Accounting for Interrupt Handling Costs in Dynamic Priority Task Systems

Kevin Jeffay[*] *Donald L. Stone*[†]

University of North Carolina at Chapel Hill
Department of Computer Science
Chapel Hill, NC 27599-3175
{jeffay,stone}@cs.unc.edu

Abstract: In order to apply the results of formal studies of real-time task models, a practitioner must account for the effects of phenomena present in the implementation but not present in the formal model. We study the feasibility and schedulability problems for periodic tasks that must compete for the processor with interrupt handlers — tasks that are assumed to always have priority over application tasks. The emphasis in the analysis is on deadline driven scheduling methods. We develop conditions that solve the feasibility and schedulability problems and demonstrate that our solutions are computationally feasible. Lastly, we compare our analysis with others developed for static priority task systems.

1. Introduction

The problem of scheduling regularly occurring tasks to meet deadlines is central to the design and analysis of real-time systems. The scheduling analysis of real-time systems concerns two fundamental problems: *feasibility* and *schedulability*. For a given set of task execution requirements, the feasibility problem is that of determining relations on model parameters that are necessary and sufficient for ensuring there exists a schedule in which no task ever misses a deadline. An example execution requirement is that an invocation of a task occur no sooner than the deadline of the previous invocation. For a given set of scheduling constraints, the schedulability problem is that of determining if there exists a scheduling algorithm that is capable of scheduling any feasible task set such that no task indeed ever misses a deadline. An example scheduling constraint is that a scheduler never idle the processor when there exist uncompleted task invocations.

Research on the feasibility problem has given relatively little attention to the representation of system implementation overhead in workload models. In trying to apply existing feasibility analyses to actual task systems, the effects of such phenomena as context switching costs, interrupt handling, DMA, *etc*., must be taken into consideration. For example, the effect on CPU scheduling of memory bus cycle stealing due to DMA has been reported in [9]. The effect of operating system overhead and non-preemptable operations is considered in [6].

Research on the schedulability problem has focused on two scheduling paradigms: scheduling based on a static assignment of priorities to tasks, of which the *rate monotonic* priority assignment is the dominant approach, and scheduling based on a dynamically changing assignment of priority to tasks, of which the *deadline driven* priority assignment dominates. Under a rate monotonic priority assignment, tasks that are requested frequently have higher priority than tasks requested less frequently. Under a deadline driven priority assignment, tasks with nearer deadlines have priority over tasks with later deadlines. For particular task models, the rate monotonic priority assignment is known to be an optimal static priority assignment and the deadline driven priority assignment is known to be an optimal dynamic priority assignment [8].

In this paper we consider the problem of accounting for the effects of interrupt handling. We present an analysis of the feasibility and schedulability problem for a real-time workload that explicitly includes interrupt handlers. The emphasis is primarily on scheduling algorithms that dynamically vary the priority of tasks. Our interest in dynamic priority scheduling arises from the fact that dynamic priority algorithms are capable of guaranteeing a correct execution to any task set that is schedulable using a static priority assignment, while the reverse is not true. Moreover, as we discuss in Section 4, it is possible to model the effects of interrupt handlers in static priority systems using results from the literature (as well as those presented below). Lastly, in related work, we are applying results for dynamic priority scheduling to the design and implementation of a real-time multimedia system [4].

Throughout, we assume a task model in which interrupt handlers always have a higher priority than real-time application tasks. Our model is based on the hybrid

[*] Supported by a grant from the National Science Foundation (number CCR-9110938).
[†] Supported by a graduate fellowship from the IBM Corporation.

In: Proceedings of the 14th IEEE Real-Time Systems Symposium, Raleigh-Durham, NC, December 1993, pp. 212-221.

static/dynamic priority scheduling model presented in [8]. We develop quantitative conditions that solve the feasibility and schedulability problems and demonstrate that our solutions are computationally feasible. Our results will aid practitioners who wish to employ deadline driven scheduling techniques either by using our model directly or by using our framework to analyze similar models.

The following section describes our system model. We begin with task systems consisting of periodic application tasks and periodic interrupt handlers. Section 3 solves the feasibility problem for this model by deriving a recurrence relation that quantifies the amount of processor time lost to interrupt handlers in an arbitrary interval and solves the schedulability problem by showing that a scheduling algorithm that uses the deadline driven priority assignment is optimal. Section 3 also discusses extensions of our analyses to other dynamic priority task systems. Section 4 compares our schedulability analysis to that for static priority systems. We conclude with an outline of future extensions and applications of this work.

2. System Model

We consider a real-time system to comprise two distinct classes of tasks: *application tasks* and *interrupt handlers*. Tasks of either type are sequential programs that are repeatedly *invoked* by occurrences of *events*. Typically, interrupt handlers execute in response to externally generated events (*e.g.*, device interrupts). Application tasks execute in response to internally generated events (*e.g.*, the arrival of a message). Events are *periodic*. A periodic event occurs every p time units for some constant p.

Formally, an *application task* T is a pair (c, p) where c is the maximum amount of processor time required to execute (the sequential program of) task T to completion on a dedicated uniprocessor, and p is the interval between occurrences of the event v that leads to invocations of T. Event v (and thus invocations of T) occurs at time kp, for all $k \geq 0$. The i^{th} execution of task T terminates no later than the *deadline* of $(i+1)p$. This will require that c units of processor time be allocated to the execution of T in the (closed) interval $[ip, (i+1)p]$. If this does not occur then task T is said to have missed a deadline at time $(i+1)p$.

An *interrupt handler* I is a pair (e, a) where e is the maximum amount of processor time required to execute interrupt handler I to completion on a dedicated uniprocessor, and a is the interval between occurrences of the event v' that leads to invocations of I. Event v' (and thus invocations of I) occurs at time ka, for all $k \geq 0$.

We assume time is discrete and clock ticks are indexed by the natural numbers. Interrupts and task invocations occur at clock ticks; parameters c, p, a, and e are expressed as integer multiples of the interval between successive clock ticks. We further assume application tasks and interrupt handlers are independent in the sense that the time at which an interrupt handler or application task is invoked is unrelated to any invocation of any other task or handler other than previous invocation of the same program.

