priority:

Release of τ_E at 40:

Calculate R_6^{40}. Tuple set β is

$$\beta = \{(1,7), (1,17), (1,27), (1,37), (1,8), (1,18), (1,28), (1,38),$$
$$(5,13), (5,33), (8,20), (6,13)\}$$

Stepping through time, we derive $R_6^{40} = 3$.

Calculate K_6^{40}. Tuple set $\eta = \{(1,47), (1,48), (5,53), (6,53)\}$. Stepping through time, $K_6^{40} = 3$.

Giving $R_6^{40} + K_6^{40} + C_E = 14 = D_E$. Hence τ_E is feasible at priority level 6.

We move to assign priority level 5.

Priority Level 5

Let $\Psi(5) = \tau_F$. Now we test the feasibility of τ_F at priority level 5. We have no need to refine offsets as O_F is the minimum offset of Δ^*. By theorem 6, the deadlines of τ_F need to be checked for releases occuring in $[S_F, S_F + P_F)$ where $P_F = 40$ and $S_F = \lceil 27/40 \rceil 40 = 40$. Hence the feasibility interval is [40, 80), requiring the examination of the release of τ_F at 40 assuming τ_A, τ_B, τ_C, τ_D have higher priority:

Release of τ_F at 40:

Calculate R_5^{40}. Tuple set β is

$\beta = \{(1,4), (1,14), (1,24), (1,34), (1,5), (1,15), (1,25), (1,35), (5,0), (5,20), (8,7)\}$

Stepping through time, we derive $R_5^{40} = 0$.

Calculate K_5^{40}. Tuple set η is

$\eta = \{(1,44), (1,54), (1,64), (1,45), (1,55), (1,65), (5,40), (5,60), (8,47)\}$

Stepping through time, we derive $K_5^{40} = 24$.

Giving $R_5^{40} + K_5^{40} + C_F = 30 = D_F$. Hence τ_F is feasible at priority level 5.

We move to assign priority level 4.

Priority Level 4

Let $\Psi(4) = \tau_D$. We test the feasibility of τ_D at priority level 4. According to theorem 4 we refine task offsets: $O_A = 7, O_B = 8, O_C = 13, O_D = 0$. The feasibility interval for τ_D is $[S_D, S_D + P_D)$ where $S_D = \lceil 13/40 \rceil 40 = 40$ and $P_E = 40$, giving [40, 80). We check the release of τ_D at 40 assuming τ_A, τ_B and τ_C are of higher priority.

Release of τ_D at 40:

Calculate R_4^{40}. Tuple set β is

$\beta = \{(1,7), (1,17), (1,27), (1,37), (1,8), (1,18), (1,28), (1,38), (5,13), (5,33)\}$

Stepping through time, we derive $R_4^{40} = 0$.

Calculate K_4^{40}. Tuple set $\eta = \{(1,47), (1,48)\}$. Stepping through time, $K_4^{40} = 2$.

Giving $R_4^{40} + K_4^{40} = 10 > D_D$. Hence τ_D is not feasible at priority level 4.

Let $\Psi(4) = \tau_C$. We test the feasibility of τ_C at priority level 4. Since $O_C = 0$ we do not need to rearrange task offsets. The feasibility interval for τ_C is $[S_C, S_C + P_C)$ where $S_C = \lceil 14/20 \rceil 20 = 20$ and $P_E = 40$, giving [20, 60). We check the releases of τ_C at 20 and 40, assuming τ_A, τ_B and τ_D are of higher priority.

Release of τ_C at 20:

Calculate R_4^{20}. Tuple set $\beta = \{(1,4), (1,14), (1,5), (1,15), (8,7)\}$. Stepping through time, we derive $R_4^{20} = 0$.

Calculate K_4^{20}. Tuple set $\eta = \{(1,25), (1,24)\}$ with $K_4^{20} = 2$.

Giving $R_4^{20} + K_4^{20} + C_C = 7 > D_C$. Hence τ_C is not feasible at priority level 4.

Let $\Psi(4) = \tau_B$. We test the feasibility of τ_B at priority level 4. We refine offsets according to theorem 4: $O_A = 9, O_C = 15, O_D = 2, O_B = 0$. The feasibility interval for τ_B is $[S_B, S_B + P_B)$ where $S_B = \lceil 15/10 \rceil 10 = 20$ and $P_B = 40$, giving [20, 60). We check the releases of τ_B at 20, 30, 40 and 50 assuming τ_A, τ_C and τ_D are of higher priority.

