Timing Analysis for Fixed Priority Scheduling of Hard Real-Time Systems

Michael Gonzalez Harbour[1]
Departmento de Electronica
Universidad de Cantabria
39005 - Santander, Spain

Mark H. Klein[2]
Software Engineering Institute
Carnegie Mellon University
Pittsburgh, PA 15213

John P. Lehoczky[3]
Department of Statistics
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract

This paper considers the problem of timing analysis for a quite general hard real-time periodic task set on a uniprocessor using fixed priority methods. Periodic tasks are decomposed into serially executed subtasks, where each subtask is characterized by an execution time, a fixed priority, and a deadline. A method for determining the schedulability of each task is presented along with its theoretical underpinnings. This method can be used to analyze the schedulability of complex task sets which involve interrupts, certain synchronization protocols, non-preemptible sections, certain precedence constraints, and, in general, any mechanism that contributes to a complex priority structure. The method is illustrated with a realistic example.

[1]Michael Gonzalez Harbour's research is sponsored in part by the Direccion General de Investigacion Cientifica y Tecnica of the Spanish Government.

[2]Mark Klein's research at the Software Engineering Institute is sponsored by the U.S. Department of Defense.

[3]John Lehoczky's research is sponsored in part by the Office of Naval Research under contract number N0014-84-K-0734, in part by the Naval Ocean Systems Center under contract N66001-87-C-0155, and in part by the Federal Systems Division of IBM Corporation under University Agreement Y-278067.

1 Introduction

Timing behavior is important to all computing systems. However, the criticalness of timing behavior for real-time systems and the difficulty in understanding, predicting and controlling their timing behavior has distinguished these systems. As real-time systems grow in size and complexity, it is becoming increasingly important to manage timing complexity throughout the life of a system; from timing requirement specification, through design, development and testing and after deployment. Thus it has become very important to develop models and theory that can be used to reason about the behavior of real-time systems. In particular, fixed priority scheduling theory is proving to be a means for developing a sound theoretical foundation for analyzing the timing behavior of systems and designing systems that are amenable to such analysis. This paper builds on previous work and extends the domain of analyzable real-time systems to cover an important class of systems; namely message passing systems that use fixed priority preemptive scheduling.

A theoretical treatment of fixed priority scheduling first appeared in 1973 when Liu and Layland [8] introduced the *rate monotonic scheduling algorithm* for independent periodic tasks. They proved the optimality (for fixed priority scheduling) of a rate monotonic priority assignment for the case where task deadlines are coincident with the end of a task's period. They also derived a sufficient condition for the schedulability of task sets that use a rate monotonic priority assignment. Leung and Whitehead [7] later showed the optimality of the *deadline monotonic* priority assignment for the case where periodic tasks have deadlines that are at or before the end of their periods.

Fixed priority scheduling theory has received renewed consideration over the last five years. A collection of results has been developed that is proving to be very useful and is gaining popularity as a basis for reasoning about the timing behavior of real-time systems. These results are summarized by Sha, Klein, and Goodenough [15], Sha and Goodenough [12] and Lehoczky, et al. [6]. Aperiodic task scheduling has been treated in [16], synchronization requirements treated in [9, 10, 14], and mode change requirements treated in [11]. In addition, hardware scheduling support has been treated in [4, 13], implications for Ada scheduling rules discussed in [2], and schedulability analysis of input/output paradigms discussed in [3].

All of this work lays the foundation for considering a more general class of problems where logical threads of execution are not constrained to execute at a single priority level. There are many existing systems in which the original problem naturally maps onto a software architecture consisting of multiple tasks that respond to and generate events using a message passing or signal/wait paradigm. Each of these tasks is assigned a priority based upon a deadline or semantic importance. In these systems it is possible to identify execution sequences composed of several processing steps in a specific order at varying priority levels.

Even in cases where one strives to use a fixed priority assignment for periodic tasks, the task dispatching mechanism may violate this premise since periodic tasks are generally initiated by clock interrupt service routines, which usually execute at a priority level different from the task. Borger, Klein, and Veltre [1] present a schedulability analysis that considers the effects of task dispatching. The basic inheritance protocols also change execution priority as necessary. Sha and Goodenough [12] discuss a mechanism for emulating the priority ceiling protocol by setting the priority of a server task to be one level higher than the priority of all of the server's clients. This represents another instance of varying priorities. It is not uncommon for operating systems to ensure internal consistency by disabling interrupts for short periods of time, effectively creating small intervals of nonpreemptibility, which need to be considered when analyzing the timing behavior of a system.