Our execution model is priority-driven, preemptive execution. The execution rules for interrupt handlers and application tasks are as follows. Interrupt handlers execute with priority strictly greater than application tasks. Thus, if at any time t, $t \geq 0$, there exists an invocation of an interrupt handler that has not completed execution, then in the interval $[t, t+1]$, an interrupt handler must execute. Interrupt handlers therefore may preempt application tasks at arbitrary points.

We define a task system τ as a set of m interrupt handlers $(e_1, a_1) \ldots (e_m, a_m)$, and n application tasks, $(c_1, p_1) \ldots (c_n, p_n)$. A task system is *feasible* if it is possible to schedule the application tasks such that each invocation of each application task completes execution at or before its deadline. We posit no correctness conditions for the interrupt handlers (*e.g.*, response time properties). Our emphasis is on understanding the impact of interrupt handlers on application tasks. Because interrupt handlers execute with priority strictly higher than that of all application tasks, they are unaffected by application tasks and can thus be analyzed in isolation using methods previously reported in the literature (*e.g.*, [2]).

3. Feasibility and Schedulability of Dynamic Priority Task Systems

The analysis of sets of tasks scheduled by dynamic priority algorithms has recently been analyzed within the context of *processor demand* [3, 5]. For a set of periodic tasks τ, and an interval of length l, $l \geq 0$, the processor demand is defined as the least upper bound on the amount of processing time required by τ in the interval $[t, t+l]$ to complete execution of all possible invocations of tasks with deadlines in the interval $[t, t+l]$. If τ is schedulable then for all non-negative t and l, the processor demand in $[t, t+l]$, must be less than or equal to l when τ is scheduled using an optimal algorithm. In general, the processor demand in the interval $[t, t+l]$ will be a function of the processor time required to complete execution of the most recent invocation (*i.e.*, before time t) of each task, the costs and periods of the tasks, and the length of the interval.

Preemptive Task Systems

Here we analyze the feasibility and schedulability problems when application tasks are allowed to preempt one another at arbitrary points.

Our analysis is an alternative to Liu and Layland's original analysis of preemptive periodic tasks [8]. They have shown that if tasks are allowed to preempt one another at arbitrary points, then a task system consisting of just a set of n application tasks will be feasible if and only if

$$\sum_{i=1}^{n} \frac{c_i}{p_i} \le 1 \qquad (1)$$

If a periodic task T_i has a cost c_i and period p_i, then c_i/p_i is the fraction of processor time consumed by T_i over the lifetime of the system, *i.e.*, the utilization of the processor by T_i. This result expresses feasibility as a function of the cumulative processor utilization. The necessity of (1) follows from the fact that for a set of tasks to be feasible on a single processor system, the tasks cannot overload the processor. The sufficiency of (1) is demonstrated by showing that if (1) holds then the *deadline driven* scheduling algorithm, an algorithm that assigns higher priority to task invocations with nearer deadlines, will schedule the tasks without any task ever missing a deadline.

To illustrate the use of processor demand, the following theorem gives an equivalent feasibility condition.

Theorem 3.1: A set of periodic tasks will be feasible if and only if for all L, $L \ge 0$,

$$L \ge \sum_{i=1}^{n} \left\lfloor \frac{L}{p_i} \right\rfloor c_i . \qquad (2)$$

Proof: It is straightforward to demonstrate that (1) ⇔ (2) (see Appendix). □

The right-hand side of inequality (2) is the processor demand in the interval $[0, L]$ when all tasks are invoked at time $t = 0$. In the interval $[0, L]$ there are exactly $\lfloor L/p_i \rfloor$ invocations of task T_i that must complete execution at or before time L. As each invocation requires c_i units of processor time, the processor demand due to task T_i in $[0, L]$ is thus $\lfloor L/p_i \rfloor c_i$. Theorem 3.1 requires that for all L, the cumulative processor demand in every interval $[0, L]$ be less than the amount of processor time available for executing tasks in the interval. Since we assume a single processor system, the amount of available processor time is simply L, the length of the interval.

Theorem 3.1 can be extended to include the effects of interrupt handlers on application tasks. Since interrupt handlers execute with priorities strictly greater than those of application tasks, their effect is to reduce the amount of time available for processing application tasks. To quantify the time spent executing interrupt handlers, consider a set of m interrupt handlers $(e_1, a_1) \ldots (e_m, a_m)$. Let $f(l)$ be a function from the non-negative integers to the non-negative integers defined by the following recurrence relation:

$$f(0) = 0,$$

$$\forall l > 0,\ f(l) = \begin{cases} f(l-1) & \text{if } f(l-1) = \sum_{i=1}^{m} \left\lceil \frac{l}{a_i} \right\rceil e_i \\ f(l-1)+1 & \text{if } f(l-1) < \sum_{i=1}^{m} \left\lceil \frac{l}{a_i} \right\rceil e_i \end{cases}$$

Note that for all l, $l \ge 0$, $f(l) \le \text{MIN}(l,\ \sum_{i=1}^{m} \lceil l/a_i \rceil e_i)$.

We show $f(l)$ is the least upper bound on the amount of time spent executing interrupt handlers in any interval of length l. First, we show that for all l, there exist intervals in which the amount of processor time consumed by interrupt handlers is exactly $f(l)$. This shows that the least upper bound on the time spent executing interrupt handlers in an interval of length l must be at least $f(l)$.

Lemma 3.2: Let τ be a task system with m interrupt handlers $(e_1, a_1) \ldots (e_m, a_m)$. For all l, $l \ge 0$, $f(l)$ is the amount of processor time consumed by interrupt handlers in the interval $[0, l]$.

Proof: By induction on l.

Basis: $l = 0$. Since time is assumed to be discrete, no execution occurs in an interval of length 0 and hence $f(0) = 0$.

Induction Step: Assume $f(k–1)$ is the processor time consumed by interrupt handlers in the interval $[0, k–1]$ for some k, $k \ge 1$. We show the theorem must hold for $l = k$.

