On Non-Preemptive Scheduling of Periodic and Sporadic Tasks

Kevin Jeffay[*] *Donald F. Stanat* *Charles U. Martel*[**]

University of North Carolina at Chapel Hill
Department of Computer Science

University of California at Davis
Computer Science Division

Abstract: This paper examines a fundamental problem in the theory of real-time scheduling, that of scheduling a set of periodic or sporadic tasks on a uniprocessor without preemption and without inserted idle time. We exhibit a necessary and sufficient set of conditions ***C*** for a set of periodic or sporadic tasks to be schedulable for arbitrary release times of the tasks. We then show that any set of periodic or sporadic tasks that satisfies conditions ***C*** can be scheduled with an *earliest deadline first* (EDF) scheduling algorithm.

We also address the question of schedulability of a set of tasks with specified release times. For sets of sporadic tasks with specified release times, we show that the conditions ***C*** are again necessary and sufficient for schedulability. However, for sets of periodic tasks with specified release times, the conditions ***C***, while sufficient, are not necessary. In fact, we show that determining whether a set of periodic tasks with specified release times is schedulable is intractable (*i.e.*, NP-hard in the strong sense). Moreover, we show that the existence of a universal algorithm for scheduling periodic tasks with specified release times would imply that P = NP.

1. Introduction

The concept of a task that is invoked repeatedly is central to both the design and analysis of real-time systems. In particular, formal studies of real-time systems frequently represent the time-constrained processing requirements of the system as a set of *periodic* or *sporadic* tasks with deadlines [Liu & Layland 73, Leung & Merrill 80, Mok 83]. A periodic task is invoked at regular intervals, while a sporadic task is invoked at arbitrary times but with a specified minimum time interval between invocations.

In practice, periodic tasks are commonly found in applications such as avionics and process control when accurate control requires continual sampling and processing of data. Sporadic tasks are associated with event-driven processing such as responding to user inputs or non-periodic device interrupts; these events occur repeatedly, but the time interval between consecutive occurrences varies and can be arbitrarily large. Periodic and sporadic tasks were used, for example, to represent the timing constraints in an interactive 3-dimensional graphics display system used for research in *virtual worlds* [Chung *et al.* 89, Jeffay 91]. The graphics system uses a head-mounted display system consisting of a helmet with miniature television monitors embedded in it, and hardware for tracking the position of the helmet and a hand-held pointing device. A computer generated image of a 3-dimensional "virtual world" is displayed on the television monitors in the helmet. The goal of the system is to track the user's head and the pointing device in real-time and to update the image displayed in the helmet so as to maintain the illusion that the user is immersed in an artificial world. There are two separate real-time concerns in this application. First, the system must provide an image to update the display approximately every 30 milliseconds. Generating a new image is naturally represented as a periodic process. Second, as the user's head or the pointing device is moved, the motions must be tracked and the consequences incorporated into the generation of the next image. Because both the user's head and the pointing device may remain stationary for some time, the process associated with tracking them is invoked sporadically.

Given a real-time system, the goal is to schedule the system's tasks on a processor, or processors, so that each task completes execution before a specified deadline. In this paper we consider a fundamental real-time scheduling problem, that of non-preemptive scheduling of a set of periodic or sporadic tasks on a uniprocessor. Non-preemptive scheduling on a uniprocessor is important for a variety of reasons:

[*] Supported in parts by grants from the National Science Foundation (number CCR-9110938), and from Digital Equipment Corporation.
[**] Supported by a grant from the National Science Foundation (number CCR-9023727).

In: Proceedings of the Twelfth IEEE Real-Time Systems Symposium, San Antonio, Texas, December 1991, IEEE Computer Society Press, pp. 129-139.

- In many practical real-time scheduling problems such as I/O scheduling, properties of device hardware and software either make preemption impossible or prohibitively expensive.
- Non-preemptive scheduling algorithms are easier to implement than preemptive algorithms, and can exhibit dramatically lower overhead at run-time.
- The overhead of preemptive algorithms is more difficult to characterize and predict than that of non-preemptive algorithms. Since scheduling overhead is often ignored in scheduling models (including ours), an implementation of a non-preemptive scheduler will be closer to the formal model than an implementation of a preemptive scheduler.
- Non-preemptive scheduling on a uni-processor naturally guarantees exclusive access to shared resources and data, thus eliminating both the need for synchronization and its associated overhead.
- The problem of scheduling all tasks without preemption forms the theoretical basis for more general tasking models that include shared resources [Jeffay 89b, 90].

Many others have looked at variations of this problem; most describe sufficient conditions for scheduling tasks. We give necessary and sufficient conditions. Furthermore, we show that a particular algorithm can always be used for scheduling a large class of sets of tasks. (We will review related work in more detail in Section 3.)

The remainder of this paper is composed of five major sections. The following section presents our scheduling model. Section 3 briefly reviews the literature in real-time scheduling. Section 4 proves the non-preemptive EDF algorithm is universal for sets of tasks, whether they be periodic or sporadic. Section 5 demonstrates the absence of a universal algorithm for periodic tasks with specified release times and proves that the problem of deciding schedulability of a set of concrete periodic tasks is intractable. Section 6 discusses these results.

2. The Model

A task is a sequential program that is *invoked* by each occurrence of a particular *event*. An event is a stimulus generated by a process that is either external to the system (*e.g.*, interrupts from a device) or internal to the system (*e.g.*, clock ticks). We assume that events are generated repeatedly with some maximum frequency; thus, the time interval between successive invocations of a task will be of some minimal length. Each invocation of a task results in a single execution of the task at a time specified by a scheduling algorithm.

Formally, a ***task*** T is a pair (c, p) where

- c is the *computational cost*: 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 $period$: the minimal interval between invocations of task T.

