Scheduling Sporadic Tasks with Shared Resources in Hard-Real-Time Systems

Kevin Jeffay[*]

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

Abstract

The problem of scheduling a set of sporadic tasks that share a set of serially reusable, single unit software resources on a single processor is considered. The correctness conditions are that (1) each invocation of each task completes execution at or before a well-defined deadline, and (2) a resource is never accessed by more than one task simultaneously. We present an optimal on-line algorithm for scheduling a set of sporadic tasks. The algorithm results from the integration of a synchronization scheme for access to shared resources with the *earliest deadline first* algorithm. A set of relations on task parameters that are necessary and sufficient for a set of tasks to be schedulable is also derived. Our model for the analysis of processor scheduling policies is novel in that it incorporates minimum as well as maximum processing time requirements of tasks. The scheduling algorithm and the sporadic tasking model have been incorporated into an operating system kernel and used to implement several real-time systems.

1. Introduction

Hard-real-time systems are commonly structured as a set of tasks that are invoked repetitively. Two frequently studied classes of repetitive tasks are *periodic* tasks, *i.e.*, tasks that are invoked at constant intervals [16], and *sporadic* tasks, *i.e.*, tasks that are invoked at random but with a minimum inter-invocation interval [17]. In both cases, each invocation of a task must complete execution before a well-defined deadline. Our contribution to the study of repetitive, real-time workloads is the consideration of tasks that share a set of serially reusable resources. Our notion of a resource is a software object, *e.g.*, a data structure, that is shared among a group of tasks and must be accessed in a mutually exclusive manner. Operations on a shared resource therefore constitute a critical section. For example, within the context of a concurrent programming language in which shared data is encapsulated within a monitor [14], a resource would be an individual monitor.

We consider a characterization of a hard-real-time system as a set of sporadic tasks that share a set of serially reusable software resources. This paper examines the problem of scheduling sporadic tasks that require exclusive access to a set of software resources. The problem is to sequence a set of sporadic tasks on a uniprocessor such that in all cases it is guaranteed that:

- each invocation of each task completes execution at or before its deadline, and
- a resource is never accessed by more than one task simultaneously.

Our work makes two contributions to the theory of real-time scheduling and resource allocation. The first is the development of an on-line algorithm for sequencing a set of sporadic tasks on a uniprocessor such that the above criteria are met. The algorithm results from the integration of a synchronization scheme for access to shared resources with the *earliest deadline first* algorithm of Liu and Layland; a preemptive, priority-driven scheduling algorithm with dynamic priority assignment [16]. The algorithm is optimal with respect to the class of scheduling policies that do not use inserted idle time [6]. The algorithm is optimal in the sense that it can schedule a set of tasks, without inserted idle time, whenever it will be possible to do so. The second contribution is a derivation of a set of relations on task parameters that are necessary and sufficient for a set of tasks to be schedulable. With these conditions one can efficiently decide whether it will be possible to schedule a set of tasks without executing or simulating the execution of the tasks. Our model for the analysis of processor scheduling policies is novel in that it incorporates minimum as well as maximum processing time requirements of tasks.

This work is part of a larger design system for hard-real-time systems. The on-line scheduling algorithm we develop has been implemented in the YARTOS operating system kernel [9,12] and the sporadic tasking model we present has been used to implement and analyze several fully functional real-time systems. These include a workstation-based conferencing system using digital audio and video [8], an interactive 3-dimensional graphics display system used for research in *virtual realities* [5], and a HiPPI data link controller [2].

Several approaches to scheduling real-time tasks that share resources have been described in the literature [3,4,10,15,17-21]. Most consider the case where tasks are periodic and develop heuristic algorithms for scheduling the tasks. The model we present is simpler than many previously considered, however, for this model we are able to

[*] Supported by a grant from the National Science Foundation (number CCR-9110938).

In: Proceedings of the 13th IEEE Real-Time Systems Symposium, Phoenix, AZ, December 1992, pp. 89-99.

establish fundamental optimality and complexity properties. Moreover, our experience in applying the model to actual systems indicates that it is powerful enough to build actual systems without undue effort [13]. We focus on the study of sporadic tasks for two reasons. First, our experiences indicate that in practice sporadic tasks more naturally capture the real-rime behaviors of time constrained computational processes. Second, when tasks have preemption constraints (*i.e.*, share resources) scheduling problems for periodic tasks are intractable. For example, Mok has shown that the problem of deciding if it is possible to schedule a set of periodic tasks that use semaphores to enforce mutual exclusion is NP-hard [17]. The general problem of deciding if it is possible to schedule a set of periodic tasks non-preemptively is NP-hard in the strong sense [11]. Moreover, if an optimal non-preemptive scheduling algorithm exists for periodic tasks, then P = NP.

The following section presents our model of a real-time system in greater detail and defines the objective of our study. Section 3 examines the problem of scheduling tasks that use only a single resource. An optimal algorithm is developed for this special case. Section 4 generalizes this algorithm for tasks that share a set of resources. Section 5 discusses an implementation of the scheduling policy and revisits the assumptions and restrictions in our model.

2. System Model

We define a hard-real-time system as a set of sporadic tasks that share a set of serially reusable, single unit software resources. A sporadic task is a sequential program that is invoked in response to the occurrence of an event. An event is a stimulus that may be generated by processes external to the system (*e.g.*, an interrupt from a device) or by processes internal to the system (*e.g.*, the arrival of a message). We assume events are generated repeatedly with a (non-zero) lower bound on the duration between consecutive occurrences of the same event. Therefore, each sporadic task will be invoked repeatedly with a lower bound on the interval between consecutive invocations. Sporadic tasks are well-suited for implementing computational processes that are required to execute periodically (with a constant interval between activations) or in response to recurring asynchronous events (with a minimum inter-arrival time). During the course of execution, a task may perform operations on shared data resources. Resources are serially reusable and must be accessed in a mutually exclusive manner. This model of software resources is motivated by the use of shared memory for efficient communication and synchronization between tasks.

Formally, a real-time system is a set of n sporadic tasks $\{T_1, T_2, ..., T_n\}$ and a set of m serially reusable, single unit resources $\{R_1, R_2, ..., R_m\}$. A task is described by a 3-tuple $T_i = (s_i, \{(c_{ij}, C_{ij}, r_{ij}) \mid 1 \le j \le n_i\}, p_i)$ where:

s_i — the release time of task T_i: the time of the first invocation of task T_i,

$\{ (c_{ij}, C_{ij}, r_{ij}) \}$ — a set of n_i *phases* where for each phase:

- c_{ij} the minimum computational cost: the minimum amount of processor time required to execute the j^{th} phase of task T_i to completion on a dedicated processor,
- C_{ij} the maximum computational cost: the maximum amount of processor time required to execute the j^{th} phase of task T_i to completion on a dedicated processor,
- r_{ij} the resource requirement: the resource (if any) that is required during the j^{th} phase of task T_i, and

p_i — the period of the task: the minimum time interval between invocations of task T_i.