This paper generalizes the fixed priority analysis methods by offering a framework for reasoning about timing behavior in the context of varying execution priority. The analytic framework includes the ability to analyze systems with varying execution priorities, synchronization requirements, some kinds of precedence constraints, and periodic and sporadic tasks. The remainder of this section presents a formal framework that defines the problem and discusses a difficulty that arises when task priorities vary.

This paper is organized as follows. Section 2 presents an algorithm and schedulability equations for checking task set schedulability. Section 3 introduces a simple but realistic real-time robotics application and illustrates how one uses the schedulability equations presented in Section 2. Section 4 proves the method's correctness, further generalizes the formal framework and discusses subtask priority assignment. Section 5 offers our conclusions.

1.1 The Framework

We assume there are n periodic tasks denoted by $\tau_1,...,\tau_n$. Each periodic task τ_i has a total computation requirement (C_i), a period (T_i), and a deadline (D_i). The computation requirement must be completed by the deadline or a timing fault occurs. In this paper, we assume that each periodic task may be composed of distinct subtasks, each of which may have its own timing requirement. Thus, τ_i consists of subtasks $\tau_{i1},...,\tau_{im(i)}$. Each of these subtasks is characterized by a set of parameters. In particular, τ_{ij} is characterized by (C_{ij}, D_{ij}, P_{ij}) where:

- C_{ij} = worst-case computation requirement of τ_{ij},
- D_{ij} = deadline of τ_{ij} relative to the arrival time of task τ_i, taking 0 to be this arrival time,
- P_{ij} = fixed priority level of τ_{ij}.

We assume that $0 \le D_{i1} \le . . . \le D_{im(i)} = D_i$, and we allow either $D_{im(i)} \le T_i$, or $D_{im(i)} > T_i$. The sum of the execution times associated with each subtask of task τ_i equals C_i. Each activation of a periodic task generates an instance of task execution called a *job* of that task.

Using this framework, we wish to derive a set of equations by which we can determine if all of the timing requirements of all of these tasks will be met under all possible phasings. We make the following assumptions concerning the execution behavior of any single task:

Assumption 1: Subtasks executing at a given priority level can be preempted by any subtask of higher priority.

Assumption 2: Tasks do not suspend themselves at any instant between their activation and their completion.

Assumption 3: The $(k+1)^{st}$ job of τ_i will not execute until the k^{th} job of τ_i has been completed. Furthermore, any subtask τ_{ij} is not ready for execution until subtasks τ_{ir}, $1 \le r < j$ have been completed.

Assumption 4: The time required to perform task scheduling, context swapping, and other overhead is ignored.[4]

There are situations, especially involving interrupts, in which one would want to modify Assumption 3 to allow early subtasks of a later job to interrupt unfinished later subtasks of an earlier job. The analysis given in Section 4 can be modified to handle this case, but we develop the theory specifically for Assumption 3.

To determine the schedulability of a task τ_i, we will apply a transformation to its priority structure to derive its *canonical form*.

Definition 1: A task is said to be in *canonical form* if it consists of consecutive subtasks that do not decrease in priority. The canonical form of a task τ_i is another task, τ'_i, that is obtained by applying the following algorithm where P'_{ij} denotes the priority of subtask τ'_{ij}.

[4]This assumption can be relaxed by introducing the context switch times into the execution times of each task, using a technique similar to that discussed in [1].

```
$P'_{im(i)} = P_{im(i)}$
for $l = m(i)$ downto 2
  if $P'_{il} < P_{il-1}$ then $P'_{il-1} = P'_{il}$
  else $P'_{il-1} = P_{il-1}$
end;
```

After applying the algorithm, consecutive subtasks of the transformed task with equal priority can be combined into a single subtask, if their deadlines are the same.

We will prove in Section 4 that the completion time of a task τ_i and the completion time of its canonical form τ'_i are the same. The canonical form has the advantage of being simpler for the purpose of understanding the worst-case phasing and performing the schedulability analysis.