Throughout this paper we assume time is discrete and clock ticks are indexed by the natural numbers. Task invocations occur and task executions begin at clock ticks; each of the parameters c and p is expressed as a multiple of (the interval between) clock ticks. If a task with cost c begins execution at time t and is executed without interruption on a uniprocessor, then the execution is completed at time $t + c$.

We consider two paradigms of task invocation: periodic and sporadic. If T is ***periodic***, the period p specifies a *constant* interval between invocations. If T is ***sporadic***, p specifies a *minimum* interval between invocations.

The definition of the behavior of a task depends on whether it is periodic or sporadic. The ***behavior of a periodic task*** $T = (c, p)$ is given by the following rules for the invocation and execution of T. If t_k is the time of the k^{th} invocation of task T, then

i) The $(k+1)^{th}$ invocation of task T will occur at time $t_{k+1} = t_k + p$.

ii) The k^{th} execution of task T must begin no earlier than t_k and be completed no later than the *deadline* of $t_k + p$. This requires that c units of processor time be allocated to the execution of T in the interval $[t_k, t_k + p]$.

The behavior of a sporadic task is slightly less constrained than that of a periodic task. The ***behavior of a sporadic task*** $T = (c, p)$ is given by the following rules for the invocation and execution of T. If t_k is the time of the k^{th} invocation of task T, then

i) The $(k+1)^{th}$ invocation of T will occur *no earlier than* time $t_k + p$; thus, $t_{k+1} \geq t_k + p$.

ii) The k^{th} execution of task T must begin no earlier than t_k and be completed no later than the *deadline* of $t_k + p$.

Thus the behaviors of periodic and sporadic tasks differ only in the first rule. We assume invocations of sporadic tasks are independent in the sense that the time a sporadic task is invoked depends only upon the time of its last invocation and not upon the invocation times of any other task.

Note that the worst case behavior of a sporadic task $T = (c, p)$ ("worst" in the sense of requiring the most processor time), occurs when T behaves like a periodic task, that is, T is invoked every p time steps.

We wish to investigate the scheduling of sets of tasks that compete for processing resources. The difficulty of scheduling tasks can be affected by the times that tasks are first invoked. A ***concrete task*** is a pair (T, R), where T is a task, and R is a non-negative integer that is the time of the first invocation, or the *release time*, of T. The ***behavior*** of (T, R) is the behavior of T constrained further by the rule that the first invocation of T occurs at time R. Once released, tasks are invoked repeatedly forever.

A ***set of periodic (sporadic) tasks*** $\tau = \{T_1, T_2, ..., T_n\}$ is a set of tasks indexed from 1 to n, where for each i, $1 \le i \le n$, $T_i = (c_i, p_i)$. A ***concrete set of periodic (sporadic) tasks*** $\omega = \{(T_1,R_1), (T_2,R_2) ..., (T_n,R_n)\}$ is a set of concrete tasks indexed from 1 to n, where R_i is the release time of task T_i.[1] There is a natural many-to-one relation between concrete tasks and tasks. We say the task T ***generates*** a concrete task (T, R) and a concrete task (T, R) ***is generated from*** the task T. This relation extends naturally to a relation between concrete task sets and task sets. Let $\tau = \{T_1, T_2, ..., T_n\}$ be a task set and let $\omega = \{(T_1,R_1), (T_2,R_2) ..., (T_n,R_n)\}$ be a concrete task set. Then the task set τ ***generates*** the concrete task set ω and ω ***is generated from*** τ.

If an execution of a task has a deadline of time t_d, and execution is not complete at time t_d, then we say the task has ***missed a deadline***. A scheduling algorithm specifies, at each time t, which task if any shall begin, continue, or resume execution. A *concrete task set* ω is ***schedulable*** if it is possible to schedule the executions of tasks of ω so that no task ever misses a deadline when tasks are released at their specified release times. A *task set* τ is ***schedulable*** if *every* concrete task set ω generated from τ is schedulable. A scheduling algorithm ***schedules*** a concrete task set ω if no task of ω ever misses a deadline when the algorithm is applied.

In this paper, we restrict ourselves to the case of *non-preemptive* scheduling on a uniprocessor; that is, we assume a scheduling algorithm that does not interrupt the execution of any task once it has begun. We also restrict ourselves to scheduling on a uniprocessor *without inserted idle time*; which means that the scheduling algorithm does not permit the processor to be idle if there is a task that has been invoked but has not completed execution. To save space and avoid tedium, we will not mention these restrictions in the remainder of the paper.

Note that a task set is schedulable if and only if the tasks can be scheduled for *any* set of release times. In contrast, each member of a concrete task set has a specified release time, and showing that a concrete task set is schedulable only establishes that its specified release times can be accommodated. For example, under the restrictions of no preemption and no inserted idle time, a periodic task set that is *not* schedulable may generate sets of concrete tasks that *are* schedulable as well as sets which are not. For example, the set of two periodic tasks $\tau = \{(3, 5), (4, 10)\}$ generates both schedulable and unschedulable concrete task sets: the set consisting of $\omega' = \{((3,5), 0), ((4,10), 0)\}$ is schedulable but the set consisting of $\omega'' = \{((3,5), 1), ((4,10), 0)\}$ is not.

A scheduling algorithm is said to be ***universal for concrete periodic (sporadic) tasks*** if the algorithm schedules every schedulable set of concrete periodic (sporadic) tasks. A scheduling algorithm is said to be ***universal for periodic (sporadic) tasks*** if the algorithm schedules any concrete periodic (sporadic) task set generated from a set of schedulable tasks. We will show that a deadline-driven scheduling algorithm that is a non-preemptive version of the *earliest deadline first* (EDF) algorithm [Liu & Layland 73], is universal for either periodic or sporadic tasks as well as for concrete sporadic tasks. For concrete periodic tasks, however, things are more complex. If a set of concrete periodic tasks ω is generated from a periodic task set τ that is schedulable, then ω is schedulable (and indeed can be scheduled by the EDF algorithm). But if τ is not schedulable, then ω may or not be schedulable. In the general case, we show that determining whether ω is schedulable is NP-hard in the strong sense. Moreover, we establish that if there exists a universal scheduling algorithm for concrete periodic tasks that takes only a polynomial amount of time to make each scheduling decision, then P = NP. Thus it is unlikely that there exists a universal algorithm for scheduling concrete periodic tasks.