The "period" of a sporadic task is simply the minimum time between any two successive invocations of the task [17]. In general an arbitrary amount of time may lapse between successive invocations of a task. Each task T_i is partitioned into a sequence of n_i disjoint phases. A phase is a contiguous sequence of statements that together require exclusive access to a resource. A task may have multiple phases that require the same resource. The resource required by T_i during the j^{th} phase of its computation is represented by an integer r_{ij}, $0 \le r_{ij} \le m$. If $r_{ij} = k$, $k \ne 0$, then the j^{th} phase of T_i's computation requires exclusive access to resource R_k. For a given invocation of T_i, in the interval between the time phase j commences execution and the time it completes execution, no other phase of a task that requires resource R_k may execute. If $r_{ij} = 0$, then the j^{th} phase of T_i's computation requires no resources. In this case the j^{th} phase of T_i imposes no mutual exclusion constraints on the execution of other tasks. Within the context of a concurrent programming language with monitors, if $r_{ij} \ne 0$, then the j^{th} phase of T_i would consist of a call to an entry procedure of a monitor that encapsulates resource r_{ij}. If $r_{ij} = 0$, then the j^{th} phase of T_i would consist of either code in the main body of the task or reentrant procedure code called by the main body of the task. Note that since different tasks may perform different operations on a resource (*e.g.*, call different monitor entry procedures), it is reasonable to assume that phases of tasks that access the same resource have varying computational costs. If a phase of a task requires a resource then the computational cost of the phase represents only the cost of using the required resource and not the cost (if any) of acquiring or releasing the resource. A minimum cost of zero indicates that a phase of a task is optional. A fundamental restriction is that each phase of each task will require access to at most one resource at a time. Other paradigms of resource usage and task decomposition will be discussed briefly in Section 5.

Throughout this paper we assume a discrete time model. In this domain all task parameters as well as all values of time are expressed as integer multiples of some indivisible time unit. Without loss of generality, assume these quantities are integers.

The behavior of a sporadic task is given by the following rules. Let t_k be the time of the k^{th} invocation of T_i.

i) The initial invocation of T_i occurs at time $t_1 = s_i$.

ii) If T_i has period p_i, then for all $k \geq 1$, the $(k+1)^{st}$ invocation of T_i occurs at $t_{k+1} \geq t_k + p_i \geq s_i + kp_i$.

iii) Each invocation of T_i consists of the execution of n_i phases in sequence. The execution of an invocation of T_i commences in phase 1. The j^{th} phase of each execution of T_i does not commence until the $(j-1)^{st}$ phase has terminated.

iv) Execution of the j^{th} phase of T_i requires at least c_{ij} units of processor time and at most C_{ij} units of processor time, $C_{ij} \geq c_{ij} \geq 0$.

v) The k^{th} invocation of T_i must be completed no later than time $t_k + p_i$. This time is commonly referred to as the *deadline* of the k^{th} invocation of T_i.

If the k^{th} invocation of task T_i occurs at time t, then the closed interval $[t, t+p_i]$ is called the *k^{th} invocation interval*, or simply an *invocation interval*, of task T_i. If task T_i is invoked at time t and does not complete execution at or before time $t + p_i$, then we say that T_i has *failed*. A set of sporadic tasks τ is said to be *feasible* on a uniprocessor if it is possible to schedule τ on a uniprocessor such that:

- no task fails, *i.e.*, every invocation of every task completes execution at or before the end of its invocation interval, and
- for each task T_i, and for all phases j, $1 \leq j \leq n_i$, if $r_{ij} \neq 0$, then the j^{th} phase of each invocation of T_i has exclusive access to the resource $R_{r_{ij}}$ from the time the phase commences execution until the phase terminates execution.

An algorithm *succeeds* in scheduling a set of tasks if it can sequence the tasks such that both criteria above will be met. A scheduling algorithm is said to be *optimal* if it can succeed for any task set that is feasible. Our goal is to develop an optimal uniprocessor scheduling algorithm. In doing so, we assume that in principle tasks are preemptable at arbitrary points. However, the requirement of exclusive access to resources places two restrictions on the preemption and execution of tasks. For all tasks i and k, if $r_{ij} = r_{kl}$ and $r_{ij}, r_{kl} \neq 0$, then (1) the j^{th} phase of task T_i may neither preempt the l^{th} phase of task T_k, nor (2) execute while the l^{th} phase of task T_k is preempted.

3. Single Phase Task Systems

We first consider the problem of scheduling sporadic tasks that consist of only a single phase. As will be shown in Section 4, the general problem of scheduling tasks with multiple phases can largely be reduced to the problem of scheduling tasks with only a single phase. The following sub-section establishes conditions that are necessary for a set of single phase sporadic tasks to be feasible in the absence of inserted idle time. Section 3.2 then develops an algorithm for scheduling such tasks and demonstrates its optimality.

3.1 Feasibility Conditions

Consider a set of single phase sporadic tasks $\{T_1, ..., T_n\}$, where $T_i = (s_i, (c_i, C_i, r_i), p_i)$,[1] that share a set of m serially reusable, single unit resources $\{R_1, ..., R_m\}$. It will be useful to refer to the period of the "shortest" task that uses resource R_i. For resource R_i, let P_i represent this period. That is, $P_i = \min_{1 \leq j \leq n}(p_j | r_j = i)$.

Our results rely on the fact that the feasibility of a set of sporadic tasks is not a function of their release times. If a set of sporadic tasks is feasible, then the tasks will be feasible for any combination of release times. A proof of the following can be found in [11].

Lemma 3.1: Let τ be a set of sporadic tasks. If τ is feasible then the set of sporadic tasks τ' obtained from τ by replacing the release times of tasks with arbitrary values will also be feasible.

The following theorem establishes necessary conditions for feasibility.

Theorem 3.2: Let τ be a set of single phase sporadic tasks $\{T_1, T_2, ..., T_n\}$, sorted in non-decreasing order by period (for all pairs of tasks T_i and T_j, if $i > j$, then $p_i \geq p_j$), that share a set of m serially reusable, single unit resources $\{R_1, R_2, ..., R_m\}$. If τ can be scheduled on a uniprocessor without inserted idle time, then:

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

$$2) \forall i, 1 < i \leq n \wedge r_i \neq 0, \forall L, P_{r_i} < L < p_i: \; L \geq C_i + \sum_{j=1}^{i-1} \left\lfloor \frac{L-1}{p_j} \right\rfloor C_j$$

Condition (1) can be viewed as a requirement that the processor not be overloaded (the cumulative processor utilization cannot exceed unity). 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 resource requesting task T_i is scheduled, and ending sometime before the end of the invocation interval. For a set of tasks to be feasible, the processor demand in this interval must always be less than or equal to the length of the interval. If this is not the case then a task can fail. Although condition (2) is semantically similar to the requirement that the processor not be over-utilized, we will demonstrate that conditions (1) and (2) are in fact not related. The intuition behind these conditions is developed further in the proof.

Proof: By Lemma 3.1, it suffices to show that there exist release times for which conditions (1) and (2) are necessary for τ to be feasible. We first show that (1) is necessary. For a set of tasks τ, the *achievable processor demand* in the time interval $[a, b]$, written $D_{a,b}$, is defined as the maximum amount of processing time required by τ in the interval $[a, b]$ to complete all invocations of tasks with deadlines in the interval $[a, b]$. That is, $D_{a,b}$ is the

[1] Since tasks consist of only a single phase, the second subscript on the parameters C, c, and r will be omitted.

processing time required, in the worst case, by τ in the interval $[a, b]$ to ensure that no task fails in the interval $[a, b]$. The worst case occurs when tasks are periodic from point a onward. If a set of tasks τ is feasible, then for all a and b, $a < b$, $D_{a,b} \le b - a$.

For all i, $1 \le i \le n$, let $s_i = 0$ and let $t = p_1 p_2 \ldots p_n$. In the interval $[0, t]$, tC_i/p_i is the maximum processor time that must be allocated to T_i to ensure that T_i does not fail in the interval $[0, t]$, hence $D_{0,t} = \sum_{i=1}^{n} \frac{t}{p_i} C_i = t \sum_{i=1}^{n} \frac{C_i}{p_i}$. If τ is feasible then it must be the case that $D_{0,t} \le t$, hence condition (1) must hold.