1.2 A Difficulty

The possibility that a task's deadline, $D_{im(i)}$, exceeds its period, T_i, requires one to check more than just the deadline of the first job of a particular task. As shown in [5], all deadlines in a particular time interval called a busy period must be checked. Suppose, however, that $D_{im(i)} \le T_i$, $1 \le i \le n$, so that a job of task τ_i must be completed before the next job of τ_i is ready (assuming task deadlines are met). Consequently, earlier jobs of τ_i do not compete with later jobs, and it would seem that one could follow a standard Liu and Layland approach, namely constructing a worst-case phasing of all other tasks and checking that the first job of τ_i meets its deadline. The difficulty with this reasoning is that it overlooks another way in which earlier jobs of τ_i can influence later jobs. If an early subtask of τ_i, say τ_{ip}, has a relatively low priority, P_{ip}, while a later subtask, τ_{ir}, $p < r$, has a relatively high priority, $P_{ip} < P_{ir}$, then τ_{ir} can delay a medium priority subtask τ_{kq}, $k \neq i$ where $P_{ip} < P_{kq} < P_{ir}$. This delay in τ_{kq} creates an intermediate phasing which cannot be created as an initial phasing and which can lead to longer response times for the next job of subtask τ_{ip}. Consider the following example.

Example: Let n=2, where task τ_1 has one subtask given by $C_{11} = 4$, $D_{11} = 10$, and $T_1 = 10$. Task τ_2 has period $T_2 = 14$ and two subtasks characterized by $C_{21} = 6$, $D_{21} = 14$, and $C_{22} = 2$, $D_{22} = 14$. Suppose further that $P_{21} < P_{11} < P_{22}$.

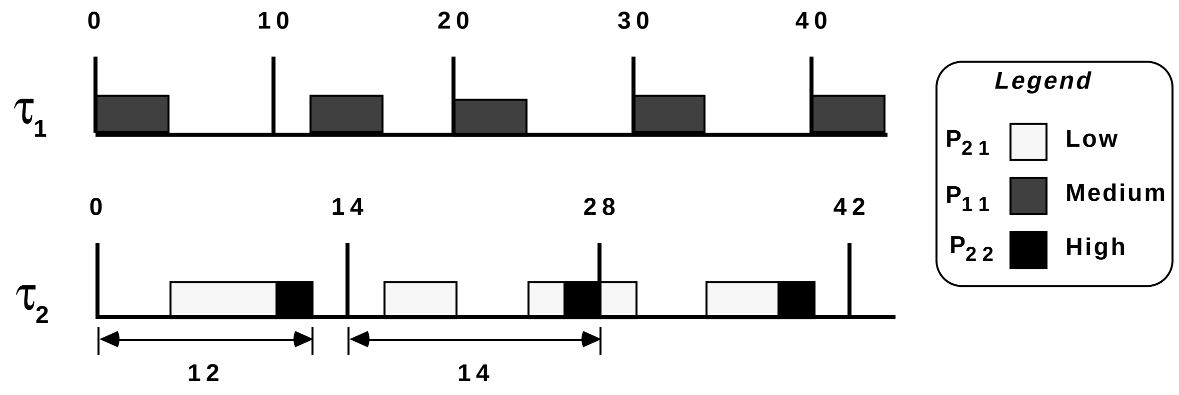


Figure 1: Considering the First Job is Not Sufficient

Under the traditional (Liu and Layland) worst-case phasing shown in Figure 1, the longest response time for τ_{22} is given by 14 for the second job of τ_2, whereas the first job has a response time of 12. Thus, the longest response time of this task is associated with its second job rather than the first. The cause of this phenomenon is the high priority accorded to τ_{22} which delays the start of the second job of τ_1. This, in turn, creates the long response times for the subtasks of τ_2. Note that had τ_i not been divided into two parts, but had been given its rate monotonic priority, then the task set would not be schedulable. We will return to this point in Section 4.11.

1.3 Busy Period

The above example shows that one may need to check the deadlines of more than one job of a particular task. The criterion for which deadlines need to be checked is based on the concept of a *busy period*. The concept of a busy period is well known in queuing theory and was first introduced in real-time scheduling by Lehoczky [5]; however, we need to modify this concept slightly to accommodate the fact that a single task may have subtasks with different priorities. Let $P_i = \min(P_{ij}, 1 \le j \le m(i))$ denote the minimum priority level of all of the subtasks of task τ_i.

Definition 2: A τ_i-*idle instant* is any time *t* such that all work of priority P_i or higher started before *t* and all τ_i jobs also started before *t* have completed at or before *t*.

Definition 3: A τ_i-*busy period* is an interval of time [*A*,*B*] such that both *A* and *B* are τ_i-idle instants and there is no time $t \in (A,B)$ such that t is a τ_i-idle instant.