For all $a \ge 0$, $b \ge 0$, let $g(a,b)$ be the amount of processor time consumed by interrupt handlers in the interval $[a, b]$. In the interval $[0, k]$, the only interrupt handlers that execute are those invoked in the (closed) interval $[0, k–1]$. There are at most $\sum_{i=1}^{m} \lceil k/a_i \rceil$ such invocations and hence $g(0,k) \le \sum_{i=1}^{m} \lceil k/a_i \rceil e_i$. If $f(k–1) < \sum_{i=1}^{m} \lceil k/a_i \rceil e_i$ then there necessarily exists an invocation of an interrupt handler that has not completed execution at time $k–1$ and hence one will execute in the interval $[k–1, k]$. In this case $g(0,k) = f(k–1) + 1$. If $f(k–1) = \sum_{i=1}^{m} \lceil k/a_i \rceil e_i$ then all interrupt handlers invoked in the (closed) interval $[0, k–1]$ have completed execution at or prior to time $k–1$. Moreover, no interrupt occurred at time $k–1$. In this case $g(0,k) = f(k–1)$. In either case $g(0,k) = f(k)$. This proves the lemma. □

The following lemma shows that $f(l)$ is an upper bound on the time spent executing interrupt handlers in an interval of length l.

Lemma 3.3: Let τ be a task system with m interrupt handlers $(e_1, a_1) \ldots (e_m, a_m)$. For all t and l, $t \ge 0$, $l \ge 0$, $f(l)$ is an upper bound on the amount of processor time spent executing interrupt handlers in the interval $[t, t+l]$.

Proof: By contradiction.

Let $g(a,b)$ be defined as in the proof of Lemma 3.2. Suppose $f(l)$ is not an upper bound on $g(t, t+l)$ for all $t \geq 0$ and $l \geq 0$. Then there exists some x and k such that

$$f(k) < g(x, x+k). \quad (3)$$

Choose the smallest x for which (3) holds, and for that x, choose the smallest k. For this choice of x and k

$$g(x-1, x+k-1) \leq f(k) < g(x, x+k) \quad (4)$$

Since the two intervals $[x-1, x+k-1]$ and $[x, x+k]$ overlap except at the endpoints, (4) holds only if the processor is executing interrupt handlers in the unit interval $[x+k-1, x+k]$ and not in the unit interval $[x-1, x]$. Hence

$$g(x, x+k) = g(x, x+k-1) + 1$$

It also follows from the choice of x and k that

$$g(x, x+k-1) \leq f(k-1)$$

Hence

$$g(x, x+k) \leq f(k-1) + 1$$
$$f(k) < f(k-1) + 1$$

Since for all l, $f(l-1) \leq f(l)$, it follows that $f(k) = f(k-1)$ and thus by the definition of $f(l)$, $f(k) = \sum_{i=1}^{m} \lceil k/a_i \rceil e_i$.

Since interrupt handlers do not execute in the interval $[x-1, x]$, the only interrupt handlers that execute in the interval $[x, x+k]$ are those actually invoked in the interval $[x, x+k-1]$. Therefore, the maximum amount of processor time that can be spent executing these interrupt handlers is $\sum_{i=1}^{m} \lceil k/a_i \rceil e_i$, and thus

$$g(x, x+k) \leq \sum_{i=1}^{m} \left\lceil \frac{k}{a_i} \right\rceil e_i = f(k).$$

However this contradicts inequality (3). Therefore, there cannot exist an interval of length k in which the amount of processor time spent executing interrupt handlers is greater than $f(k)$. □

Theorem 3.4: Let τ be a task system with m interrupt handlers $(e_1, a_1) \ldots (e_m, a_m)$. For all t and l, $t > 0$, $l \geq 0$, $f(l)$ is a least upper bound on the amount of processor time spent executing interrupt handlers in the interval $[t, t+l]$.

Proof: Lemma 3.3 has demonstrated that $f(l)$ is an upper bound on the amount of processor time spent executing interrupt handlers in the interval $[t, t+l]$. Lemma 3.2 has demonstrated that the amount of processor time spent executing interrupt handlers in the interval $[0, l]$ is $f(l)$. Therefore no upper bound can be smaller than $f(l)$. It follows that $f(l)$ is a least upper bound. □

Note that results of the previous theorem and lemma are independent of how the processor is allocated amongst the interrupt handlers.

The following theorem gives necessary and sufficient conditions for the feasibility of a set of application tasks in the presence of interrupt handlers.

Theorem 3.5: Let τ be a task system with n application tasks $(c_1, p_1) \ldots (c_n, p_n)$ and m interrupt handlers $(e_1, a_1) \ldots (e_m, a_m)$. τ will be feasible if and only if for all $L \geq 0$,

$$L - f(L) \geq \sum_{i=1}^{n} \left\lfloor \frac{L}{p_i} \right\rfloor c_i. \quad (5)$$

Proof: ($\Rightarrow$) For all t, $t > 0$, in the interval $[0, t]$, there are t time units available in which work can be done. By Lemma 3.2, $f(t)$ of these time units are spent processing interrupts leaving $t - f(t)$ time units for processing application tasks. Each application task will require $\lfloor t/p_i \rfloor c_i$ units of processor time in the interval $[0, t]$ to ensure the task does not miss a deadline in the interval $[0, t]$. Therefore τ can be scheduled only if

$$t - f(t) \geq \sum_{i=1}^{n} \left\lfloor \frac{t}{p_i} \right\rfloor c_i.$$

($\Leftarrow$) To show the sufficiency of (5) we show that a if task system τ satisfies (5) for all L, $L > 0$, then a deadline driven scheduler will succeed in scheduling τ. This is shown by contradiction.

Assume for all L, $L > 0$, τ satisfies (5) but yet an application task in τ misses a deadline when scheduled according to a deadline driven algorithm. Let t_d be the earliest time at which a deadline is missed and let t be the later of:

- the end of the last interval prior to t_d in which the processor has been idle (or 0 if the processor has never been idle), or,
- the latest time prior to t_d at which an invocation of an application task with deadline after time t_d executes (or time 0 if such an invocation does not execute prior to t_d).