For condition (2) choose a task T_i, $1 < i \le n$, such that $r_i \ne 0$ (*i.e.*, T_i is a resource requesting task) and $p_i > P_{r_i}$ (*i.e.*, the period of T_i is greater than that of the smallest task that requests resource r_i). Let $s_i = 0$ and $s_j = 1$ for all j, $1 \le j \le n$, $j \ne i$. This gives rise to the pattern of initial task invocations shown in Figure 3.1. Initially only T_i is eligible for execution. Since inserted idle time is not allowed, T_i must execute in the interval [0,1]. For all L, $L > P_{r_i}$, the interval $[0, L]$ contains at least one invocation of some task T_k with $r_k = r_i$. Since T_k shares a resource with T_i and since this resource is in use by T_i at time 1, the initial invocation of T_k may not be scheduled until after the invocation of T_i made at time 0 has completed execution. Therefore, to ensure that the initial invocation of T_k does not fail, the initial invocation of T_i must be completed before time $p_k + 1 \le P_{r_i} + 1$. Hence for this choice of release times, for all L, $P_{r_i} < L < p_i$, in the interval $[0, L]$ the achievable processor demand is $D_{0,L} = C_i + \sum_{j=1}^{i-1} \left\lfloor \frac{L-1}{p_j} \right\rfloor C_j$

The demand consists of the maximum cost of executing the initial invocation of T_i plus the achievable 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 invocation intervals contained in the interval $[1, L]$ and hence can not fail in the interval $[1, L]$. Therefore they do not contribute to the achievable processor demand in the interval $[1, L]$.) For τ to be feasible it must be the case that $L \ge D_{0,L}$, hence condition (2) must hold. □

The constructions in the proof of Theorem 3.2 characterize the worst case inter-leavings of task invocations for a set of sporadic tasks. In essence, it will be shown in Section 3.2 that if a set of tasks can be scheduled when invoked as in Figure 3.1, then the tasks are feasible. The notion of a worst case interleaving is important as Lemma 3.1 indicates that such an interleaving can always occur during the execution of any task set.

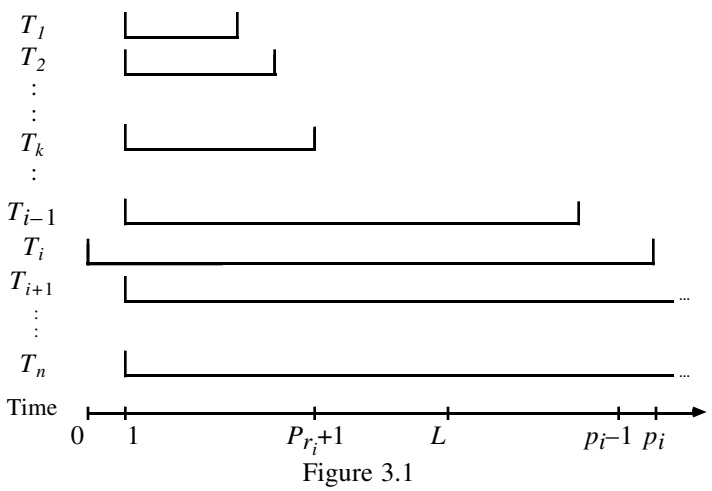

Figure 3.1

Some special cases of Theorem 3.2 are worth noting. A set of single phase sporadic tasks τ where $r_i = 0$, for $1 \le i \le n$, corresponds to a set of tasks with no resources and hence no mutual exclusion constraints. In such a system a task would, in principle, be preemptable at any time during its execution by any other task. In this case condition (2) is void (the quantification of i is empty) and condition (1) alone is necessary for feasibility. This agrees with results reported for the preemptive scheduling of periodic tasks (a special case of sporadic tasks) [16]. Similarly, if tasks require resources but the resources are not shared (*i.e.*, only one task requires each resource) then condition (2) is again void (the quantification of L is empty for all tasks). At the other extreme, a set of single phase sporadic tasks in which for all i, $1 \le i \le n$, $r_i = k$, for some $k \ne 0$, corresponds to a set of tasks that all share a single resource. Such single phase tasks must be scheduled non-preemptively. In this case condition (2) applies to all tasks and the feasibility conditions agree with those reported in [11] for the non-preemptive scheduling of sporadic tasks.

3.2 Scheduling Single Phase Task Systems

We seek an algorithm that will sequence a set of single phase sporadic tasks on a single processor whenever it is possible to do so. Such an algorithm must ensure that (1) all task invocations complete execution before their deadline and that (2) the mutual exclusion constraints on the execution of resource requesting tasks are respected. It is the latter requirement that motivates the development of a new scheduling policy. Our approach is to incorporate a synchronization protocol for mutual exclusion into an existing real-time scheduling policy. The basis of our scheduling policy is the preemptive *earliest deadline first* (EDF) algorithm [16]. Our choice of an EDF policy is motivated by the fact that it is an optimal policy both when tasks have no execution constraints [16] and when preemption is not allowed [11]. The problem currently under consideration lies between these two extremes. We begin with some definitions.

When a task is invoked, if the resource the task requires is in use by another task, then the requesting task is said to be *blocked*; otherwise the task is said to be *ready*. When an invocation of a task is executing on a

processor, the task is *executing*. If a task is preempted while executing then it returns to the ready state. After completion of an invocation, and prior to the first invocation, a task is *terminated*. The EDF scheduling discipline dictates that at all points in time, the ready task with the nearest deadline should be executing. An EDF scheduler makes scheduling decisions (dispatches tasks) whenever a task is invoked or terminates. At each of these scheduling points an EDF scheduler dispatches the ready task with the nearest deadline; preempting the previously executing task if necessary. Ties between tasks with identical deadlines are broken arbitrarily. The EDF scheduling discipline can be extended to ensure exclusive access to shared resources by re-examining the concept of an execution deadline. If tasks share resources then when a resource requesting task T_i is invoked, it is no longer sufficient for the invocation to complete execution within p_i time units. It can be the case that a resource requesting task must complete execution *before* the end of its current invocation interval. This situation can occur when an invocation of a task with a deadline becomes blocked. For example, consider the problem of scheduling the following task set according to a naive application of the traditional preemptive EDF discipline:

$$\tau = \{\ T_1 = (2, (1,1,1), 4),\ T_2 = (1, (3,3,0), 10),$$
$$T_3 = (0, (3,3,1), 20)\ \}.$$

τ consists of three single phase tasks and one shared resource (R_1). The initial interleaving of invocations of these tasks is illustrated in Figure 3.2. Since inserted idle time is not allowed, task T_3 will be scheduled at time 0 as shown at the top of Figure 3.2. At time 1, task T_2 has a nearer deadline than the executing task T_3. Since $r_2 \neq r_3$, task T_2 may preempt task T_3 and hence an EDF scheduler might preempt the execution of T_3 at time 1 in favor of task T_2. At time 2 task T_1 is invoked and has the nearest deadline. However, since T_1 requires the resource that is in use by task T_3, T_1 is blocked by T_3 and hence T_2 continues execution at time 2. At time 3, task T_2 completes execution and task T_3 resumes execution (since task T_1 is stilled blocked by task T_3). This scenario causes task T_1 to eventually fail at time 6. This failure is due to the fact that at time 2, it is no longer sufficient for the invocation of task T_3 occurring at time 0 to be completed by its nominal deadline at time 20. Since tasks T_1 and T_3 share a resource, when task T_1 is invoked at time 2, the invocation of task T_3 occurring at time 0 must now be completed no later than time 6; the initial deadline of task T_1. (Of course the initial invocation of task T_3 must actually be completed by time $6 - C_1 = 5$. It will turn out, however, that this is not a useful observation.)