3. Previous Work

Previous work in the area of real-time scheduling has mainly focused on the analysis of preemptive scheduling algorithms. A well-known result is that the preemptive

[1] More properly, τ and ω are multisets since there can exist more than one task in τ with the same cost and period and more than one task in ω with the same cost, period, and release time.

EDF algorithm is universal for all sets of concrete periodic tasks for which the release times are all 0 [Liu & Layland 73]. This result generalizes to all periodic task sets (*i.e.*, for concrete periodic tasks with arbitrary release times) [Jeffay 89a]. The extension of the preemptive problem to multiprocessors was considered in [Dhall & Liu 78] and [Bertossi & Bonuccelli 83].

Work with non-preemptive scheduling algorithms has typically been confined to consideration of models where processes are invoked only once, there is a precedence order between the processes, and each process requires only a single unit of computation time and must be completed before a deadline [Garey *et al.* 81, Frederickson 83].

A more general characterization of periodic tasks has been considered in [Leung & Merrill 80], [Lawler & Martel 81], [Leung & Whitehead 82], and [Mok 83]. In these works, when a task is invoked, it may have a deadline nearer than the time of the next task invocation. For this more general model, Mok has shown that the problem of deciding schedulability of a set of periodic tasks which use semaphores to enforce mutual exclusion constraints is NP-hard [Mok 83]. Our paper demonstrates the intractability of deciding schedulability for an even simpler characterization of periodic tasks and additionally provides strong evidence that there may not exist a universal non-preemptive scheduling algorithm for periodic tasks with specified release times.

4. Non-Preemptive Scheduling of Periodic and Sporadic Tasks

We first consider the problem of scheduling a set of periodic or sporadic tasks non-preemptively on a single processor. We begin by developing a set of relations on the costs and periods of tasks that must hold if a task set is to be schedulable. If the elements of a task set do not satisfy these relationships then no scheduling algorithm can schedule the tasks. We show that periodic and sporadic task sets have the same requirements for schedulability. Having identified necessary conditions for schedulability, we then exhibit an algorithm which schedules any set of periodic or sporadic tasks that satisfy the necessity conditions. This establishes directly that the algorithm is universal for scheduling sets of tasks and proves that the necessary conditions are also sufficient.

The following theorem establishes necessary conditions for schedulability for a periodic task set. Our development of these conditions is motivated by the early work of Sorenson [Sorenson 74, Sorenson & Hamacher 75].

Theorem 4.1: Let $\tau_p = \{T_1, T_2, ..., T_n\}$, where $T_i = (c_i, p_i)$, be a set of periodic tasks sorted in non-decreasing order by period (*i.e.*, for any pair of tasks T_i and T_j, if $i > j$, then $p_i \geq p_j$). If τ_p is schedulable then

1) $\sum_{i=1}^{n} \frac{c_i}{p_i} \leq 1$,

2) $\forall i,\ 1 < i \leq n;\ \forall L,\ p_1 < L < p_i$:

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

Informally, condition (1) can be thought of as a requirement that the processor not be overloaded. If a periodic task T has a cost c and period p, then c/p is the fraction of processor time consumed by T over the lifetime of the system (*i.e.*, the utilization of the processor by T). The first condition simply stipulates that the cumulative processor utilization cannot exceed unity; reflecting our restriction to a uniprocessor.[2]

Condition (2) reflects our restriction to non-preemptive scheduling without inserted idle time. The right hand side of the inequality in condition (2) is a least upper bound on the processor demand that can be realized in an interval of length L starting at the time an invocation of a task T_i is scheduled, and ending sometime before the deadline for the invocation. For a set of tasks to be schedulable, the demand in the interval L must always be less than or equal to the length of the interval. Although this is semantically similar to the requirement that the processor not be over-utilized, it can easily be shown that conditions (1) and (2) are in fact not related. It is possible to conceive of both *schedulable* task sets that have a processor utilization of 1.0, and *unschedulable* task sets that have arbitrarily small processor utilization.

Proof: We prove the contrapositive of the Theorem: if a set of periodic tasks τ_p does not satisfy condition (1) or condition (2) then there exists a concrete set of periodic tasks, generated from τ_p that is not schedulable.

For a concrete set of tasks ω, define the *processor demand* in the time interval $[a, b]$, written $d_{a,b}$, as the maximum amount of processing time required by ω in the interval $[a, b]$ to complete execution of all invocations of tasks with deadlines in the interval $[a, b]$. The processor demand

[2] In [Liu & Layland 73] it was shown that a concrete set of periodic tasks $\omega_p = \{(T_1,R_1), (T_2,R_2), ..., (T_n,R_n), \}$ where $R_i = 0$ for all i (*i.e.*, all tasks are released at time 0), is schedulable on a uniprocessor when preemption is allowed at arbitrary points in time if and only if condition (1) alone is satisfied.

in the interval $[a, b]$ will be a function of costs and periods of the tasks in ω, the length of the interval, the invocation times of tasks prior to or at time a, and the amount of computation time required to complete execution of task invocations that occurred prior to time a with deadlines at or before time b that have not completed execution by time a. ω is schedulable if and only if for all intervals $[a, b]$, $d_{a,b} \leq b - a$.