By choice of t, no invocation of an application task with deadline after t_d executes in the interval $[t, t_d]$. If scheduling is deadline driven then the processor demand in the interval $[t, t_d]$, is at most $\sum_{i=1}^{n} \lfloor (t_d - t)/p_i \rfloor c_i$. Moreover, at most $f(t_d - t)$ time units are spent executing interrupt handlers in the interval $[t, t_d]$ and hence at least $(t_d - t) - f(t_d - t)$ time units are available for executing application tasks. Since a deadline is missed at time t_d it follows that

$$\sum_{i=1}^{n} \left\lfloor \frac{t_d - t}{p_i} \right\rfloor c_i > (t_d - t) - f(t_d - t).$$

However this contradicts our assumption that τ satisfies (5) for all L. Hence if τ satisfies (5) then a deadline driven scheduler will succeed in scheduling τ. It follows that

satisfying (5) for all L, $L > 0$, is a sufficient condition for feasibility. □

The proof of Theorem 3.5 also establishes the optimality of the deadline driven scheduling algorithm for scheduling application task sets in the presence of interrupt handlers, as the condition that is necessary for feasibility guarantees the correctness of the algorithm. Moreover, note that Theorem 3.1 is a corollary of Theorem 3.5. The task system in Theorem 3.1 is the special case of that considered in Theorem 3.5 obtained when there are no interrupt handlers (*i.e.*, $m = 0$).

While Theorem 3.5 gives necessary and sufficient conditions for feasibility, that condition (5) be evaluated for all $L \geq 0$ implies that Theorem 3.5 cannot be used directly as a basis of test for feasibility. Unfortunately, there is no equivalent formulation of (5) (that we know of) in terms of a more easily computed function such as processor utilization (see Section 5). However, by restricting the set of task systems, the feasibility condition of Theorem 3.5 can be reduced to one that can be efficiently evaluated.

Define the achievable processor utilization as:

$$U = \sum_{i=1}^{n} \frac{c_i}{p_i} + \sum_{i=1}^{m} \frac{e_i}{a_i}.$$

In the remainder of this section we restrict ourselves to task systems that do not fully utilize the processor (*i.e.*, ones for which $U < 1$).

Theorem 3.6: Let τ be a task system as in Theorem 3.5 with $U < 1$. Let

$$B = \frac{\sum_{i=1}^{m} e_i}{1-U}$$

and let $\boldsymbol{P} = \{kp_i \mid kp_i < B \wedge k \geq 0 \wedge 1 \leq i \leq n\}$ be the set of non-negative multiples, less than B, of the periods of the application tasks. τ will be feasible if and only if for all L, $L \in \boldsymbol{P}$:

$$L - f(L) \geq \sum_{i=1}^{n} \left\lfloor \frac{L}{p_i} \right\rfloor c_i \qquad (6)$$

Proof: The necessity of (6) for all L, $L \in \boldsymbol{P}$, is a direct consequence of Theorem 3.5. We demonstrate the sufficiency of (6) in two parts. First, we show that (6) need only hold at multiples of the periods of the application tasks in order for the tasks to be feasible. Next, we show that it suffices to consider only values of L less than B. Our proof of the former fact is taken from the proof of Theorem 9 in [8].

Let $\boldsymbol{Q} = \{kp_i \mid k \geq 0 \wedge 1 \leq i \leq n\}$. Choose t, $t' \in \boldsymbol{Q}$, $t < t'$, such that there does not exist an $r \in \boldsymbol{Q}$, $t < r < t'$. Let ε be an integer such that $0 \leq \varepsilon < t' - t$. It follows that for all i, $1 \leq i \leq n$, $\lfloor t/p_i \rfloor = \lfloor (t+\varepsilon)/p_i \rfloor$. Moreover, since at most ε time units can be spent executing interrupt handlers in the interval $[t, t+\varepsilon]$, $f(t+\varepsilon) \leq f(t)+\varepsilon$. If (6) is satisfied at t then

$$t - f(t) \geq \sum_{i=1}^{n} \left\lfloor \frac{t}{p_i} \right\rfloor c_i$$

$$t - f(t) \geq \sum_{i=1}^{n} \left\lfloor \frac{t+\varepsilon}{p_i} \right\rfloor c_i$$

$$t + \varepsilon - \left(f(t) + \varepsilon\right) \geq \sum_{i=1}^{n} \left\lfloor \frac{t+\varepsilon}{p_i} \right\rfloor c_i$$

$$t + \varepsilon - f(t+\varepsilon) \geq \sum_{i=1}^{n} \left\lfloor \frac{t+\varepsilon}{p_i} \right\rfloor c_i$$

Therefore, if (6) holds at multiples of the periods of application tasks (and 0), it also holds at all values in between. It thus suffices to only consider positive multiples of the p_i. We next show that it further suffices to consider only points less than B. Consider the function

$$g(L) = \sum_{i=1}^{n} \left\lfloor \frac{L}{p_i} \right\rfloor c_i + f(L) - L.$$

By Theorem 3.5, τ will be feasible if and only if $g(L) \leq 0$ for all $L \geq 0$. $g(L)$ is bounded from above by the function $h(L) = L(U-1) + \sum_{j=1}^{m} e_j$ since

$$\begin{aligned} g(L) &\leq \sum_{i=1}^{n} \left\lfloor \frac{L}{p_i} \right\rfloor c_i + \sum_{j=1}^{m} \left\lceil \frac{L}{a_j} \right\rceil e_j - L \\ &\leq \sum_{i=1}^{n} \frac{L}{p_i} c_i + \sum_{j=1}^{m} \left(\frac{L}{a_j} + 1 \right) e_j - L \\ &= L \sum_{i=1}^{n} \frac{c_i}{p_i} + L \sum_{j=1}^{m} \frac{e_j}{a_j} + \sum_{j=1}^{m} e_j - L \\ &= L(U-1) + \sum_{j=1}^{m} e_j. \end{aligned}$$

$h(L)$ is a linear function in L with slope $U - 1$ and an L-intercept at the point $L = B = \frac{\sum_{j=1}^{m} e_j}{1-U}$. Since $U < 1$, $h(L)$ has negative slope and thus for all $L \geq B$, $g(L) \leq h(L) \leq 0$. Hence τ will be feasible if and only if $g(L) \leq 0$ for all $L \in \boldsymbol{P}$. This proves the theorem. □

Complexity of Deciding Feasibility

In this section we show that if the utilization of a task system is bounded above by a constant $u < 1$, then feasibility can be decided in time $O(n^2 + m + p_{min})$ and space $O(p_{min})$ where p_{min} is the smallest period in the system.