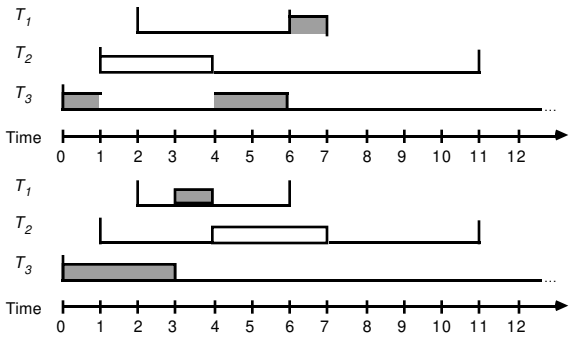

Figure 3.2

The challenge is to quantify precisely *when* a task invocation must be completed. We claim that an invocation of a resource requesting task should have *two* notions of a deadline: one for the initial acquisition of the processor, and one for subsequent execution. Specifically, when a resource requesting task T_i is invoked at time t_r, the invocation should have an initial deadline equal to $t_r + p_i$ as in traditional EDF scheduling. This deadline will be referred to as the *initial* or *contending* deadline. Let t_s be the time that the invocation of task T_i occurring at time t_r is first scheduled (commences execution). We claim that after time t_s, the invocation of task T_i should have a deadline at time $\text{MIN}(t_r + p_i, (t_s + 1) + P_{r_i})$. Thus, when a scheduler first dispatches an invocation of task T_i, the scheduler will potentially assign T_i a nearer deadline. This deadline will be referred to as the *execution* deadline. Since we assume a discrete time domain, a resource requesting task T_i has a contending deadline at all points in time in the closed interval $[t_r, t_s]$ and, assuming $C_i > 1$, has an execution deadline at all points in the closed interval $[t_s+1, t_c-1]$, where t_c is the time that the execution of the invocation terminates. (In the interval between the completion of one invocation and the start of the next, a task logically has a deadline of infinity.) This is illustrated in Figure 3.3 which plots the deadline of an invocation of a task T_i that has an execution deadline of $(t_s + 1) + P_{r_i}$ as a function of time. If a resource consuming task has a maximum computational cost of 1, then it will never have an execution deadline. Non-resource requesting tasks require no special treatment. If a non-resource requesting task T_j is invoked at time t_r, the invocation will have a deadline at time $t_r + p_j$ for the duration of its execution. We will refer to our scheme of dynamically altering the deadlines of resource requesting tasks as the *dynamic deadline modification* (DDM) strategy.

The application of the dynamic deadline modification strategy to the tasks in the previous example results in the non-preemptive schedule illustrated at the bottom of Figure 3.2. Under this policy the initial invocation of task T_3 has a contending deadline at time 20 as before. However, once task T_3 is scheduled it will execute with a deadline equal to $\text{MIN}(t_r + p_i, (t_s + 1) + P_{r_i}) = \text{MIN}(0 + 20, 0 + 1 + 4) = 5$. That is, at times 1 and 2, task T_3 has a deadline at time 5. When task T_2 is invoked at time 1, its invocation will have an initial deadline at time $1 + p_2 = 11$. At time 1, T_3 has a nearer deadline than T_2 and hence an EDF scheduler will not allow T_2 to preempt T_3 at time 1.

The imposition of separate deadlines for execution and initial acquisition of the processor ensures that blocked

tasks become unblocked as soon as possible. Although an invocation of a resource requesting task may now execute with a deadline that occurs before the end of the invocation interval, this "deadline" is indeed a deadline. We will show that a task can fail if an invocation of a resource requesting task does not complete by its execution deadline.

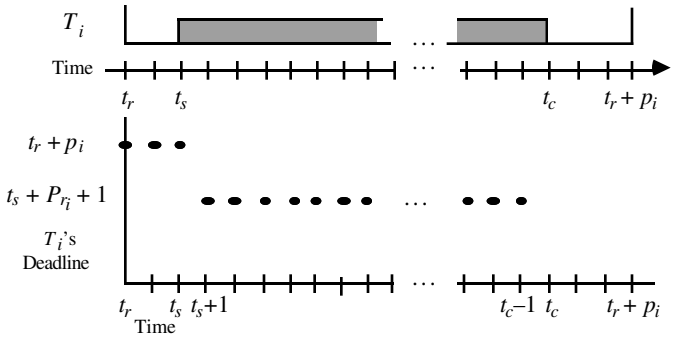

Figure 3.3

A final point to address is the mutual exclusion constraints on access to resources. The combination of EDF scheduling with the dynamic deadline modification strategy is sufficient for ensuring mutually exclusion. There is, however, one subtlety in the case that there exist multiple outstanding invocations with the earliest deadline. To guarantee that the mutual exclusion constraints are respected, when there exist multiple tasks with outstanding invocations with the earliest deadline, a scheduler must (1) allow the currently executing task to continue execution if it has the earliest deadline, and (2) select a task with an outstanding invocation that has been preempted before selecting any task whose outstanding invocation has not begun execution. The combination of an EDF task selection rule with dynamic deadline modification and tie breaking rules will be called *earliest deadline first* scheduling with *dynamic deadline modification* (EDF/DDM). The EDF/DDM scheduling policy is validated by demonstrating that it is an optimal discipline (with respect to the class of disciplines that do not use inserted idle time) for scheduling a set of single phase tasks that share a set of resources. To prove optimality it suffices to show that the satisfaction of conditions (1) and (2) from Theorem 3.2 is sufficient for ensuring that the EDF/DDM discipline will succeed in scheduling a set of tasks with shared resources. To demonstrate that the discipline succeeds in scheduling a set of tasks it must be shown that (1) all invocations of all tasks complete execution before the end of their respective invocation intervals and that (2) the mutual exclusion constraints on the execution of resource requesting tasks are respected. The following lemma demonstrates that the EDF/DDM scheduling discipline enforces the mutual exclusion constraints on access to resources.

Lemma 3.3: The EDF/DDM scheduling discipline satisfies the preemption constraints on access to resources.

Proof: It suffices to show that a task that requires resource R_j can neither preempt another task that requires R_j nor execute while such a task is preempted when scheduled by the EDF/DDM scheduling discipline.[2] Let T_i be a R_j requesting task. Let t_s be a point in time at which an invocation of T_i commences execution. Let $t > t_s$ be a point in time at which this invocation is either executing or is preempted. Let T_k be a R_j requesting task with an invocation that is contending for the processor at time t. Let t_r be the time at which this invocation by T_k was made. Under a EDF/DDM scheduling discipline, in order for T_k to preempt T_i or to execute while T_i is preempted, it must be the case that $t_s < t_r \leq t$ (and $t_r + p_k < t_s + p_i$). The invocation of T_k occurring at t_r will have an initial deadline at time $d_k = t_r + p_k$. Since T_i is scheduled at t_s, its invocation must have a deadline no later than at $d_i = t_s + P_j + 1 \leq t_s + p_k + 1$. Since $t_s < t$, it follows that $d_i \leq d_k$. If $d_i < d_k$, then the invocation of T_k occurring at t_r will not be scheduled until after the invocation of T_i occurring at t_s has completed execution. If $d_i = d_k$, then since the EDF/DDM scheduling discipline gives priority to the currently executing task and then to preempted tasks, T_k will again not be scheduled until after the outstanding invocation of T_i has completed execution. Therefore, a task that requires R_j can neither preempt another R_j requesting task nor execute while such a task is preempted. □

Theorem 3.4: Let τ be a set of single phase sporadic tasks $\{T_1, T_2, ..., T_n\}$, sorted in non-decreasing order by period, that share a set of serially reusable, single unit resources $\{R_1, R_2, ..., R_m\}$. The EDF/DDM discipline will succeed in scheduling τ if conditions (1) and (2) from Theorem 3.2 hold.