Release of τ_B at 20:

Calculate R_4^{20}. Tuple set $\beta = \{(1,9), (1,19), (5,15), (8,2)\}$. Stepping through time, we derive $R_4^{20} = 1$.

Calculate K_4^{20}. Since tuple set $\eta = \{\}$, $K_4^{20} = 0$.

Giving $R_4^{20} + K_4^{20} + C_B = 2 = D_B$. Hence τ_B is feasible at priority level 4 for the release at 20.

Release of τ_B at 30:

Calculate R_4^{30}. We note that $L_4^{21} = 0$, that is the remaining workload of τ_A, τ_C and τ_D at the deadline of the release of τ_B at 20 was 0. Tuple set $\beta = \{(1,29)\}$. Stepping through time, we derive $R_4^{30} = 0$.

Calculate K_4^{30}. Since tuple set $\eta = \{\}$, $K_4^{30} = 0$.

Giving $R_4^{30} + K_4^{30} + C_B = 1 < D_B$. Hence τ_B is feasible at priority level 4 for the release at 30.

Release of τ_B at 40:

Calculate R_4^{40}. We note that $L_4^{31} = 0$, that is the remaining workload of τ_A, τ_C and τ_D at the deadline of the release of τ_B at 30 was 0. Tuple set $\beta = \{(1,39), (5,35)\}$. Stepping through time, $R_4^{40} = 1$.

Calculate K_4^{40}. Since tuple set $\eta = \{\}$, $K_4^{40} = 0$.

Giving $R_4^{40} + K_4^{40} + C_B = 2 = D_B$. Hence τ_B is feasible at priority level 4 for the release at 40.

Release of τ_B at 50:

Calculate R_4^{50}. We note that $L_4^{41} = 0$, that is the remaining workload of τ_A, τ_C and τ_D at the deadline of the release of τ_B at 40 was 0. Tuple set $\beta = \{(8,42), (1,49)\}$. Stepping through time, $R_4^{40} = 1$.

Calculate K_4^{50}. Since tuple set $\eta = \{\}$, $K_4^{50} = 0$.

Giving $R_4^{50} + K_4^{50} + C_B = 2 = D_B$. Hence τ_B is feasible at priority level 4 for the release at 50. Therefore, τ_B is feasible for releases at 20, 30, 40 and 50 and so is feasible at priority level 4.

We move to assign priority level 3.

Priority Level 3

Let $\Psi(3) = \tau_D$. We test the feasibility of τ_D at priority level 3. We refine offsets according to theorem 4: $O_A = 7, O_C = 13, O_D = 0$. The feasibility interval for τ_D is $[S_D, S_D + P_D)$ where $S_D = \lceil 13/40 \rceil 40 = 40$ and $P_B = 40$, giving [40, 80). We check the release of τ_D at 40, assuming τ_A and τ_C are of higher priority.

Release of τ_D at 40:

Calculate R_4^{40}. Tuple set $\beta = \{(1,7), (1,17), (1,27), (1,37), (6,13), (6,33)\}$. Stepping through time, we derive $R_4^{40} = 0$.

Calculate K_4^{40}. Tuple set $\eta = \{(1,47)\}$. Stepping through time, $K_4^{40} = 1$.

Giving $R_4^{40} + K_4^{40} + C_D = 9 = D_D$. Hence τ_D is feasible at priority level 3.

We move to priority level 2.

Priority Level 2

Let $\Psi(2) = \tau_C$. We test the feasibility of τ_C at priority level 2. There is no need to refine deadlines according to theorem 4 as $O_C = 0$. The feasibility interval for τ_C is $[S_C, S_C + P_C)$ where $S_C = \lceil 14/20 \rceil 20 = 20$ and $P_C = 20$, giving [20, 40). We check the release of τ_C at 20, assuming τ_A is of higher priority.

Release of τ_C at 20:

Calculate R_4^{20}. Tuple set $\beta = \{(1,4), (1,14)\}$. Stepping through time, we derive $R_4^{20} = 0$.

Calculate K_4^{20}. Tuple set $\eta = \{(1,24)\}$. Stepping through time, we derive $K_4^{20} = 1$.

Giving $R_4^{40} + K_4^{40} + C_C = 6 = D_C$. Hence τ_D is feasible at priority level 2.

We move to priority level 1.

Priority Level 1

One unassigned task remains, hence let $\Psi(1) = \tau_A$. Refining offsets, $O_A = 0$. The feasiblity interval is $[S_A, S_A + P_A)$ where $S_A = \lceil 0/10 \rceil 10 = 0$ and $P_A = T_A = 10$, giving [0, 10). We check the release os τ_A at 0 only. Trivially, since no higher priority tasks exist, $R_1^0 = K_1^0 = 0$, giving $C_A = 1 = D_A$. Thus τ_A is feassible at priority level 1.