First note that if a task system is feasible then

$$\sum_{j=1}^{m} e_j + c_{min} \leq p_{min}. \qquad (7)$$

If this is not the case then an invocation of the application task (c_{min}, p_{min}) will miss a deadline if it occurs simultaneously with invocations of all interrupt handlers.

We begin each feasibility test with a simple $O(m)$ test for (7) (given that p_{min} is known). If (7) does not hold then the task system is infeasible. For the remainder of this section we assume (7) holds.

The values of $f(L)$ for $0 < L < B$, can be computed in time $O(B + \sum_{j=1}^{m} \lfloor B/a_j \rfloor)$ and space $O(B)$. The time complexity bound can be simplified to

$$B + \sum_{j=1}^{m} \left\lfloor \frac{B}{a_j} \right\rfloor \le B + \sum_{j=1}^{m} \frac{B}{a_j} \le B\left(1 + \sum_{j=1}^{m} \frac{1}{a_j}\right) \le 2B$$

since $\sum_{j=1}^{m} 1/a_j < 1$ if $U < 1$. Therefore, computing $f(L)$ for $0 < L < B$ requires time $O(B)$. This is $O(p_{min})$ since from the definitions of B and u, $B \le p_{min}/(1-u)$.

By Theorem 3.6, to decide feasibility one must evaluate (6) at most $\sum_{i=1}^{n} \lfloor B/p_i \rfloor$ times. Since

$$\sum_{i=1}^{n} \left\lfloor \frac{B}{p_i} \right\rfloor \le \sum_{i=1}^{n} \frac{B}{p_i} \le B\sum_{i=1}^{n} \frac{1}{p_i} \le B\sum_{i=1}^{n} \frac{1}{p_{min}} \le$$

$$B\frac{n}{p_{min}} \le \frac{p_{min}}{(1-u)} \frac{n}{p_{min}} \le \frac{n}{1-u},$$

(6) must be evaluated $O(n)$ times. An evaluation of (6) requires time $O(n)$ (if $f(L)$ is pre-computed) hence the complexity of deciding the feasibility of a task system is $O(n^2 + p_{min} + m)$.

Some notes are in order. First, the p_{min} term in the complexity bound is not a function of the length of the input and hence the complexity of deciding feasibility is pseudo-polynomial time (*i.e.*, polynomial in both the length and magnitude of the inputs [1]). Second, note that B is not an input and must be computed. B can, however, be computed in time $O(n + m)$ and hence does not effect the overall complexity of deciding feasibility. The same is true of the complexity of finding p_{min}. Finally, in practice, since systems with thousands of tasks are not typically encountered but systems with high utilization are arguably more common, the actual time and space requirements for deciding feasibility are likely to be dominated by the constant factor $\frac{1}{1-u}$.

Other Dynamic Priority Task Systems

The proof of Theorem 3.5 is structured similar to the original Liu and Layland proof of the optimality of the preemptive deadline driven scheduling discipline. The basic arguments in the Liu and Layland proof are unaffected by the presence of interrupt handlers. In fact this observation holds for proofs of other results concerning deadline driven scheduling. That is, one can easily adapt the results for other dynamic priority schedulability problems to include the effects of interrupt handlers.

Two other schedulability problems that we have considered are: scheduling periodic tasks non-preemptively [3] and scheduling periodic tasks that share a set of serially reusable, non-preemptible resources [5].[1] The schedulability analysis reported for each problem can be extended to include interrupt handlers in exactly the same manner as Theorem 3.5 extends Theorem 3.1. For example, in [3] it was shown that if tasks are not allowed to preempt one another then a task system consisting of a set of n application tasks (with no interrupt handlers) can be scheduled by a non-preemptive, deadline driven scheduling algorithm if:

1. $\sum_{i=1}^{n} \frac{c_i}{p_i} \le 1$
2. $\forall i, 1 < i \le n, \forall L,\ p_1 < L < p_i, \quad L \ge c_i + \sum_{j=1}^{i} \left\lfloor \frac{L-1}{p_j} \right\rfloor c_j,$

(assuming that the tasks are sorted in non-decreasing order by period). The first condition requires that the processor not be overloaded. The second condition requires that for a specific set of interval lengths, the processor demand in intervals of those lengths cannot exceed the available processor time.

The following theorem shows how this result can be extended to include the effects of interrupt handlers. Application tasks are now not allowed to preempt one another but may be preempted at any time by interrupt handlers (*i.e.*, there are no new restrictions on the behavior of interrupt handlers).

Theorem 3.7: Let τ be a task system with n application tasks (c_1, p_1) … (c_n, p_n), sorted in non-decreasing order by period, and m interrupt handlers (e_1, a_1) … (e_m, a_m). If application tasks are not allowed to preempt one another then τ will be feasible if

1. $\forall L, L \ge 0, \quad L - f(L) \ge \sum_{i=1}^{n} \left\lfloor \frac{L}{p_i} \right\rfloor c_i$
2. $\forall i, L, 1 < i \le n, p_1 < L < p_i, L - f(L) \ge c_i + \sum_{j=1}^{i} \left\lfloor \frac{L-1}{p_j} \right\rfloor c_j$

Proof: The proof is a straightforward extension of the original. (See Appendix.) □

The results reported in [5] for the problem of scheduling periodic tasks that share a set of serially reusable, non-preemptible resources can be similarly extended.

---

[1] The focus in [5] was on *sporadic* tasks: repetitive tasks where p is only a lower bound on the time between successive invocations. However, as noted in [5], all schedulability conditions presented for sporadic tasks are also schedulability conditions for periodic tasks.

4. Feasibility Analysis of Static Priority Task Systems

To illustrate the generality of our analysis techniques, in this section we give feasibility conditions for task systems in which application tasks have fixed priority. These conditions are developed in much the same way as those in Section 3. They can also be derived from results previously reported in the literature.

The feasibility of sets of tasks with constant execution priority has been analyzed within the context of a *critical instant*. For a task T_i, a critical instant is a point in time at which if task T_i is invoked it will have its longest response time. For preemptible, periodic tasks, Liu and Layland have shown that a critical instant occurs whenever a task is invoked simultaneously with all tasks of higher priority [8].