Proof: Lemma 3.3 has shown that the EDF/DDM scheduling discipline always maintains the mutual exclusion constraints on access to resources. It remains to show that the use of the EDF/DDM scheduling discipline guarantees that tasks will not fail if conditions (1) and (2) of Theorem 3.2 hold. This will be shown by contradiction.

Assume the contrary, *i.e.*, that conditions (1) and (2) of Theorem 3.2 hold and yet a task fails when τ is scheduled by the EDF/DDM algorithm. For a set of tasks τ, define the *actual processor demand*, or simply the *processor demand*, in the interval $[a, b]$, written $d_{a,b}$, as the least upper bound on the amount of processing time actually required by τ in the time interval $[a, b]$ to ensure that no task fails in $[a, b]$. If a set of tasks τ is feasible, then for all a and b, $a < b$, it follows that $d_{a,b} \leq D_{a,b} \leq b - a$. The proof proceeds by deriving upper bounds on the actual processor demand (*i.e.*, the achievable processor demand) for an interval ending at the time at which a task fails.

Let t_d be the earliest point in time at which a task fails. τ can be partitioned into three disjoint subsets A_1, A_2, and A_3, where

[2] Note that the first tie breaking rule ensures that there can exist only one preempted task with the earliest deadline.

A_1 = the set of tasks that have an invocation with an initial deadline at time t_d,

A_2 = the set of tasks that have an invocation occurring prior to time t_d with initial deadline after t_d, and

A_3 = the set of tasks not in A_1 or A_2.

Tasks in A_3 either have a release time greater than t_d, or are not invoked immediately prior to time t_d. To bound the actual processor demand prior to t_d, it suffices to concentrate on the tasks in A_2. Let $b_1, b_2, ..., b_k$ be the invocation times immediately prior to t_d of the tasks in A_2. There are two main cases to consider.

Case 1: None of the invocations of tasks in A_2 occurring at times $b_1, ..., b_k$ are scheduled prior to time t_d.

Let t_0 be the end of the last period 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 actual processor demand is the total processing requirement of tasks that are invoked at or after t_0, with deadlines at or before t_d. This gives

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

Since there is no idle period in $[t_0, t_d]$ and since a task fails at t_d, it must be the case that $d_{t_0,t_d} > t_d - t_0$. Therefore

$$t_d - t_0 < \sum_{j=1}^{n} \left\lfloor \frac{t_d - t_0}{p_j} \right\rfloor C_j \le \sum_{j=1}^{n} \frac{t_d - t_0}{p_j} C_j = (t_d - t_0) \sum_{j=1}^{n} \frac{C_j}{p_j} .$$

This is a contradiction of condition (1). Therefore, if conditions (1) and (2) hold and the EDF/DDM scheduling discipline fails to schedule τ, then an invocation of at least one task in A_2 must have been scheduled prior to t_d.

Case 2: Some of the invocations of tasks in A_2 occurring at times $b_1, ..., b_k$ are scheduled prior to time t_d.

Let T_i be the last task in A_2 to execute prior to t_d. Let t_i be the point in time at which the invocation of T_i occurring immediately prior to t_d is scheduled for the first time. Note that because of deadline-based scheduling, if a task T_k fails at t_d then $t_i < t_d - p_k$. We show that if the invocation interval of T_i containing the point t_d is scheduled prior to t_d, then there must have existed enough processor time in $[t_i, t_d]$ to schedule all invocations of tasks occurring after t_i with deadlines at or before t_d. There are two sub-cases to consider depending on whether or not the invocation of T_i scheduled at t_i has an execution deadline less than or equal to t_d. If this is the case then this invocation of T_i must be completed at or before t_d.

Case 2a: The invocation of task T_i scheduled at time t_i has an execution deadline less than or equal to time t_d.

In this case, since T_i is in A_2, T_i must be a resource requesting task. We proceed by deriving the achievable processor demand for the interval $[t_i, t_d]$. If a task fails at time t_d then the following facts hold for Case 2a:

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 an invocation of T_i is scheduled at t_i and has an execution deadline less than or equal to t_d, every other task scheduled in $[t_i, t_d]$ must have had an initial deadline at or before t_d. Therefore, if an invocation of a task T_j, with period greater than or equal to $t_d - t_i$, executes in the interval $[t_i, t_d]$, then this invocation of T_j must have been available for execution at t_i. Consequently, since the invocation of T_i in question had an initial deadline greater than t_d, the EDF/DDM algorithm would have chosen T_j before T_i in the interval $[t_i, t_d]$. Therefore, no task with period greater than or equal to $t_d - t_i$ executes in the interval $[t_i, t_d]$.

ii) Other than task T_i, no task which executes in $[t_i, t_d]$ could have been invoked at time t_i.

Again, other than T_i, every task that executes in $[t_i, t_d]$ has an initial deadline at or before t_d. If a task $T_{i'}$ that executes in $[t_i, t_d]$ had been invoked at t_i, the EDF/DDM algorithm would have scheduled $T_{i'}$ instead of T_i at time t_i.

iii) The processor is fully utilized in the interval $[t_i, t_d]$.

If the processor is ever idle in the interval $[t_i, t_d]$, then the analysis of Case 1 can be applied to the interval $[t_0, t_d]$ (where $t_0 > t_i + C_i$ is the end of the last idle period prior to t_d) to reach a contradiction of condition (1).

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

$$d_{t_i,t_d} \le D_{t_0,t_d} = C_i + \sum_{j=1}^{i-1} \left\lfloor \frac{t_d - (t_i + 1)}{p_j} \right\rfloor C_j .$$

Let $L = t_d - t_i$. Since the invocation of T_i scheduled at t_i has an execution deadline less than or equal to t_d, it follows that $(t_i + 1) + P_{r_i} \le t_d$. Hence $t_d - (t_i + 1) \ge P_{r_i}$, $t_d - t_i > P_{r_i}$, $p_i > t_d - t_i > P_{r_i}$, $p_i > L > P_{r_i}$. Since (*iii*) indicates that there is no idle time in $[t_i, t_d]$, and since a task failed at t_d, it follows that $d_{t_i,t_d} > t_d - t_i$ and hence $d_{t_i,t_d} > L$. Combining this with the inequality above yields

$$L < C_i + \sum_{j=1}^{i-1} \left\lfloor \frac{L-1}{p_j} \right\rfloor C_j .$$

This contradicts the assumption that condition (2) was true.

Case 2b: The invocation of task T_i scheduled at time t_i has an execution deadline greater than time t_d.