Consider the concrete set of periodic tasks $\omega_p = \{(T_1,R_1), (T_2,R_2), ..., (T_n,R_n)\}$, generated from τ_p where $R_i = 0$ for all i, $1 \leq i \leq n$ (*i.e.*, the concrete set of tasks wherein all tasks are released at time 0). Let $t = p_1 p_2 \ldots p_n$. In the interval $[0,t]$, task i must receive $\frac{t}{p_i} c_i$ units of processor time to ensure it does not miss a deadline in the interval $[0, t]$. Therefore, in the interval $[0, t]$

$$d_{0,t} = \sum_{i=1}^{n} \frac{t}{p_i} c_i ,$$

and hence

$$\frac{d_{0,t}}{t} = \sum_{i=1}^{n} \frac{c_i}{p_i} .$$

If condition (1) does not hold then $d_{0,t} > t$, and hence ω_p is not schedulable.

For condition (2), consider the concrete set of periodic tasks $\omega_p = \{(T_1,R_1), (T_2,R_2), ..., (T_n,R_n)\}$ generated from τ_p, where for some value of i, $1 < i \leq n$, $R_i = 0$, and $R_j = 1$ for $1 \leq j \leq n, j \neq i$. This gives rise to the pattern of task invocations shown in Figure 4.1. Since neither preemption nor inserted idle time are allowed, task T_i must execute in the interval $[0, c_i]$. For all L, $p_1 < L < p_i$, in the interval $[0, L]$, the processor demand, $d_{0,L}$, is given by

$$d_{0,L} = c_i + \sum_{j=1}^{i-1} \left\lfloor \frac{L-1}{p_j} \right\rfloor c_j .$$

The demand consists of the cost of executing the initial invocation of task T_i plus the processor demand due to tasks 1 through i–1 in the interval $[1, L]$. (Note that tasks with periods greater than or equal to p_i have no invocations with deadlines in the interval $[0, L]$ and hence do not contribute to the processor demand in the interval $[0, L]$.)

If condition (2) does not hold then $d_{0,L} > L$, and hence ω_p is not schedulable. □

Conditions (1) and (2) from Theorem 4.1 are also necessary for scheduling a set of sporadic tasks non-preemptively.

Corollary 4.2: If a set of sporadic tasks $\tau_s = \{T_1, T_2, ..., T_n\}$, sorted in non-decreasing order by period, is

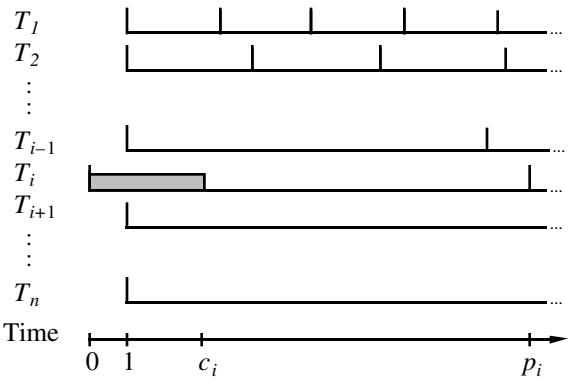

Figure 4.1

schedulable then τ_s satisfies conditions (1) and (2) from Theorem 4.1.

Proof: This can be proved independently of Theorem 4.1, however, it follows from Theorem 4.1 using the fact that one of the behaviors of a concrete set of sporadic tasks is as a concrete set of periodic tasks. □

The constructions used in the proof of Theorem 4.1, in fact, precisely characterize the worst case pattern of task invocations for any set of tasks. We will show that if a set of tasks can be scheduled (without preemption) when invoked as shown in Figure 4.1, then the tasks are indeed schedulable. Specifically, we demonstrate the existence of a non-preemptive scheduling algorithm which is guaranteed to schedule any periodic or sporadic task set that satisfies the necessity conditions.

The basic scheduling algorithm we consider is the *earliest deadline first* (EDF) algorithm [Liu & Layland 73]. When selecting a task for execution, an EDF scheduling algorithm chooses the task with an uncompleted invocation with the earliest deadline. Ties between tasks with identical deadlines are broken arbitrarily. With a non-preemptive formulation of the EDF algorithm, once a task is selected, the task is immediately executed to completion. Unless the processor is idle, such a scheduler will make dispatching decisions only when a task terminates an execution. If the processor is idle then the first task to be invoked is scheduled. If multiple tasks are invoked simultaneously then the one with the nearest deadline is scheduled. We assume that both the task selection process and the process of dispatching a task take no time in our discrete time system.

We next demonstrate the universality of the EDF algorithm for scheduling sporadic tasks without preemption. This means if any non-preemptive algorithm schedules a set of sporadic tasks, then the EDF algorithm

will as well. To prove universality, it suffices to show that conditions (1) and (2) are sufficient to ensure that the EDF algorithm schedules any concrete set of sporadic tasks generated from a set of schedulable sporadic tasks.

Theorem 4.3: Let τ_s be a set of sporadic tasks $\{(c_1, p_1), (c_2, p_2), ..., (c_n, p_n)\}$ sorted in non-decreasing order by period. If τ_s satisfies conditions (1) and (2) from Theorem 4.1 then the non-preemptive EDF scheduling algorithm will schedule any concrete set of sporadic tasks generated from τ_s.

Proof: By contradiction. Assume the contrary, *i.e.*, that τ_s satisfies conditions (1) and (2) from Theorem 4.1 and yet there exists a concrete set of sporadic tasks ω_s generated from τ_s, such that a task in ω_s misses a deadline at some point in time when ω_s is scheduled by the EDF 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. ω_s can be partitioned into three disjoint subsets:

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

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

S_3 = the set of tasks not in S_1 or S_2.