Intuitively, a τ_i-busy period is a time interval during which the processor is continuously processing at priority level P_i or higher. It is also "self-contained" in the sense that any job of task τ_i that is started during the busy period is also completed during the busy period. Moreover, all work of priority level P_i or higher which is ready at some time during the busy period is also finished by the end of the busy period.

2 A Method for Determining Schedulability

This section describes a procedure for determining the schedulability of a task set of the general form described in the previous section. We will assume that we are analyzing a single task in the task set, and the method used can be subsequently applied to all of the other tasks.

2.1 Final Deadline

First, we will focus on the task’s final deadline. Our goal is to determine if task τ_i will meet its final deadline. The general strategy is very similar to that used by Liu and Layland [8] and Lehoczky [5]. One must first find the phasing of the other tasks relative to task τ_i that results in a critical instant for task τ_i. A *critical instant* is a point in time such that if task τ_i is activated at that point, its completion time will be the longest. In order to determine the critical instant phasing, we will first divide the other n-1 tasks into several groups.

2.1.1 Task Groups: Tasks are placed into groups based upon the priorities of their subtasks relative to τ_i. A key criterion is the priority of the first subtask relative to the minimum priority of all subtasks of task τ_i. For example, a task that starts with a high-priority subtask will eventually be able to preempt the subtask of task τ_i with the minimum priority. On the other hand, a task that starts with a lower priority subtask will never have this opportunity. Since the task groupings are relative to the priority structure of task τ_i, the groups will vary as a function of the task being analyzed.

Recall that $P_i = \min\left(P_{ij}, 1 \le j \le m(i)\right)$ denotes the minimum priority of all of the subtasks of task τ_i. We refer to a sequence of consecutive subtasks as a *segment*. An *H segment* comprises a sequence of consecutive subtasks, each of which has a priority equal to or greater than P_i. Note that this is a possibly pessimistic treatment of equal priority subtasks, allowing for a worst-case analysis and thus covers all scheduling policies for handling equal priorities. Similarly, an *L segment* refers to any set of consecutive subtasks, each of which has priority strictly less than P_i. In the following description of task types, a "+" denotes one or more patterns. A "0" denotes zero or one patterns. An effect due to preemption by a high priority first segment will be referred to as a *preemption effect*. An effect due to a high-priority segment that occurs after a low-priority segment will be referred to as a *blocking effect*. Figure 1 shows preemption effects of τ_1 over τ_2 at times t=0, t=12, t=20, and t=30; a blocking effect of τ_2 on τ_1 is shown at time t=10.

The five types of tasks are:

- **Type 1 (H)** tasks may be able to preempt task τ_i more than once per τ_i-busy period. Each task in this group is used to determine the worst-case completion time for task τ_i.
- **Type 2 ($(HL)^+$)** tasks are such that each high-priority segment is followed by a low-priority segment. Consequently, each task in this set can preempt task τ_i only once per τ_i-busy period because the low-priority segments will have to wait until the busy period is complete before they can execute. Depending on types 3 and 4, one task in this set may be used for its blocking effect rather than its preemption effect.
- **Type 3 ($(HL)^+H$)** differ from the previous type in that they end with a high-priority segment. In general these tasks are treated like tasks of type 2. However, under special circumstances (described below) one task may exhibit both a preemption effect and a blocking effect.
- **Type 4 ($(LH)^+L^0$)** tasks are solely blocking tasks. Moreover, at most one task in this group can contribute a blocking effect.
- **Type 5 (L)** tasks have no effect on the completion time of task τ_i and thus can be ignored.

2.1.2 Schedulability Equations: In this section a step-by-step procedure is described for determining if tasks can meet their final deadlines. The procedure entails methodically identifying blocking effects and preemption effects, and constructing equations that account for how these effects contribute to the completion time of task τ_i.

We will assume throughout this section that task τ_i has been transformed to canonical form and that adjacent subtasks of equal priority have been compressed into a single subtask. This results in a task where each subtask has a priority strictly higher than its predecessor. The rest of the tasks remain unchanged.

The procedure determines the completion time of the first subtask of the transformed canonical form. It then iteratively determines the completion of the (j+1)st subtask as a function of the jth subtask until the completion time of the final subtask has been determined. This is performed for every job in the τ_i-busy period.