This will be the case if T_i is either a non-resource requesting task ($r_i = 0$), or if $(t_i + 1) + P_{r_i} > t_d$. The significance of this case is that the invocation of T_i scheduled at t_i has a deadline after t_d, and hence may be preempted by *any* task with an invocation interval contained in $[t_i, t_d]$. This is because, since $t_d - t_i \le P_{r_i}$, T_i can not share a resource with any task that can possibly have an invocation interval contained in $[t_i, t_d]$. Let $t_0 > t_i$ be the later of the end of the last idle period in $[t_i, t_d]$ or the time T_i last stops execution prior to t_d. Since the invocation of T_i scheduled at t_i has a deadline greater than t_d and since T_i is preemptable by any task that executes in $[t_i, t_d]$, all invocations of tasks occurring prior to t_0 with deadlines less than or equal to t_d must have completed execution by t_0. The analysis of Case 1 can be applied directly to $[t_0, t_d]$ to reach a contradiction of condition (1).

This concludes Case 2. We have shown that in all cases, if the EDF/DDM scheduling discipline fails, then either condition (1) or condition (2) from Theorem 3.2 must have been violated. This proves the theorem. □

Corollary 3.5: With respect to the class of algorithms that do not use inserted idle time, the EDF/DDM discipline is optimal for scheduling a set of sporadic tasks that share a set of serially reusable, single unit resources.

Proof: The proof follows immediately from Theorems 3.2 and 3.4. □

4. Multiple Phase Task Systems

We next demonstrate how the EDF/DDM algorithm can be extended to schedule multiple phase sporadic tasks that share a set of resources. The extension is straightforward and preserves the optimality of the EDF/DDM discipline. Due to space limitations, the proofs in this section are abbreviated. Complete proofs are available from the author.

4.1 Feasibility Conditions

The following gives necessary conditions for scheduling multiple phase tasks.

Theorem 4.1: Let $\tau = \{T_i = (s_i, \{(c_{ij}, C_{ij}, r_{ij}) \mid 1 \le j \le n_i\}, p_i) \mid 1 \le i \le n\}$ be a set of multiple phase sporadic tasks sorted in non-decreasing order by period, that share a set of serially reusable, single unit resources $\{R_1, ..., R_m\}$. If τ can be scheduled without inserted idle time, then:

1) $\sum_{i=1}^{n} \frac{E_i}{p_i} \le 1$

2) $\forall i, 1 < i \le n, \forall k, 1 \le k \le n_i \wedge r_{ik} \ne 0, \forall L, P_{r_{ik}} < L < p_i - S_{ik}$:

$$L \ge C_{ik} + \sum_{j=1}^{i-1} \left\lfloor \frac{L-1}{p_j} \right\rfloor E_j$$

where:

- $P_{r_{ik}} = \min_{1 \le j \le n} (p_j \mid \exists l, 1 \le l \le n_j : r_{jl} = r_{ik})$,
- $E_j = \sum_{l=1}^{n_j} C_{jl}$, and
- $S_{ik} = \begin{cases} 0 & \text{if } k = 1 \\ \sum_{j=1}^{k-1} c_{ij} & \text{if } 1 < k \le n_i \end{cases}$

The feasibility conditions are similar to those for single phase tasks. The parameter E_i represents the maximum cost of an invocation of task T_i and replaces the C_i term in condition (1). Condition (2) now applies to only a resource requesting phase of task T_i rather than to the task as a whole. Because of this, the range of L in condition (2) is more restricted than in the single phase case. The range is more restricted since the k^{th} phase of a task T_i cannot start until all previous phases have terminated, and thus the earliest time phase k can be scheduled is S_{ik} time units after the start of an invocation of T_i. For the k^{th} phase of a task, the range of intervals of length L in which one must compute the achievable processor demand will be shorter than in the single phase case by the sum of the minimum costs of phases 1 through k–1. Also note that no demand due to phases of T_i other than k appear in (2). In the event that each task in τ consists of only a single phase, conditions (1) and (2) reduce to the conditions of Theorem 3.2.

Proof: By Lemma 3.1, it suffices to demonstrate the existence of release times for which conditions (1) and (2) are necessary for feasibility. The construction for the necessity of condition (1) is identical to the one used in the proof of Theorem 3.2. For (2) choose a task T_i, $1<i\le n$, and choose a phase k of T_i, $1\le k\le n_i$, such that $r_{ik}\ne 0$, and $P_{r_{ik}}<p_i$. Let $s_i=0$ and $s_j=S_{ik}+1$ for all j, $1\le j\le n$, $j\ne i$. This gives the pattern of task invocations shown in Figure 4.1.

For all L, $L>P_{r_{ik}}$, the interval $[S_{ik}, S_{ik}+L]$ contains at least one invocation of a task that requires resource r_{ik}. If τ is to be feasible then, in the worst case, the computation of task T_i started at time 0 must have its k^{th} phase completed in $[S_{ik}, S_{ik}+L]$. Thus for all L, $P_{r_{ik}} < L < p_i - S_{ik}$, in $[S_{ik}, S_{ik}+L]$, the achievable processor demand, is

$$D_{S_{ik}, S_{ik}+L} \ge C_{ik} + \sum_{j=1}^{i-1} \left\lfloor \frac{L-1}{p_j} \right\rfloor E_j$$

In general it is not necessary for phases of T_i beyond phase k to execute in $[0, L]$ in order to ensure that a task does not fail in $[0, L]$. For τ to be feasible it must be case that $L \ge D_{S_{ik}, S_{ik}+L}$, hence condition (2) must hold. □

4.2 Scheduling Multiple Phase Task Systems

The EDF/DDM scheduling discipline was defined for single phase tasks. It can be extended to handle tasks with multiple phases, by viewing a multiple phase task $T_i = (s_i, \{(c_{ij}, C_{ij}, r_{ij}) \mid 1 \le j \le n_i\}, p_i)$, as set of n_i single phase tasks $\{T_{ij} = (s_i, (c_{ij}, C_{ij}, r_{ij}), p_i) \mid 1 \le j \le n_i\}$. For a given value of i, all tasks in $\{T_{ij}\}$ conceptually are invoked simultaneously and are scheduled such that the k^{th} invocation of T_{ij}, $1 < j \le n_i$, is not scheduled until the k^{th} invocation of T_{ij-1} has terminated. (Note that for a given value

of i, since all tasks in $\{T_{ij}\}$ are invoked simultaneously, outstanding invocations of tasks T_{ij} will always have the same deadline. Therefore, the EDF/DDM scheduling discipline can be made to enforce the precedence constraints on the execution of these single phase tasks by further biasing its algorithm for selecting a task for execution when there exist more than one ready task with the earliest deadline.) It should be clear that the execution of the set of single phase tasks $\{T_{ij}\}$ defined above will be equivalent to the execution of a multiple phase task T_i. This motivates the treatment of each phase of a multiple phase task as a logical single phase task. Specifically, each resource requesting phase of a multiple phase task should have both a contending and an execution deadline.