Lehoczky *et al.* [7] give exact feasibility conditions for sets of periodic tasks in which task priorities are assigned in rate-monotonic order (*i.e.*, if $p_i < p_j$, then task T_i has higher execution priority than T_j). Consider a set of n periodic application tasks (c_1, p_1), ..., (c_n, p_n) sorted in non-decreasing order by period (*i.e.*, for any pair of tasks T_i and T_j, if $i < j$, then $p_i \le p_j$). The function $W_i(t) = \sum_{j=1}^{i} \lceil t/p_j \rceil c_j$ gives the amount of processor time requested by tasks T_1, ..., T_i, in the interval $[0, t-1]$ when all tasks are initially invoked at time 0. A set of n periodic tasks will be feasible under a rate monotonic priority assignment if and only if for all i, $1 \le i \le n$, there exists a value L, $0 < L \le p_i$, such that $L \ge W_i(L)$. If such an L exists for task T_i, then the initial invocation of T_i will complete execution at the first such time L. If such an L does not exist then the initial invocation of T_i will miss its deadline. As a result of the critical instant theorem, the set of tasks will be feasible if and only if the initial invocation of every task can be shown to meet its deadline.

To extend this result to a model including interrupt handlers, we model interrupt handlers as tasks with constant (highest) priority. The proof of Lehoczky *et al.* can then be applied directly to account for the time spent executing interrupt handlers. Note that by Lemma 3.2, $f(l)$ exactly characterizes the time spent executing interrupt handlers in the interval $[0, l]$. We can show that a task system will be feasible if and only if for all i, $1 \le i \le n$, there exists a time L in the interval $[0, p_i]$ such that

$$L \ge W_i(L) + f(L) \tag{8}$$

Alternative feasibility conditions can be derived using the analysis developed by Harbour *et al.* [2]. In [2], the authors develop necessary and sufficient feasibility conditions for a model where each task consists of *phases* with an arbitrary priority. A set τ of interrupt handlers and static priority application tasks with a rate-monotonic priority assignment is a special case of this model, in which each task and interrupt handler has a single phase, the interrupt handlers are given the highest priority, and the application tasks are assigned rate-monotonic priorities. We can use the general method in the Harbour paper to derive a feasibility condition for this special case. Let $F(t)$ be the amount of work requested by interrupt handlers in the interval $[0, t-1]$:

$$F(t) = \sum_{i=1}^{m} \left\lceil \frac{t}{a_i} \right\rceil e_i$$

If there is some time L in the interval $[0, p_i]$ such that

$$L \ge W_i(L) + F(L), \tag{9}$$

then task T_i will complete at the first such L. τ will be feasible if and only if (9) holds for every task.

5. Discussion

As the basis for a feasibility test, condition (9) is more appealing than (8), since (8) requires that we evaluate a recurrence relation and (9) is a closed form. This leads us to question whether or not there exists a closed form result for the problems considered in Section 3. While a desirable closed form would express feasibility in terms of processor utilization, note that feasibility is *not* a function of processor utilization. It is possible to construct *feasible* task systems with utilization equal to 1 (*e.g.*, $I = (.5p, p)$, $T = (.5p, p)$), and because application tasks may not preempt interrupt handlers, it is also possible to construct *infeasible* task systems with arbitrarily small processor utilization (*e.g.*, $I = (p, \infty)$, $T = (\varepsilon, p)$).

Another approach to consider is the replacement of $f(L)$ by $F(L)$ in (6). Indeed, by the definition of $f(L)$, for all L, $L \ge 0$, $f(L) \le F(L)$ and hence (6) with $f(L)$ replaced by $F(L)$ will be a sufficient condition for schedulability. It is not, however, a necessary condition. Consider a task system with one interrupt handler $I = (2, 3)$ and one application task $T = (1, 4)$. This task system satisfies the conditions in Theorem 3.6, and hence is feasible, but does not satisfy (6) with $f(L)$ replaced by $F(L)$ (*e.g.*, for $L = 4$, $F(L) = \lceil L/a \rceil e = \lceil 4/3 \rceil \times 2 = 4$, $L - F(L) = 0$, and $\lfloor L/p \rfloor c = \lfloor 4/4 \rfloor \times 1 = 1$. Hence $L - F(L) < \lfloor L/p \rfloor c$.)

The reason one can use $F(L)$ in place of $f(L)$ in (8) but not in (6) lies in a fundamental difference in the feasibility tests for static and dynamic priority systems. Both tests evaluate a similar condition at the endpoints of a well-defined set of intervals, all starting at time 0. Each condition has the form $L \ge W(L) + i(L)$, where L is a point in time, $i(L)$ is the time spent executing interrupt handlers in the interval $[0, L]$, and $W(L)$ is the demand due to application

tasks with invocations that must complete in the interval [0, L]. To analyze a static priority system, it suffices to demonstrate that there exists one L for which the condition is true. If such an L exists, the system is feasible. To analyze a dynamic priority system, it suffices to demonstrate that there does not exist an L for which the condition fails. If no such L exists, the system is feasible. In either case, for each value of L considered, $i(L)$ must be the exact amount of time spent executing interrupt handlers.

For static priority task systems, it is the case that the only values of L for which the condition can hold are values of L for which $f(L) = F(L)$. Therefore, it suffices to use $F(L)$ to represent the amount of time spent executing interrupt handlers in the interval [0, L]. Intuitively, this is because the condition can hold only at times when all interrupt handlers have completed execution.

6. Summary and Conclusions

Formal models of real-time systems frequently consist of sets of tasks that are invoked at regular intervals and which must complete execution before well-defined deadlines. In order to apply the results of analyses of these models, a practitioner must account for the effects of phenomena present in the implementation but not present in the formal model. One important factor is the cost of interrupt handling. We have studied, and solved, the feasibility and schedulability problems for periodic application tasks that must compete for the processor with interrupt handlers. Interrupt handlers are modeled as periodic tasks that always have priority over application tasks. For the feasibility problem, the emphasis has been on application tasks that may be preempted at arbitrary points, although other paradigms of non-preemptive execution have been considered. For the schedulability problem, the emphasis has been on deadline driven methods.