Tasks in S_3 either have a release time greater than t_d, or they have not been invoked immediately prior to time t_d. As will shortly become apparent, to bound the processor demand prior to t_d, it suffices to concentrate on the tasks in S_2. 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 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

$$d_{t_0,t_d} \leq \sum_{j=1}^{n} \left\lfloor \frac{t_d - t_0}{p_j} \right\rfloor c_j .$$

(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 $d_{t_0,t_d} > t_d - t_0$. Therefore

$$t_d - t_0 < \sum_{j=1}^{n} \left\lfloor \frac{t_d - t_0}{p_j} \right\rfloor c_j \leq \sum_{j=1}^{n} \frac{t_d - t_0}{p_j} c_j ,$$

and hence

$$1 < \sum_{j=1}^{n} \frac{c_j}{p_j} .$$

However, this contradicts condition (1) and 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 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 a task that misses a deadline at time t_d. Because of our choice of task T_i and our use of EDF 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 task T_i, no task with period greater than or equal to $t_d - t_i$ 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 to t_d, every other task executed in $[t_i, t_d]$ must have a deadline at or before t_d because of the EDF discipline.

ii) Other than task T_i, no task which 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 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 EDF 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 deadline 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

$$d_{t_i,t_d} \le c_i + \sum_{j=1}^{i-1} \left\lfloor \frac{t_d - (t_i + 1)}{p_j} \right\rfloor c_j . \qquad (4.1)$$

Let $L = t_d - t_i$. Substituting L into the (4.1) yields

$$d_{t_i,t_d} \le c_i + \sum_{j=1}^{i-1} \left\lfloor \frac{L - 1}{p_j} \right\rfloor c_j . \qquad (4.2)$$

Since there is no idle time in $[t_i, t_d]$, and since a task missed a deadline at t_d, it follows that $d_{t_i,t_d} > t_d - t_i$ or simply $d_{t_i,t_d} > L$. Combining this with (4.2) yields

$$L < d_{t_i,t_d} \le c_i + \sum_{j=1}^{i-1} \left\lfloor \frac{L - 1}{p_j} \right\rfloor c_j , \qquad (4.3)$$

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 $t_d - t_i > p_k \ge p_1$, and hence $L > p_1$. Therefore (4.3) contradicts condition (2) and establishes the theorem for Case 2.

We have shown that in either case, if an element of a concrete set of sporadic tasks generated from τ_s misses a deadline when scheduled by the non-preemptive EDF algorithm, then either condition (1) or condition (2) from Theorem 4.1 must have been violated. This proves the theorem. □

The following corollary shows that the EDF scheduling algorithm is universal for scheduling periodic tasks.

Corollary 4.4: Let τ_p be a set of periodic tasks $\{(c_1, p_1), (c_2, p_2), \ldots, (c_n, p_n)\}$ sorted in non-decreasing order by period. If τ_p satisfies conditions (1) and (2) from Theorem 4.1 then the non-preemptive EDF scheduling algorithm will schedule any concrete set of periodic tasks generated from τ_p.

Proof: Recall that one of the behaviors of a sporadic task is as a periodic task. Therefore, if conditions (1) and (2) are sufficient to guarantee the non-preemptive EDF algorithm will schedule a concrete set of sporadic tasks, then the conditions are also sufficient to guarantee the algorithm will schedule a concrete set of periodic tasks. □

Since the non-preemptive EDF algorithm is universal for both periodic and sporadic tasks, in order to decide if a set of tasks is schedulable, one need only consider if conditions (1) and (2) from Theorem 4.1 hold. Deciding if condition (1) holds is straightforward and can be performed in time $O(n)$. A set of tasks can be tested against condition (2) in pseudo-polynomial time $O(p_n)$ by using a dynamic programming technique [Jeffay 89a]. (Recall that p_n is the period of the "largest" task.)

5. Non-Preemptive Scheduling of Concrete Tasks

The non-preemptive EDF algorithm is universal for both periodic and sporadic tasks. In this section we examine the problem of scheduling a concrete set of periodic or sporadic tasks. Recall that a concrete task set consists of a task set together with release times of the tasks. For concrete sporadic tasks we show that the non-preemptive EDF scheduling algorithm is again universal. However, for concrete periodic tasks the situation is more complex. We show that the problem of deciding if a concrete set of periodic tasks is schedulable for any non-preemptive scheduling algorithm (including those that allow inserted idle time) is intractable (*i.e.*, NP-hard in the strong sense). Moreover, we show that if a universal algorithm exists for scheduling concrete periodic tasks without preemption then P = NP.

To begin, we consider scheduling concrete sporadic tasks. By the definition of schedulability, if a set of sporadic tasks τ_s is schedulable then any set of concrete sporadic tasks ω_s generated from τ_s is schedulable. The following theorem demonstrates that the schedulability of a concrete set of sporadic tasks is not a function of the assignment of release times to tasks.

Theorem 5.1: Let $\omega_s = \{(T_1,R_1), (T_2,R_2), \ldots, (T_n,R_n)\}$ be a concrete set of sporadic tasks generated from the set of sporadic tasks $\tau_s = \{T_1, T_2, \ldots, T_n\}$. Then ω_s is schedulable if and only if τ_s is schedulable.

Proof: ($\Rightarrow$) This follows immediately from the definition of schedulability. ($\Leftarrow$) We must show that if the tasks of τ_S can be scheduled so as to not miss any deadlines when the task release times are given by $R_1 \ldots R_n$, then the same is true for any other set of release times. Suppose this is

not the case, that is, for some set of release times $R'_1...R'_n$, there exists some pattern of task invocations for which some task of τ_s must miss a deadline. By the definition of the behavior of a sporadic task, an arbitrary time interval may elapse between a task's deadline and its next invocation. Let D be the maximum value of $R_i + p_i$, where p_i is the period of task T_i. Note that all initial invocations of tasks with release times $R_1...R_n$ are completed at or prior to D. We can now map the pattern of task invocations with release times of $R'_1...R'_n$ to a similar pattern of task invocations that begins at time D, in effect, starting τ_s over again with a set of "release times" $R'_i + D$ unrelated to the original release times. Clearly if some pattern of task invocations could force some task to miss a deadline for release times $R'_1...R'_n$, the same pattern of invocations shifted in time by D will cause some task of the concrete task set ω_s to miss a deadline sometime after D. But this contradicts the hypotheses that ω_s is schedulable and establishes the theorem. □

Theorem 5.1 shows that the problem of scheduling sporadic tasks is equivalent to the problem of scheduling concrete sporadic tasks. It follows that conditions (1) and (2) from Theorem 4.1 are necessary and sufficient for schedulability of concrete sporadic task sets. Moreover, the non-preemptive EDF scheduling algorithm is universal for these task sets.