Some notation is needed prior to discussing the procedure. Let MP_{ij} denote the set of tasks that can have more than one preemptive effect relative to the priority of the subtask τ_{ij}. Recall that $P_p = \min(P_{pj}, 1 \le j \le m(p))$.

$$MP_{ij} = \left\{ \tau_p \,\middle|\, p \neq i \wedge P_p \ge P_{ij} \right\}$$

For example, since τ_i is assumed to be in canonical form, MP_{i1} is simply the set of type 1 tasks for τ_i. Since the priority of the second subtask of the canonical form is greater than the first, not all of the tasks that are multiply preemptive (i.e., can preempt more than once during a task τ_i-busy period) relative to τ_{i1} will continue to be multiply preemptive during the execution of the second subtask. Consider the example in Figure 1. Task 1 is multiply preemptive until task 2 enters its second subtask; then task 1 can no longer preempt.

Let SP_{ij} be the set of singly preemptive tasks relative to τ_{ij}. Furthermore, given a task $\tau_p \in SP_{ij}$, let $h(p,i,j)$ denote the number of subtasks that comprise the initial H segment of task τ_p, $p \neq i$, relative to subtask τ_{ij}, where

$$h(p,i,j) = \left(h \,\middle|\, p \neq i \ \wedge \ (P_{p1},\ldots,P_{ph} \ge P_{ij}) \wedge \ (P_{ph+1} < P_{ij}) \right)$$

The execution time associated with the leading H segment of one of these tasks is denoted as:

$$C_p^{h(p,i,j)} = \sum_{k=1}^{h(p,i,j)} C_{pk}$$

In the context of a task τ_p and a subtask τ_{ij}, h will serve as notational shorthand for $h(p,i,j)$ and C_p^h for $C_p^{h(p,i,j)}$.

The stepwise procedure for determining if task τ_i can meet its final deadline follows.

Step 1: Find the worst-case phasing for the other n-1 tasks.

In this step tasks other than τ_i are placed into the groups defined in the previous section, MP_{i1} and SP_{i1} are determined, and then the blocking term, B_i, is identified.

1. Let MP_{i1} be the set of type 1 tasks for τ_i and set SP_{i1} to the union of type 2 and type 3 tasks.
2. Determine the longest H segment of the type 4 tasks. Denote the length of this as B'.
3. For each type 2 and 3 task, denote the length of the longest inner blocking H segment (if any) as U, the final blocking H segment (if any) as V, and the initial H segment as W. If there is no inner blocking segment then set U=0. If there is no final blocking segment then set V=0.

   Calculate max(U-W-B', V-B') for each task and choose the task with the largest value (let τ_m be this task). If this value is negative then B_i=B'. If this value is positive then a new blocking term must be computed. We must determine if the blocking term will be from the inner segment or the final segment of task τ_m. If U-W > V, then set B_i=U and remove task τ_m from SP_{i1}; otherwise set B_i=V and τ_m remains in SP_{i1}.

Step 2: Determine how many jobs of task τ_i must be checked.

1. The length of the task-τ_i busy period is:

$$L_i = \min\Big(t > 0 \;\Big|\; B_i + \sum_{\tau_p \in MP_{i1}} \lceil t/T_p \rceil C_p + \sum_{\tau_p \in SP_{i1}} C_p^h + \lceil t/T_i \rceil C_i = t \Big)$$

   The length of the τ_i-busy period is determined by considering processing contributed by:
   - the blocking term,
   - the multiply preemptive tasks (relative to the first subtask),
   - the singly preemptive tasks (relative to the first subtask), and
   - complete jobs of τ_i.

   The idea is to look for the minimum time t, such that all work of priority P_i or higher initiated in the interval [0,t] is completed at time t. Notice that multiply preemptive tasks and τ_i may contribute processing more than once during the τ_i-busy period, and the blocking term and singly preemptive tasks contribute exactly once during this busy period.

2. The number of jobs of τ_i in the busy period is:

$$N_i = \lceil L_i/T_i \rceil$$

Step 3: Check the completion time of each of the N_i jobs in the τ_i-busy period.

1. The completion time of the first subtask of the k^th^ job of task τ_i is represented by $E_{i1}(k)$.

$$E_{i1}(k) = \min\Big(t > 0 \;\Big|\; B_i + \sum_{\tau_p \in MP_{i1}} \lceil t/T_p \rceil C_p + \sum_{\tau_p \in SP_{i1}} C_p^h + (k-1)C_i + C_{i1} = t\Big)$$

   To understand this equation, consider as an example the first subtask of the first job in canonical form. Under worst-case phasing, the completion of this subtask is impacted by the blocking term, all higher priority processing that is initiated at the same instant and the execution time of the subtask itself. The right side of the equation represents the first point in time t, where all of the processing initiated by multiply preemptive and single preemptive tasks in the interval [0,t], the processing associated with the blocking term, and the processing associated with the subtask itself has been completed.