The approach has been to quantify the processor time spent executing interrupt handlers in an arbitrary interval. We derived a recurrence relation that bounds this time and showed that by expressing feasibility and schedulability in terms of a processor demand function, we can use the recurrence to extend several existing schedulability results concerning deadline driven scheduling to include the effects of interrupt handlers. For preemptible, periodic tasks, feasibility in the presence of interrupt handlers can be decided in time $O(n^2 + p_{min} + m)$ where n is the number of application tasks, p_{min} is the smallest application task period, and m is the number of interrupt handlers. Lastly, we compared the analysis of the static priority schedulability problem to the dynamic priority schedulability problem.

In the future we will consider several refinements to the model presented in Section 2. First, the present work assumed that interrupt handlers are strictly periodic. In practice this is rarely the case. We will next consider models of interrupt handlers that are invoked sporadically. For purely preemptive systems, this generalization will likely have little effect, however, for non-preemptive systems (where fundamental differences between periodic and sporadic tasks have already been demonstrated) this will be more challenging. Second, here we assumed that interrupt handlers and application tasks are independent in the sense that their invocations are not related. In most systems application tasks are invoked in response to interrupt handlers. When application tasks are sporadic, a coupling of interrupt handlers and application tasks complicates the schedulability analysis. Lastly, in practice it is not always the case that interrupt handlers execute with priority strictly higher that those of application tasks. For example, application tasks may disable interrupts (*e.g.*, as a side effect of a system call) and thus execute when an interrupt handler would otherwise have done so. An implication of this is that interrupt handlers can no longer be analyzed separately from application tasks. Moreover, disabling interrupts affects correctness conditions for interrupt handlers (an issue we have not addressed in this work).

7. References

1. Garey, M.R., Johnson, D.S., *Computing and Intractability, A Guide to the Theory of NP-Completeness*, W.H. Freeman & Co., New York, 1979.

2. Harbour, M.G., Klein, M.H., Lehoczky, J., *Fixed Priority Scheduling of Periodic Tasks with Varying Execution Priority*, Proc. 12th IEEE Real-Time Systems Symp., San Antonio, TX, December 1991, pp. 116-128.

3. Jeffay, K., Stanat, D.F., Martel, C.U., *On Non-Preemptive Scheduling of Periodic and Sporadic Tasks*, Proc. 12th IEEE Real-Time Systems Symp., San Antonio, TX, December 1991, pp. 129-139.

4. Jeffay, K., Stone, D.L., Poirier, D., *YARTOS: Kernel support for efficient, predictable real-time systems*, in "Real-Time Programming," W. Halang and K. Ramamritham, eds., Pergamon Press, Oxford, UK, 1992, pp. 7-12.

5. Jeffay, K., *Scheduling Sporadic Tasks with Shared Resources in Hard-Real-Time Systems*, Proc. 13th IEEE Real-Time Systems Symp., Phoenix, AZ, December 1992, pp. 89-99.

6. Katcher, D.I., Arakawa, H., Strosnider, J.K., *Engineering and Analysis of Fixed Priority Schedulers*, IEEE Trans. on Software Eng., 1993 (to appear).

7. Lehoczky, J., Sha, L., Ding, Y., *The Rate Monotonic Scheduling Algorithm: Exact Characterization and*

Average Case Behavior, Proc. of the 10th IEEE Real-Time Systems Symp., Santa Monica, CA, December 1989, pp. 166-171.

8. Liu, C.L., Layland, J.W., *Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment*, Journal of the ACM, Vol. 20, No. 1, (January 1973), pp. 46-61.

9. Rajkumar, R., Sha, L, Lehoczky, J.P., *On Countering the Effects of Cycle-Stealing in a Hard Real-Time Environment*, Proc. 8th IEEE Real-Time Systems Symp., December 1987, San Jose, CA, pp. 2-11.

Appendix

The following are the proofs of Theorems stated but not proved in the text.

Theorem 3.1: A set of periodic tasks will be feasible if and only if for all $L > 0$,

$$L \geq \sum_{i=1}^{n} \left\lfloor \frac{L}{p_i} \right\rfloor c_i \qquad (10)$$

Proof: We demonstrate that (1) $\Leftrightarrow$ (10).

If $\sum_{i=1}^{n} c_i/p_i \leq 1$, then for all L, $L > 0$,

$$\sum_{i=1}^{n} \frac{Lc_i}{p_i} \leq L$$

$$\sum_{i=1}^{n} \left\lfloor \frac{L}{p_i} \right\rfloor c_i \leq L$$

and hence (1) $\Rightarrow$ (10). We show (1) $\Leftarrow$ (10) by establishing the contrapositive (*i.e.*, $\neg(1) \Rightarrow \neg(10)$). To show $\neg(10)$, it suffices to demonstrate the existence of an $L > 0$ for which (10) does not hold. If $\sum_{i=1}^{n} c_i/p_i > 1$, then for $L = \mathrm{LCM}(p_1, p_2, ..., p_n)$,

$$L < \sum_{i=1}^{n} \frac{Lc_i}{p_i} = \sum_{i=1}^{n} \left\lfloor \frac{L}{p_i} \right\rfloor c_i$$

and hence (1) $\Leftarrow$ (10). □

Theorem 3.7: Let τ be a task system with n application tasks $(c_1, p_1) \ldots (c_n, p_n)$, sorted in non-decreasing order by period, and m interrupt handlers $(e_1, a_1) \ldots (e_m, a_m)$. If application tasks are not allowed to preempt one another then τ will be feasible if

$$1.\ \forall L, L \geq 0, \quad L - f(L) \geq \sum_{i=1}^{n} \left\lfloor \frac{L}{p_i} \right\rfloor c_i$$

$$2.\ \forall i, 1 < i \leq n, \forall L, p_1 < L < p_i,$$

$$L - f(L) \geq c_i + \sum_{j=1}^{i} \left\lfloor \frac{L-1}{p_j} \right\rfloor c_j$$

Proof: We show that a non-preemptive deadline driven scheduling algorithm can schedule a task system if it satisfies conditions 1 and 2. This is shown by contradiction.

Assume that τ satisfies conditions 1 and 2 and yet an application task in τ misses a deadline at some point in time when τ is scheduled by the deadline driven algorithm. The proof proceeds by deriving upper bounds on the processor demand for an interval ending at the time at which a task misses a deadline.

Let t_d be the earliest point in time at which a deadline is missed. τ can be partitioned into two disjoint subsets:

S_1 = the set of application tasks that have an invocation with a deadline at time t_d.