Unlike concrete sporadic tasks, schedulability of concrete periodic tasks *is* a function of the assignment of release times. A periodic task set that is *not* schedulable may generate sets of concrete tasks that *are* schedulable as well as sets which are not (an example was given in Section 2). In order to properly study the problem of scheduling concrete periodic tasks, the definition of universality presented in Section 2 must be refined to include some notion of efficiency. It has been assumed that a scheduling algorithm can select a task to execute in zero time. Therefore, a scheduler that enumerated all possible schedules would be a universal, albeit uninteresting, scheduler. In addition to scheduling all schedulable sets of tasks, a reasonable requirement for a universal scheduling algorithm is that each scheduling decision be made in time polynomial in the number of tasks. For this refined notion of universality, we will show that if there exists a universal non-preemptive scheduling discipline for scheduling concrete periodic tasks then P = NP.

The following theorem shows that the complexity of deciding if a set of concrete periodic tasks is schedulable when one is allowed to consider any non-preemptive scheduling discipline (including those that allow inserted idle time) is NP-hard in the strong sense. This means that unless P = NP, a pseudo-polynomial time algorithm does not exist for deciding this question [Garey & Johnson 79]. This provides strong evidence that the problem is intractable. This decision problem can be formally stated as follows.

NON-PREEMPTIVE SCHEDULING OF CONCRETE PERIODIC TASKS (SCPT): Let $\tau_p = \{(c_1, p_1), (c_2, p_2), ..., (c_n, p_n)\}$ be a set of periodic tasks and let $\omega_p = (\tau_p, \rho)$ be a set of concrete periodic tasks generated from τ_p. Is it possible to schedule ω_p non-preemptively?

Theorem 5.2: NON-PREEMPTIVE SCHEDULING OF CONCRETE PERIODIC TASKS is NP-hard in the strong sense.

Proof: We will give a polynomial time transformation from the 3-PARTITION problem [Garey & Johnson 79] to SCPT.

An instance of the 3-PARTITION problem consists of a finite set A of $3m$ elements, a bound $B \in \mathbf{Z}^+$, and a "size" $s(a) \in \mathbf{Z}^+$ for each $a \in A$, such that each $s(a)$ satisfies $B/4 < s(a) < B/2$, and $\sum_{j=1}^{3m} s(a_j) = Bm$. The problem is to determine if A can be partitioned into m disjoint sets S_1, S_2, ..., S_m such that, for $1 \le i \le m$, $\sum_{a \in S_i} s(a) = B$. (With the above constraints on the element sizes, note that every S_i will contain exactly three elements from set A.)

The transformation is performed as follows. Let $A = \{a_1, a_2, a_3, ..., a_{3m}\}$, $B \in \mathbf{Z}^+$, and $s(a_1), s(a_2), s(a_3), ..., s(a_{3m}) \in \mathbf{Z}^+$, constitute an arbitrary instance of the 3-PARTITION problem. We create an instance of the SCPT problem by constructing a set ω_p of $n = 3m + 2$ concrete periodic tasks. Let $\tau_p = \{T_1, T_2, ..., T_{3m+2}\}$, where (recall $T = (cost, period)$)

$$\begin{aligned} & T_1 = ((8B, 20B), \\ & T_2 = ((23B, 40B), \text{ and} \\ \forall j,\ 3 \le j \le 3m+2: \quad & T_j = ((s(a_{j-2}), 40Bm), \end{aligned}$$

be a set of sporadic tasks, and let $\omega_p = \{(T_1,R_1), (T_2,R_2), ..., (T_{3m+2},R_{3m+2})\}$ where

$$\begin{aligned} & R_1 = 0, \\ & R_2 = 9B, \text{ and} \\ \forall j,\ 3 \le j \le 3m+2: \quad & R_j = 0, \end{aligned}$$

be a set of concrete sporadic tasks. The construction of the set ω_p can clearly be done in polynomial time with the largest number created in the new problem instance being $40Bm$. In this instance of SCPT, note that the processor utilization is