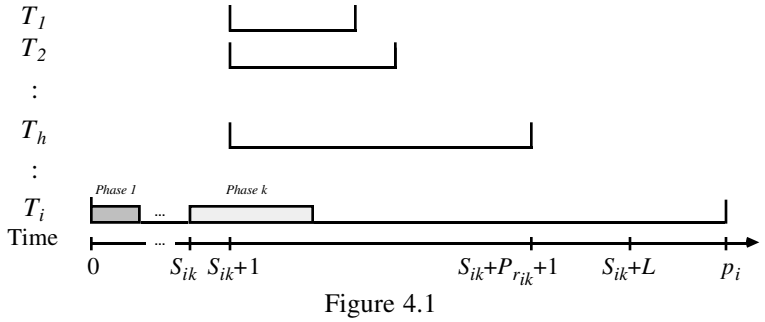

Figure 4.1

Let t_r be a point in time at which a multiple phase task T_i is invoked. For this invocation let t_{sk} be the time the k^{th} phase of T_i is first scheduled and let t_{ck} be the time this phase terminates. In the interval $[t_r, t_{s1}]$, T_i will have a *contending* deadline equal to $t_r + p_i$ as in traditional EDF scheduling. For all k, $1 \le k \le n_i$, if $r_{ik} \neq 0$ and $C_{ik} > 1$, in the interval $[t_{sk}+1, t_{ck}-1]$, T_i will have an execution deadline equal to $\text{MIN}(t_r + p_i, (t_{sk} + 1) + P_{r_{ik}})$. Between phases T_i will again contend for the processor. At the time of the completion of each phase the deadline of T_k will revert to the initial deadline for this invocation. Hence for all k, $1 \le k < n_i$, in the interval $[t_{ck}, t_{s(k+1)}]$, task T_i will have a deadline of $t_r + p_i$. Figure 4.2 illustrates how a multiple phase task's deadline changes dynamically throughout an invocation interval. It shows an execution of a multiple phase task $T_i = (s_i, \{(3,3,r_{i1}), (3,3,r_{i2}), (10,10,r_{i3})\}, p_i)$. Each phase has an execution deadline different from its contending deadline.

The extended version of the EDF/DDM scheduling discipline will be called the *generalized EDF/DDM* discipline. Its design is again validated by proving it is an optimal policy for scheduling a set of multiple phase tasks. To prove optimality it suffices to show that the satisfaction of the conditions of Theorem 4.1 are sufficient for ensuring the generalized EDF/DDM discipline will succeed in scheduling a set of multiple phase tasks with shared resources. The following lemma demonstrates that the EDF/DDM discipline respects the mutual exclusion constraints on access to resources.

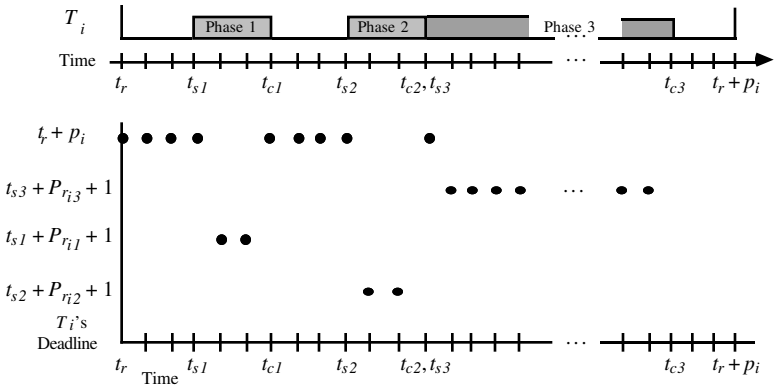

Figure 4.2

Lemma 4.2: The generalized EDF/DDM discipline satisfies the preemption constraints on access to resources.

The proof is largely identical to that of Lemma 3.3.

Theorem 4.3: Let τ be a set of multiple phase sporadic tasks as in Theorem 4.1. The generalized EDF/DDM discipline will succeed in scheduling τ if conditions (1) and (2) of Theorem 4.1 hold.

The proof is similar to the proof of Theorem 3.4.

Theorem 4.4: With respect to the class of scheduling algorithms that do not use inserted idle time, the generalized EDF/DDM discipline is an optimal discipline for scheduling a set of multiple phase sporadic tasks that share a set of serially reusable, single unit resources.

Proof: The proof follows from Theorems 4.1 and 4.3. □

Conditions (1) and (2) of Theorem 4.1 can be used as the basis of a decision procedure for deciding the feasibility of a set of sporadic tasks that share a set of resources. By Theorems 4.3 and 4.4, a set of tasks will be feasible if and only if they satisfy conditions (1) and (2). Deciding if condition (1) holds is straightforward and can be performed in time linear in the number of inputs. Condition (2) can be tested in time $O(mp_n)$ where m is the number of resources and p_n is the largest period of any task.

5. Discussion

This section discusses an implementation of our model, revisits some of the assumptions and restrictions present in the model of Section 2.

5.1 Implementation Considerations

The sporadic tasking model and EDF/DDM scheduling policy have been implemented in the YARTOS operating system kernel [9,12]. The implementation is unique in that all tasks share a single run-time stack. The use of a single stack greatly improves memory utilization as well as lowers the cost of dispatching and preempting tasks [13].

To apply the feasibility conditions of Theorem 4.1 in practice, one must account

for the overhead of an implementation of an EDF/DDM scheduler. Throughout this paper we have ignored the cost of selecting, dispatching, and preempting tasks. If the scheduling priority of tasks changes over time, as is the case in EDF/DDM scheduling, one of the most difficult implementation costs to appropriately quantify is the cost of preempting a task. It would therefore be useful to determine, for a given set, if allowing preemption between tasks is indeed necessary for feasibility. By combining individual resources into resource classes, one can represent a task system with m shared resources, as a system with k shared resources, for $1 \leq k \leq m$. (In practice this amounts to using the same monitor lock or semaphore for accessing a *set* of resources.) In this manner we can, roughly speaking, identify the "minimum" number of logical resources necessary for ensuring the schedulability of a set of tasks. For example, when using an EDF/DDM scheduler, if there exist two resources R_i and R_j, $i \neq j$, such that $P_i = P_j$, then a resource R_i requesting task will never preempt a resource R_j task (nor execute while such a task is preempted) and vice versa. Therefore, if $P_i = P_j$, one can always treat resources R_i and R_j as a single logical resource thereby simplifying the accounting for overhead when analyzing a set of tasks. For a given set of resources, there is an exponential number of possible resource classes to consider. However in practice the number of resources in a system is likely to be small and the process of enumerating and testing the feasibility of the various modified problem statements may be performed off-line.

Even if the number of logical resources required for feasibility is close to the number of actual resources in the system, we observe that in practice the number of tasks that are able to preempt other tasks is small. Note that in each (contrived) example in this paper, the schedules produced by the EDF/DDM scheduling discipline have been *non-preemptive*. For example, in the case of single phase tasks, if $P_i \leq P_j$ then no resource R_j requesting task can ever preempt a resource R_i requesting task. This implies that there will always exist a group of tasks that may never preempt any resource requesting task. Furthermore, since a task T_k may preempt a resource R_i requesting task only if $p_k < P_i$, T_k can either preempt every resource R_i requesting task or it cannot preempt any such task. Based on these observations and our experience with applying the EDF/DDM discipline to actual task sets, we conjecture that if preemption among tasks is required for feasibility, it will be limited to a few tasks. For these tasks one may account for the cost of preemption by inflating their cost parameter c to include the cost of preempting a task.

5.2 Feasibility Versus Processor Utilization

Condition (1) of Theorems 3.2 and 4.1 requires that the cumulative utilization of a set of tasks not overload the processor. Note that this is the only feasibility condition that constrains the achievable utilization of a task set. Although condition (2) of these theorems constrains the achievable utilization over a relatively short and well-defined set of intervals, it does not constrain the overall processor utilization. The feasibility of a set of sporadic tasks that share resources is not a function of processor utilization (to the extent that the tasks do not overload the processor). It is possible to conceive of both *feasible* task sets that have a processor utilization of 1.0, and *infeasible* task sets that have arbitrarily small processor utilization. An implication of this is that manipulating infeasible task sets according to such "rules-of-thumb" as lowering the processor utilization will not necessarily yield a feasible task set.