2. Given the completion time of the first subtask of the kth job, it is possible to calculate the completion of the second subtask of the kth job. The first step in this calculation is to insert the correct elements into the set SP_{i2}. Insert into SP_{i2} those tasks that were multiply preemptive relative to subtask τ_{i1}, but due to the higher priority of τ_{i2}, are singly preemptive relative to τ_{i2}.

$$SP_{i2} = \Big\{ \tau_p \;\Big|\; \tau_p \in (MP_{i1} - MP_{i2}) \;\wedge\; \Big(\exists l \;\Big|\; (P_{p1},\ldots,P_{pl} \geq P_{i2}) \wedge (P_{pl+1} < P_{i2}) \Big) \Big\}$$

SP$_{i2}$ is constructed by first selecting those tasks from MP$_{i1}$ that may have become singly preemptive. This is accomplished via the set subtraction in the equation. Secondly, one must ensure that each of the selected tasks is actually a singly preemptive task relative to τ_{i2}. This is accomplished by making sure that there exists a leading segment that has a priority greater or equal to P$_{i2}$. Notice that tasks in SP$_{i1}$ are not included in SP$_{i2}$. This is because these tasks only preempt once during a τ_i-busy period and this single preemption has already been accounted for in the calculation of E$_{i1}$.

3. Using the completion time of the first subtask and sets SP$_{i2}$ and MP$_{i2}$, the completion time of the second segment is:

$$E_{i2}(k) = \min\Big(t > 0 \;\Big|\; E_{i1}(k) + \sum_{\tau_p \in MP_{i2}} \big[\lceil t/T_p \rceil - \lceil E_{i1}(k)/T_p \rceil \big] C_p + \sum_{\tau_p \in SP_{i2}} \min\Big(1, \big[\lceil t/T_p \rceil - \lceil E_{i1}(k)/T_p \rceil \big] \Big) C_p^h + C_{i2} = t \Big)$$

The equation uses the completion time of the first subtask (in canonical form) as the starting point for calculating the completion time of the second subtask. The first summation represents the multiply preemptive processing initiated by tasks after the completion of the first subtask. The second summation represents the singly preemptive processing initiated by tasks after the completion of the first subtask. The "min" function within the second summation ensures that singly preemptive tasks can have at most one preemption effect on the remainder of the job. The execution time of the second subtask is then added in.

4. In general, given the completion time of the jth subtask of the kth job, it is possible to calculate the completion of the (j+1)st subtask of the kth job.

Once again we must first insert the proper elements into the set SP$_{ij+1}$.

$$SP'_{ij+1} = \Big\{ \tau_p \;\Big|\; \tau_p \in SP_{ij} \;\wedge\; \Big(\lceil E_{ij}(k)/T_p \rceil - \lceil E_{ij-1}(k)/T_p \rceil \Big) = 0 \;\wedge\; \Big(\exists l \;\Big|\; (P_{p1},\ldots,P_{pl} \geq P_{ij+1}) \wedge (P_{pl+1} < P_{ij+1}) \Big) \Big\}$$

SP$'_{ij+1}$ is the set of singly preemptive tasks relative to P$_{ij}$ that have not yet exhibited their singly preemptive effect and are also singly preemptive relative to P$_{ij+1}$, and

$$SP''_{ij+1} = \Big\{ \tau_p \;\Big|\; \tau_p \in (MP_{ij} - MP_{ij+1}) \wedge \Big(\exists l \;\Big|\; (P_{p1},\ldots,P_{pl} \geq P_{ij+1}) \wedge (P_{pl+1} < P_{ij+1}) \Big) \Big\}$$

SP$''_{ij+1}$ is the set of multiply preemptive tasks relative to P$_{ij}$ that are singly preemptive relative to P$_{ij+1}$. The set SP$_{ij+1}$ is the union of the above two sets.

$$SP_{ij+1} = SP'_{ij+1} \cup SP''_{ij+1}$$

The calculation for E$_{ij+1}$(k) is:

$$E_{ij+1}(k) = \min\Big(t > 0 \;\Big|\; E_{ij}(k) + \sum_{\tau_p \in MP_{ij+1}} \big[\lceil t/T_p \rceil - \lceil E_{ij}(k)/T_p \rceil \big] C_p + \sum_{\tau_p \in SP_{ij+1}} \min\Big(1, \big[\lceil t/T_p \rceil - \lceil E_{ij}(k)/T_p \rceil \big] \Big) C_p^h + C_{ij+1} = t \Big)$$

All jobs of task τ_i will meet their deadlines if the following condition is satisfied.