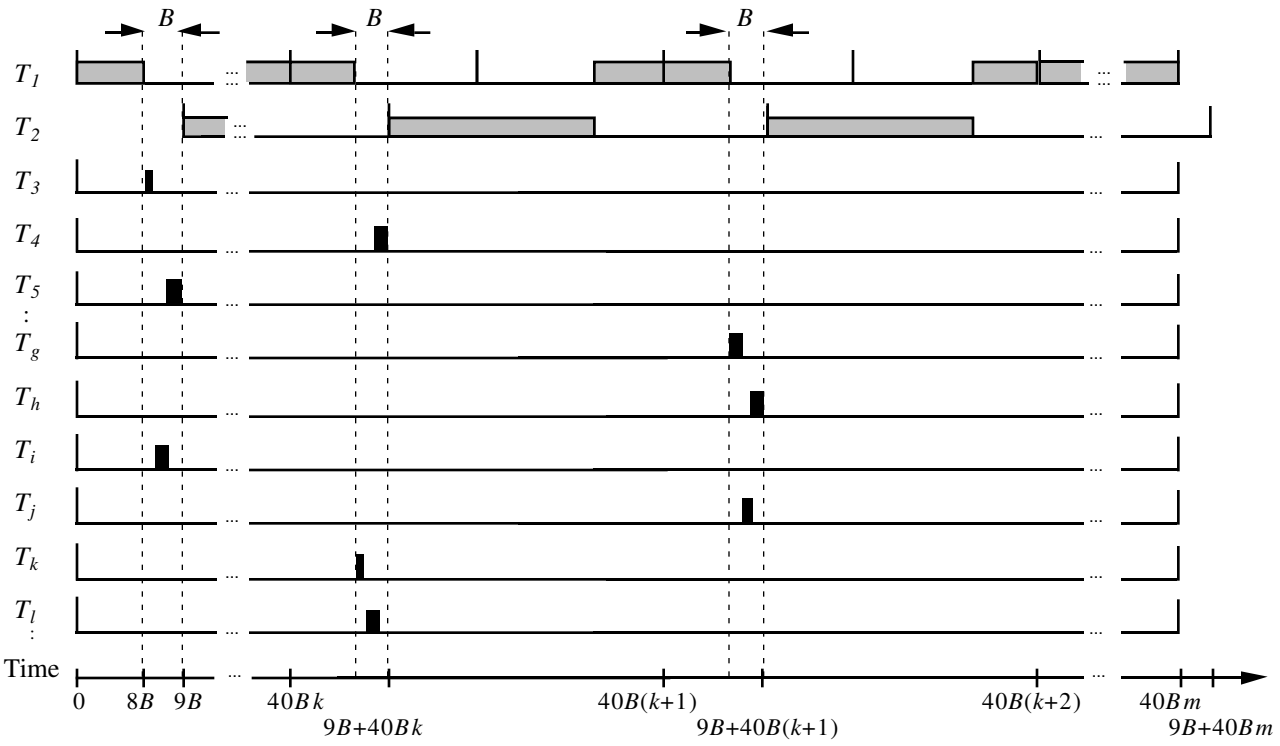

Figure 5.1

$$\sum_{j=1}^{n} \frac{c_j}{p_j} = \frac{8}{20} + \frac{23}{40} + \frac{\sum_{j=1}^{3m} s(a_j)}{40Bm} = \frac{39}{40} + \frac{Bm}{40Bm} = 1.0 .$$

By our choice of release times for T_1 and T_2, ω_p can be scheduled by a non-preemptive scheduling algorithm only if T_2 is scheduled at points in time $9B + 40Bk$, for all $k \geq 0$, and all the invocations of T_1 occurring at time $20B + 40Bk$, are scheduled at time $40B(k+1) - 8B$, for all $k \geq 0$. (See Figure 5.1.) This must be the case since if the execution of the i^{th} invocation of T_2 is scheduled at some time other than $9B + 40B(i-1)$, then the invocation of T_1 occurring at time $20B + 40B(i-1)$ will miss its deadline. Similarly, if the execution of an invocation of T_1 occurring at time $20B + 40Bk$, for some k, $k \geq 0$, is scheduled at some time other than at $40B(k+1) - 8B$, then the invocation of T_2 occurring at time $9B + 40Bk$ will miss its deadline.

Note that with these scheduling constraints, if we consider only tasks T_1 and T_2, then for all k, $k > 0$, in each interval $[40B(k-1), 40Bk]$, the processor will be idle for exactly B time units. It follows that in the interval $[0, 40Bm]$, there will be I disjoint idle periods, $m \leq I \leq 2m$, whose total duration is exactly Bm time units. For example, Figure 5.1 depicts a simulation of the scheduling of ω_p by the non-preemptive EDF algorithm. When EDF scheduling is used, in the interval $[0, 40Bm]$ there will be exactly m disjoint idle periods, each of duration B. In this case, ω_p will be schedulable if and only if the EDF algorithm can schedule tasks T_3 - T_{3m+2} in these m idle periods.

In the general case, ω_p will be schedulable by a non-preemptive scheduling algorithm if and only if there exists a partition of tasks T_3 - T_{3m+2} into m disjoint sets S_1, S_2, ..., S_m, such that for each set S_i, $\sum_{T_j \in S_i} c_j = B$. Therefore, a solution to SCPT can be used to solve an arbitrary instance of the 3-PARTITION problem by simply constructing the set of concrete periodic tasks ω_p, and then presenting these tasks to a decision procedure for SCPT. The answer from the SCPT decision procedure is the answer to the 3-PARTITION question for this problem instance. Since 3-PARTITION is known to be NP-complete in the strong sense [Garey & Johnson 79], SCPT is NP-hard in the strong sense. □

Note that the proof did not assume anything about the use of inserted idle time.

Although one cannot efficiently decide schedulability for concrete periodic tasks, recall that conditions (1) and (2) are sufficient for the EDF algorithm to schedule such

	Sets of Concrete Tasks	Sets of Tasks
Sporadic	Non-preemptive EDF	Non-preemptive EDF
Periodic	If a polynomial time algorithm exists, then P = NP	Non-preemptive EDF

Table 6.1: Universal scheduling algorithms.

	Sets of Concrete Tasks	Sets of Tasks
Sporadic	Pseudo-polynomial time	Pseudo-polynomial time
Periodic	NP-hard in the strong sense.	Pseudo-polynomial time

Table 6.2: Complexity of deciding schedulability.

tasks. (These conditions are, however, not necessary.)

The construction of ω_p in Theorem 5.2 can be used to show that if a universal non-preemptive scheduling algorithm existed for scheduling concrete periodic tasks, and this algorithm took only a polynomial amount of time (in the length of the input) to make each scheduling decision, then P = NP. That is, if there exists a universal non-preemptive scheduling algorithm for concrete periodic tasks (possibly using inserted idle time), then we can give a pseudo-polynomial time algorithm for deciding 3-PARTITION. The key observation is that if a 3-PARTITION problem instance is embedded in SCPT as described above, then only a pseudo-polynomial length portion of the schedule generated by a universal non-preemptive algorithm when scheduling ω_p, needs to be checked in order to decide the embedded 3-PARTITION problem instance.

Corollary 5.3: If there exists an universal, non-preemptive, uniprocessor scheduling algorithm for scheduling concrete periodic tasks then P = NP.