5.3 Other Paradigms of Resource Usage

We have assumed throughout that tasks require at most one resource per phase and that phases are statically ordered. The latter restriction can be mitigated to a limited extent by judicious use of the minimum phase execution time cost parameter c. A zero value for c can be used to model simple branching logic that controls the order of phase execution. An alternate approach described by Stoyenko is to explicitly test the feasibility of all possible interleavings of task invocations for all possible phase orderings [20]. We have chosen to restrict the programming model in order to ensure a simple test for feasibility.

The restriction that phases require at most one resource is certainly unrealistic for real-time systems such as in transaction systems where phases may require multiple resources simultaneously. The initial motivation for consideration of a single resource per phase arose from the use of monitors in concurrent programming languages. Operationally, we have defined a resource as a monitor. The use of multiple resources simultaneously by a task corresponds to the "nested monitor problem" in the concurrent programming literature (see [1] for a discussion). Largely because of the problems associated with deadlock, many popular concurrent programming languages such as Modula[3], Mesa, and Concurrent Euclid do not allow nested monitor calls [1]. From a pragmatic standpoint, if in practice it is the case that the number of tasks that can preempt one another is indeed small, as conjectured in Section 5.1, then we would argue that there is little to be gained by investigating more complex models of shared resources. It would be better to simply consider the resources that a phase requires simultaneously as a single logical resource. This reduces the problem to the one considered in this paper. From our perspective, a more interesting model to study is one that relaxes the mutual exclusion constraints on the access to resources. In this work resources have been required to be accessed in a mutually exclusive manner. Other models of models of exclusion, such as readers/writers, warrant consideration.

6. Summary and Conclusions

We have modeled a real-time system as a set of sporadic tasks that share a set of serially reusable, single unit re-

[3] Modula allows lexically nested monitors, however, this is compatible with our one resource per phase paradigm.

sources. Sporadic tasks are a generalization of periodic tasks and are well-suited for representing event driven processes. Tasks are composed of a sequence of phases. Each phase is a contiguous sequence of statements that possibly requires exclusive access to a resource. Resources are shared software objects, such as data structures. For an instance of the model the goal is to determine if it is possible to schedule the tasks on a uniprocessor such that (1) no task fails (every invocation of every task completes execution at or before its deadline) and (2) each instance of each resource requesting phase has exclusive access to the resource it requires for the duration of the phase.

We have identified conditions that are both necessary and sufficient for scheduling a set of sporadic tasks without the use of inserted idle time. These conditions are sufficient for scheduling periodic tasks. With respect to the class of algorithms that do not use inserted idle time, we have developed an optimal algorithm for scheduling sporadic tasks that share resources. This algorithm, called the *earliest deadline first with dynamic deadline modification* (EDF/DDM) algorithm, is an extension to the well-known EDF algorithm. Under an EDF/DDM scheduler, tasks that require exclusive access to resources have two types of deadlines: a *contending* deadline for the initial acquisition of the processor, and an *execution* deadline for subsequent execution. The EDF/DDM policy ensures tasks that become blocked due to mutual exclusion constraints are resumed as soon as possible. This policy is pessimistic in the sense that it always assumes the act of scheduling a resource requesting task will result in a competing task becoming blocked. Our analysis has demonstrated that this pessimistic approach is warranted.

The EDF.DDM scheduling algorithm and the sporadic tasking model have been implemented and used to construct several hard-real-time systems.

8. References

1. Andrews, G.R., *Concurrent Programming*, Benjamin Cummings, Redwood City, CA, 1991.
2. Becker, D, *Analysis of the NIU Firmware Performance*, University of North Carolina, unpublished manuscript, April 1992.
3. Chen, M.-I., Lin, K.-J., *Dynamic Priority Ceilings: A Concurrency Control Protocol for Real-Time Systems*, Real-Time Systems, 2, 4, (November 1990), pp. 325-346.
4. Chen, M.-I., Lin, K.-J., *A Priority Ceiling Protocol for Multiple-Instance Resources*, Proc. 12th IEEE Real-Time Sys. Symp., San Antonio, TX, December 1991, pp. 140-149.
5. Chung, J.C., *et al*., *Exploring Virtual Worlds with Head-Mounted Displays*, Non-Holographic True 3-Dimensional Display Technologies, SPIE Proc., Vol. 1083, Los Angeles, CA, January 1989.
6. Conway, R.W., Maxwell, W.L., Miller, L., *Theory of Scheduling*, Addison-Wesley, Reading, MA, 1967.
7. Garey, M.R., Johnson, D.S., *Computing and Intractability, A Guide to the Theory of NP-Completeness*, W.H. Freeman, New York, 1979.
8. Jeffay, K., Stone, D.L., Smith, F.D., *Kernel Support for Live Digital Audio and Video*, Computer Communications, 15, 6 (July 1992), pp. 388-395.
9. Jeffay, K., *On Kernel Support for Real-Time Multimedia Applications*, Proc. 3rd IEEE Wrkshp on Workstation Op. Sys., Key Biscayne, FL, April 1992.
10. Jeffay, K., *Analysis of a Synchronization and Scheduling Discipline for Realtime Tasks with Preemption Constraints*, Proc. 10th IEEE Real-Time Sys. Symp., Santa Monica, CA, Dec. 1989, pp. 295-305.
11. Jeffay, K., Stanat, D.F., Martel, C.U., *On Non-Preemptive Scheduling of Periodic and Sporadic Tasks,* Proc. 12th IEEE Real-Time Sys. Symp., San Antonio, TX, December 1991, pp. 129-139.
12. Jeffay, K., Stone, D., Poirier, D., *YARTOS: Kernel support for efficient, predictable real-time systems*, in *Real-Time Programming*, W. Halang and K. Ramamritham, eds., Pergamon Press, Oxford, 1992.
13. Jeffay, K., Stone, D.L., *The Application of Scheduling Theory to the Design and Analysis of a Real-Time Multimedia System*, University of North Carolina, in preparation.
14. Hoare, C.A.R., *Monitors: An Operating System Structuring Concept*, Comm. of the ACM, 17, 10, (October 1974), pp. 549-557.
15. Leinbaugh, D.W., *Guaranteed Response Times in a Hard-Real-Time Environment*, IEEE Trans. on Soft. Eng., 6, 1, (January 1980), pp. 85-91.
16. Liu, C.L., Layland, J.W., *Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment*, Journ. of the ACM, 20, 1, (January 1973), pp. 46-61.
17. Mok, A.K.-L., *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.
18. Mok, A.K.-L., Amerasinghe, P., Chen, M., Sutanthavibul, S., Tantisirivat, K., *Synthesis of a Message Processing System with Data-Driven Timing Constraints*, Proc. 8th IEEE Real-Time Sys. Symp., San Jose, CA, December 1987, pp. 133-143.
19. Sha, L., Rajkumar, R., Lehoczky, J.P., *Priority Inheritance Protocols: An Approach to Real-Time Synchronization*, IEEE Trans. on Computers, 39, 9, (September 1990), pp. 1175-1185.
20. Stoyenko, A.D., *A Schedulability Analyzer for Real-Time Euclid*, Proc. 8th IEEE Real-Time Sys. Symp., San Jose, CA, December 1987, pp. 218 - 227.
21. Zhao, W., Ramamritham, K., Stankovic, J.A., *Scheduling Tasks with Resource Requirements in Hard Real-Time Systems*, IEEE Trans. on Soft. Eng., 13, 5, (May 1987), pp. 564-577.