S_2 = the set of application tasks that have an invocation occurring prior to time t_d with deadline after t_d.

Let $b_1, b_2, ..., b_k$ be the invocation times immediately prior to t_d of the tasks in S_2. There are two cases to consider.

Case 1: None of the invocations of tasks in S_2 occurring at times $b_1, b_2, ..., b_k$ are scheduled prior to t_d.

Let $d_{a,b}$ be the processor demand of the application tasks in the interval $[a, b]$. Let t_0 be the end of the last period prior to t_d in which the processor was idle. If the processor has never been idle let $t_0 = 0$. In the interval $[t_0, t_d]$, the processor demand is the total processing requirement of the tasks that are invoked at or after time t_0, with deadlines at or before time t_d. This gives

$$d_{t_0,t_d} \leq \sum_{i=1}^{n} \left\lfloor \frac{t_d - t_0}{p_i} \right\rfloor c_i .$$

(Equality holds if all tasks are invoked at time t_0.) Since there is no idle period in the interval $[t_0, t_d]$ and since a task misses a deadline at t_d, it follows that $d_{t_0,t_d} > (t_d - t_0) - f(t_d - t_0)$. Therefore

$$(t_d - t_0) - f(t_d - t_0) < \sum_{i=1}^{n} \left\lfloor \frac{t_d - t_0}{p_i} \right\rfloor c_i \leq \sum_{i=1}^{n} \frac{t_d - t_0}{p_i} c_i .$$

However, this contradicts condition 1 and thus establishes the theorem for Case 1.

Case 2: Some of the invocations of tasks in S_2 occurring at times $b_1, b_2, ..., b_k$ are scheduled prior to t_d.

Let T_i be the last application task in S_2 scheduled prior to time t_d. Let $t_i < t_d$ be the point in time at which the invocation of T_i occurring immediately prior to t_d commences execution. Note that if the processor is ever idle in the interval $[t_i, t_d]$, then the analysis of Case 1 can be applied directly to the interval $[t_0, t_d]$, where $t_i < t_0 < t_d$

is the end of the last idle period prior to time t_d, to reach a contradiction of condition 1. Therefore, assume the processor is fully utilized during the interval $[t_i, t_d]$.

Let T_k be an application task that misses a deadline at time t_d. Because of our choice of task T_i and our use of deadline driven scheduling, it follows that $t_i < t_d - p_k$. That is, the invocation of the task T_k that does not complete execution by time t_d occurs within the interval $[t_i, t_d]$. We now show that if the invocation in question of task T_i is scheduled prior to time t_d, then there must have existed enough processor time in $[t_i, t_d]$ to schedule all invocations of tasks occurring after time t_i with deadlines at or before time t_d. To begin, we derive an upper bound on d_{t_i,t_d}, the processor demand for the interval $[t_i, t_d]$.

The following facts hold for Case 2:

i) Other than T_i, no application task T_j, with period p_j, such that $p_j \geq t_d - t_i$, executes in the interval $[t_i, t_d]$.

Since the invocation of task T_i scheduled at time t_i has a deadline after time t_d and is the last such invocation scheduled prior to t_d, every other application task executed in $[t_i, t_d]$ must have a deadline at or before t_d because of the deadline driven discipline.

ii) Other than T_i, no application task that is scheduled in $[t_i, t_d]$ could have been invoked at time t_i.

Again, as a consequence of the definition of task T_i, other than T_i, every application task scheduled in $[t_i, t_d]$ has a deadline at or before t_d. Therefore, if a task $T_{i'}$, that is scheduled in $[t_i, t_d]$ had been invoked at t_i, the deadline driven algorithm would have scheduled task $T_{i'}$ instead of task T_i at time t_i.

Since $p_i > t_d - t_i$, fact (*i*) above indicates that only tasks $T_1 \ldots T_i$ need be considered in computing d_{t_i,t_d}. Since the invocation of task T_i that is scheduled at time t_i has a dead line after time t_d, all task invocations occurring prior to time t_i with deadlines at or before t_d must have been satisfied by t_i and hence do not contribute to d_{t_i,t_d}. Similarly, since T_i has the last invocation with deadline after t_d that executes prior to t_d, all invocations of tasks T_1 - T_{i-1} occurring prior to time t_d with deadlines after t_d, need not be considered. Lastly, since none of the invocations of tasks T_1 - T_{i-1} that are scheduled in the interval $[t_i, t_d]$ occurred at time t_i, the demand due to tasks T_1 - T_{i-1} in the interval $[t_i, t_d]$ is the same as in the interval $[t_i + 1, t_d]$. These observations, plus the fact the invocation of task T_i scheduled at time t_i must be completed before time t_d, indicate that the processor demand in $[t_i, t_d]$ is bounded by

$$d_{t_i,t_d} \leq c_i + \sum_{j=1}^{i-1} \left\lfloor \frac{t_d - (t_i - 1)}{p_j} \right\rfloor c_j \qquad (11)$$

Let $L = t_d - t_i$. Substituting L into the (11) yields

$$d_{t_i,t_d} \leq c_i + \sum_{j=1}^{i-1} \left\lfloor \frac{L - 1}{p_j} \right\rfloor c_j \qquad (12)$$

Since there is no idle time in $[t_i, t_d]$, and since a task missed a deadline at t_d, it follows that $d_{t_i,t_d} > (t_d - t_i) - f(t_d - t_i)$ or simply $d_{t_i,t_d} > L - f(L)$. Combining this with (12) yields

$$L - f(L) < d_{t_i,t_d} \leq c_i + \sum_{j=1}^{i-1} \left\lfloor \frac{L - 1}{p_j} \right\rfloor c_j \qquad (13)$$

Since $p_i > t_d - t_i$, we have $p_i > L$. Since $t_i < t_d - p_k$ (recall that k is the index of a task that missed a deadline at time t_d) we have $t_d - t_i > p_k \geq p_1$, and hence $L > p_1$. Therefore (13) contradicts condition 2 and establishes the theorem for Case 2.

We have shown that in either case, if an application task misses a deadline when scheduled by the non-preemptive deadline driven algorithm, then either condition 1 or condition 2 must have been violated. This proves the theorem. □