Summary

The task set Δ^* is feasible with $\Psi(1) = \tau_A$, $\Psi(2) = \tau_C$, $\Psi(3) = \tau_D$, $\Psi(4) = \tau_B$, $\Psi(5) = \tau_F$ and $\Psi(6) = \tau_E$. Figure 1 shows a simulation of task set Δ^* illustrating that no deadlines are missed when the above priority assignment is used. In contrast, Figure 2 shows the result of using deadline-montonic priority ordering: tasks miss deadlines. The figures are produced using the *STRESS* real-time simulator [Aud91a]. In both figures, time increases horizontally to the right, with individual dashed timelines shown for each task. Tasks have solid horizontal times whilst preempted. Task releases are given by a circle on the timeline; execution by a hatched box; task completion by a raised circle. Deadlines are indicated by a vertical solid line with an arrow head on the timeline. Missed deadlines are shown by a raised solid bullet.

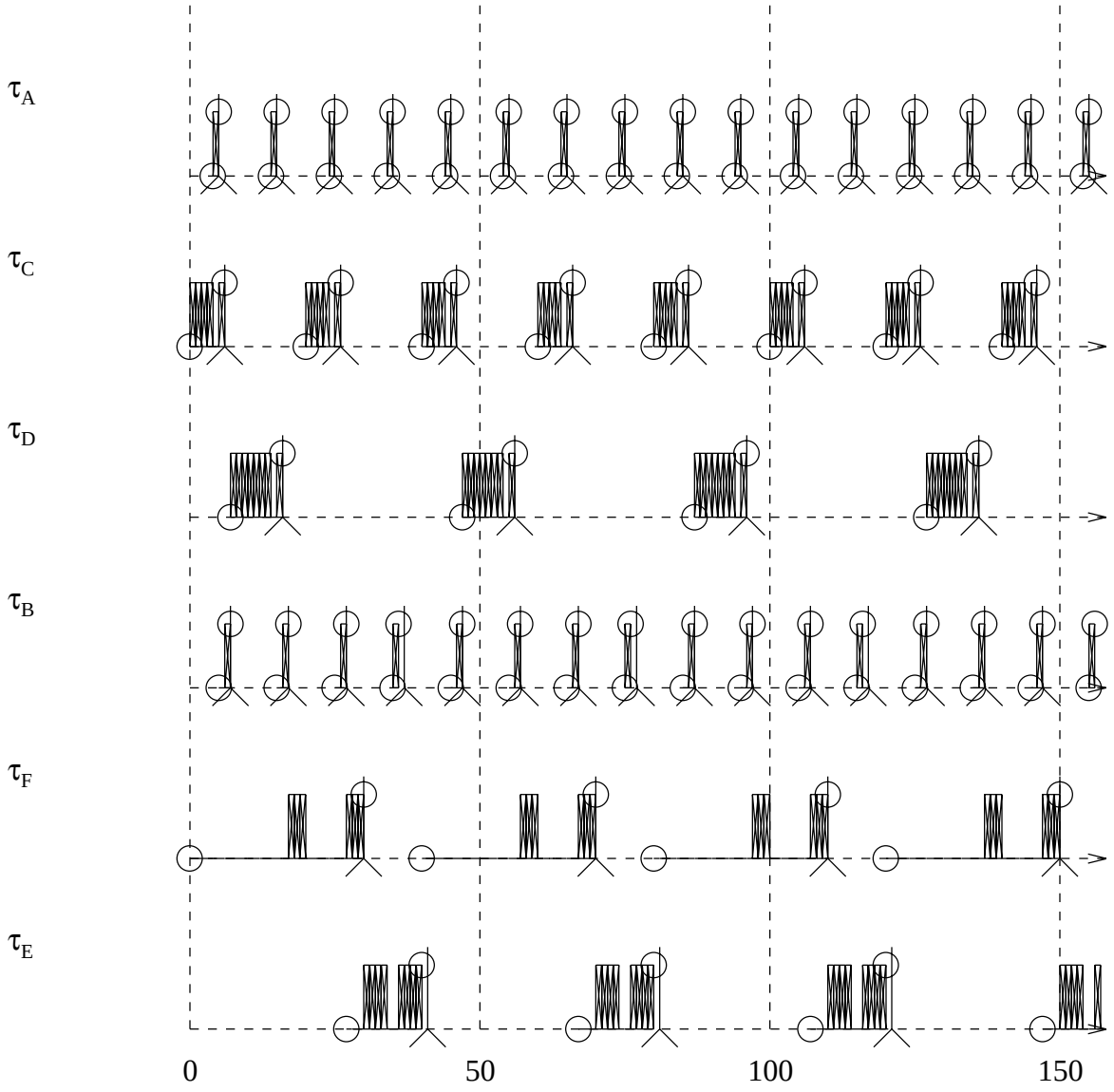

Figure 1: Task Set Δ^* with Optimally Assigned Priorities.