Proof: Assume there exists such a universal scheduling algorithm. From an instance of the 3-PARTITION problem, construct a set ω_p of concrete periodic tasks as described in the proof of Theorem 5.2. Note that if ω_p is not schedulable, then some task in ω_p will miss a deadline in the interval $[0, 9B+40Bm]$. Therefore we can simulate the universal scheduling algorithm on ω_p over the interval $[0, 9B+40Bm]$ and simply check to see if any tasks miss a deadline in this interval. The simulation and the checking of the schedule produced by the universal algorithm can clearly be performed in time proportional to Bm. By the reasoning employed in the proof of Theorem 5.2, if some task missed a deadline then there is a negative answer to the 3-PARTITION problem instance. If no task missed a deadline then there is an affirmative answer. Therefore, since 3-PARTITION is NP-complete in the strong sense and since we have given a pseudo-polynomial time algorithm for deciding 3-PARTITION, P = NP. □

Unless P = NP, Corollary 5.3 shows that we will not be able to develop a universal non-preemptive scheduling algorithm for scheduling concrete periodic tasks.

6. Summary

Non-preemptive scheduling problems arise in many forms in concurrent and real-time systems. Moreover, as non-preemptive schedulers are easier to implement and analyze (*e.g.*, assess the overhead of scheduling), it is important to understand the requirements of scheduling tasks non-preemptively. In this paper we have examined the problem of scheduling a set of periodic or sporadic tasks without preemption on a uniprocessor. The following fundamental results have been demonstrated. The *earliest deadline first* algorithm is universal for sets of sporadic and periodic tasks and for sets of concrete sporadic tasks. The universality is with respect to the class of scheduling algorithms that do not use inserted idle time. Unless P = NP, there does not exist a universal non-preemptive scheduling algorithm for concrete periodic tasks.

Given a set of sporadic, periodic, or concrete sporadic tasks, one can efficiently determine if the tasks will be schedulable. The problem of deciding schedulability for a set of concrete periodic tasks is intractable (NP-hard in the strong sense).

These results demonstrate that a fundamental distinction exists between periodic and sporadic tasking models. Specifically, the schedulability of a set of concrete sporadic tasks is not a function of their release times.

Our results are further summarized in the Tables 6.1 and 6.2.

7. Acknowledgements

We are indebted to Richard Anderson for suggesting the construction used in the proof of Theorem 5.2.

8. References

Bertossi, A.A., Bonuccelli, M.A. (1983). *Preemptive Scheduling of Periodic Jobs in Uniform Multiprocessor Systems*, **Information Processing Letters**, Vol. 16, No. 1, (January 1983), pp. 3-6.

Chung, J.C., Haris, M.R., Brooks, F.P., Fuchs, H., Kelley, M.T., Hughes, J., Ouh-young, M., Cheung, C., Holloway, R.L., Pique, M. (1989). *Exploring Virtual Worlds with Head-Mounted Displays*, Non-Holographic True 3-Dimensional Display Technologies, SPIE Proceedings, Vol. 1083, Los Angeles, CA, January 1989.

Dhall, S.K., Liu, C.L. (1978). *On a Real-Time Scheduling Problem*, **Operations Research**, Vol. 26, No. 1, (January 1978), pp. 127-140.

Frederickson, G.N. (1983). *Scheduling Unit-Time Tasks with Integer Release Times and Deadlines*, **Information Processing Letters**, Vol. 16, No. 4, (May 1983), pp. 171-173.

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

Garey, M.R., Johnson, D.S., Simons, B.B., and Tarjan, R.E. (1981). *Scheduling Unit-Time Tasks with Arbitrary Release Times and Deadlines*, **SIAM J. Computing**, Vol. 10, No. 2, (May 1981), pp. 256-269.

Jeffay, K. (1989). *The Real-Time Producer/Consumer Paradigm: Towards Verifiable Real-Time Computations*, Ph.D. Thesis, University of Washington, Department of Computer Science, Technical Report #89-09-15, September 1989.

Jeffay, K. (1989). *Analysis of a Synchronization and Scheduling Discipline for Real-Time Tasks with Preemption Constraints*, Proc. Tenth IEEE Real-Time Systems Symp., Santa Monica, CA, December 1989, pp. 295-305.

Jeffay, K. (1990). *Scheduling Sporadic Tasks With Shared Resources in Hard-Real-Time Systems,* University of North Carolina at Chapel Hill, Department of Computer Science, Technical Report TR90-038, August 1990. (Submitted for publication.)

Jeffay, K. (1991). *The Real-Time Producer/Consumer Paradigm: A paradigm for the construction of efficient, predictable real-time systems*, University of North Carolina at Chapel Hill, Department of Computer Science, April 1991. (Submitted for publication.)

Lawler, E.L., Martel, C.U. (1981). *Scheduling Periodically Occurring Tasks on Multiple Processors*, **Information Processing Letters**, Vol. 12, No. 1, (February 1981), pp.9-12.

Leung, J.Y.-T., Merrill, M.L. (1980). *A Note on Preemptive Scheduling of Periodic, Real-Time Tasks*, **Information Processing Letters**, Vol. 11, No. 3, (November 1980), pp.115-118.

Leung, J.Y.-T., Whitehead, J. (1982). *On the Complexity of Fixed Priority Scheduling of Periodic, Real-Time Tasks*, **Performance Evaluation**, Vol. 2, No. 4, (1982), pp.237-250.

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

Mok, A.K.-L. (1983). *Fundamental Design Problems of Distributed Systems for the Hard Real-Time Environment*, Ph.D. Thesis, MIT, Department of EE and CS, MIT/LCS/TR-297, May 1983.

Sorenson, P.G. (1974). *A Methodology for Real-Time System Development*, Ph.D. Thesis, University of Toronto, June 1974.

Sorenson, P.G., Hamacher, V.C. (1975). *A Real-Time Design Methodology*, **INFOR**, Vol. 13, No. 1, (February 1975), pp. 1-18.