$max((k{-}1)T_i + D_i - E_{im(i)}(k)) \geq 0 \ \ for \ \ k \leq N_i$

2.2 Deadlines for Other Subtasks

The analysis of the other subtasks is very similar (and in some cases identical) to the analysis of the final subtask. The only difference is in the analysis of the first job. Assume that subtask τ_{ij} is to be analyzed, where $j \neq m(i)$. Let $P_i(j) = \min(P_{ik}, 1 \leq k \leq j)$. Special analysis of the first job of τ_{ij} is necessary only if $P_i(j) > P_i$.

If the special analysis is needed, then truncate task τ_i after subtask τ_{ij} and use the algorithm for converting a task to canonical form on this truncated task. Determine task groupings for the truncated task and apply the stepwise method from the previous section. The schedulability test for the first job of subtask τ_{ij} is: $D_{ij} - E_{ij}(1) \geq 0$

3 Applying the Method to an Example

3.1 Problem Description

We will use an example developed from a real-time robotics application to illustrate the utility of the theory developed in this paper and how to perform a schedulability analysis using the method described in the previous section. This example is derived from a real robot system that measures the shape of pipes inside a nuclear reactor, by moving around them and using a distance sensor. The task set corresponding to this system has been simplified to reflect only the important activities relevant to our analysis, and the numbers used are not exact, although they approximate the real magnitudes. With no loss of generality, we will consider all tasks to be periodic, by using a worst-case arrival assumption for those tasks whose nature is essentially aperiodic, namely that those tasks arrive at their maximum expected rates.

The system, which has five tasks, is represented in Figure 2. A sequence of tasks that execute serially at varying priorities is considered to be a single task with multiple subtasks, for our analysis. For example, task τ_1 is considered to be a single task, but is in fact composed of two system tasks, an interrupt service routine (ISR) and Servo Control, where Servo Control executes only as a consequence of being signaled by the ISR. The five tasks are:

- **Robot control**. Task τ_1 has to control the robot's servomotors and has two subtasks, that have two different deadlines. The corresponding activities are: reading the inputs from the servo sensors and performing the control action for moving the robot.
- **Measurement subsystem**. Tasks τ_2 and τ_3 constitute the measurement subsystem, and synchronize with each other: τ_2 reads the distance sensors and does some data preprocessing, while τ_3 does some more processing and sends the results to a remote system.
- **System command**. Task τ_4 is in charge of receiving and interpreting commands arriving from the remote system, while τ_5 has to process and execute these commands. Both tasks synchronize with each other, and τ_5 also has to update some control variables that affect the operation of the rest of the tasks.

Hardware interrupts, synchronization, and existence of different deadlines lead to a task structure in which each task has several subtasks, each characterized by having different priorities and worst-case execution times. Priority ceiling protocol emulation [12] is being used for task synchronization and is responsible for the assignment of some of the priority levels. Table 1 shows the subtasks and characteristics of each task. All time values are in milliseconds.

- The lower level priorities of each task have been assigned according to rate monotonic order, using the task periods.
- Tasks τ_1 and τ_4 start with an ISR, and therefore have the priorities of their first sections fixed by the system's hardware.
- Tasks τ_2 and τ_3 synchronize with each other in their middle and final subtasks, respectively, and execute both of these subtasks at the same elevated priority. The reason for this elevated priority level will become apparent in the analysis phase.