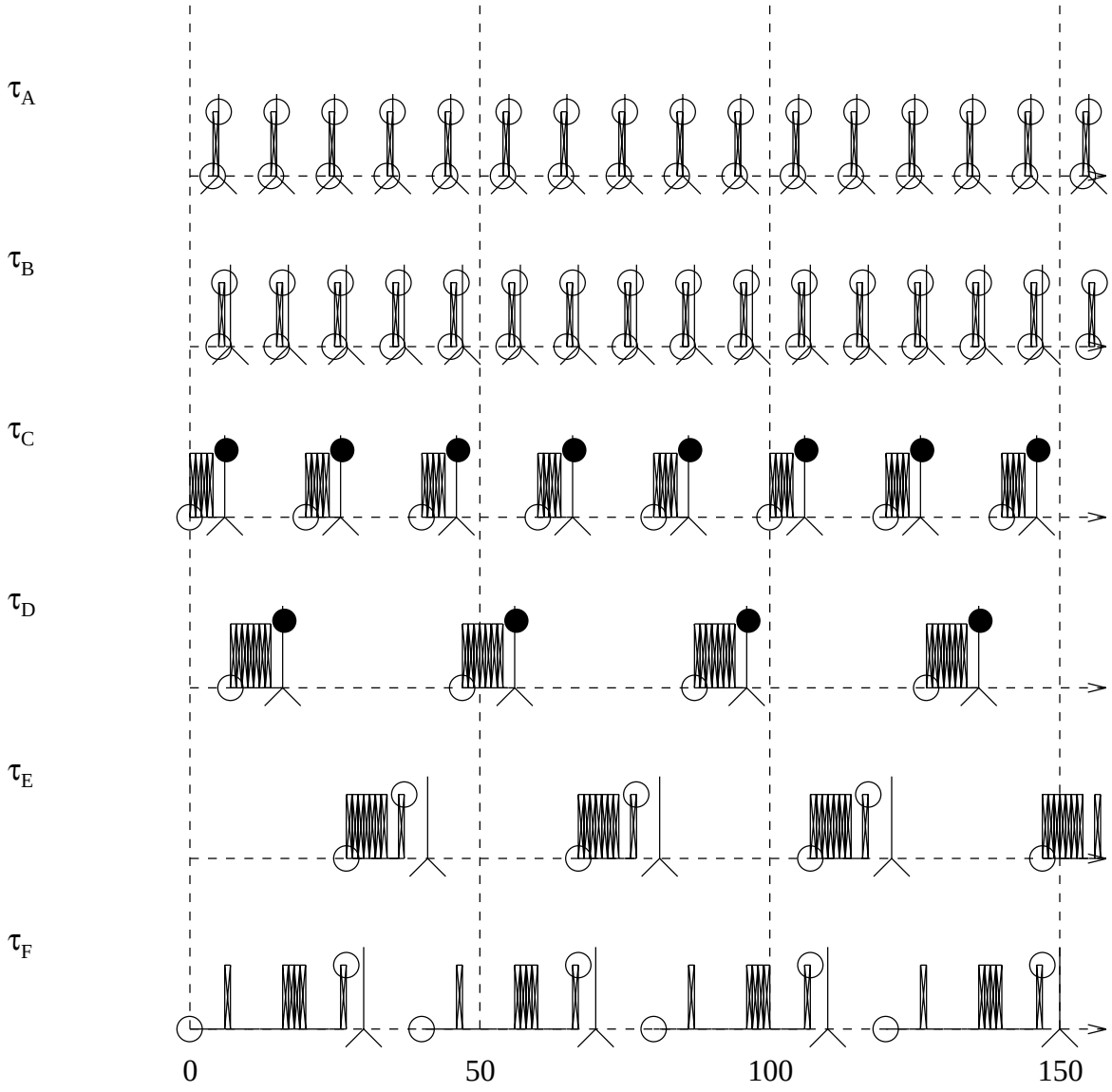

Figure 2: Task Set Δ^* with Deadline-Monotonic Assigned Priorities.