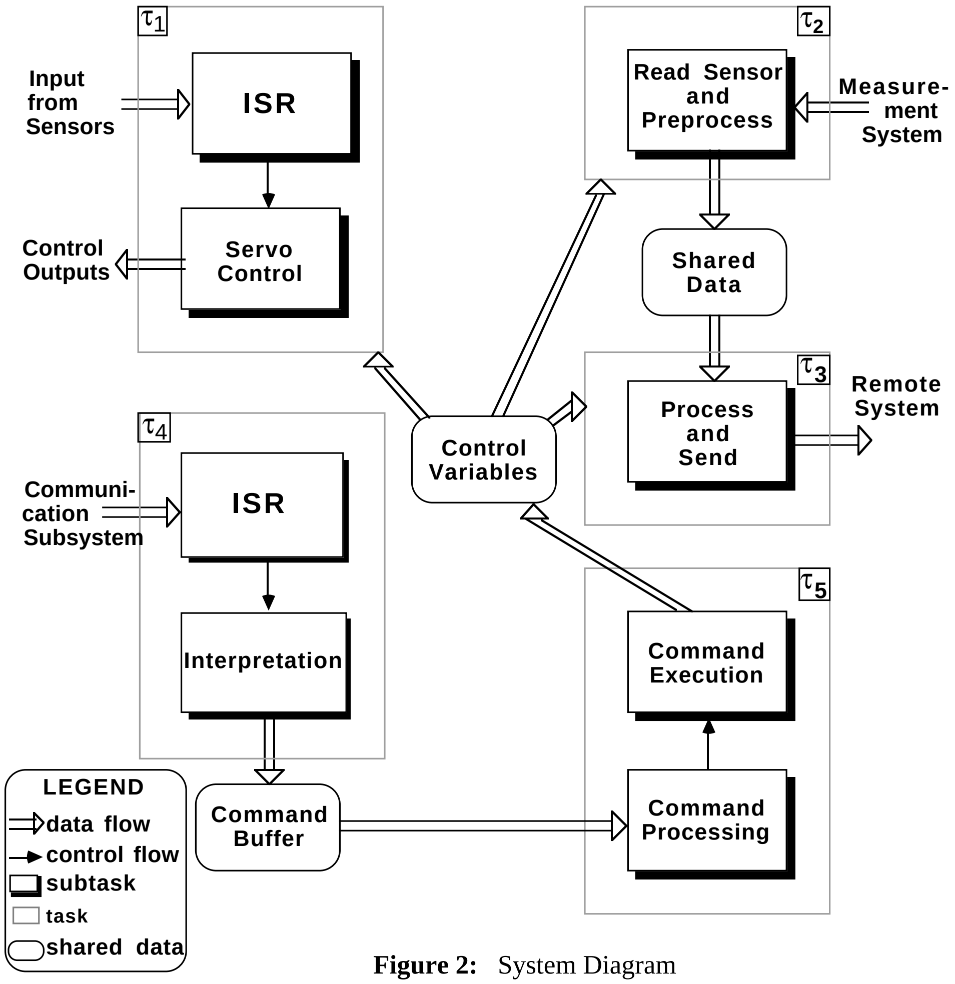


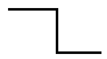
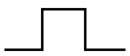
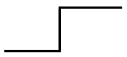
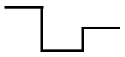
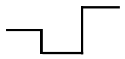
Figure 2: System Diagram

- Tasks τ_4 and τ_5 also synchronize with each other, and are assumed to have their final and initial subtasks, respectively, executing at the same priority.
- Task τ_5's final subtask must modify some control variables, and is therefore executed at relatively high priority to prevent interference from some of the other tasks.

Although tasks τ_1 and τ_4 start with an ISR and could potentially have a self-preemption effect and violate Assumption 3, the fact that their final deadlines are before or at the end of their periods ensures that this effect cannot happen, if task deadlines are met. Therefore, all of the analysis developed in this paper applies to these two tasks, as long as they meet their deadlines.

Each task has its final deadline at the end of its period, and task τ_1 has an additional deadline for its first subtask, due to physical constraints on the sensors: D_{11} = 1ms. The total CPU utilization is 97.5%. Our goal is to derive a worst-case phasing or critical instant for each task, to be able to analyze the worst-case response times. In this way we can determine if the timing requirements of each task, and of the required associated subtasks, can be met under all circumstances.

Table 1: Task Set Characteristics

	Timing Requirements					Priority Structure			
	T_i	C_{i1}	C_{i2}	C_{i3}	D_i	P_{i1}	P_{i2}	P_{i3}	Shape
τ_1	40	1	5	—	40	10	7	—	
τ_2	100	10	5	5	100	4	8	4	
τ_3	50	8	12	—	50	5	8	—	
τ_4	200	10	20	3	200	9	2	3	
τ_5	400	2	12	10	400	3	1	6	

3.2 Problem Solution

Before starting the analysis of each task we will reduce it to its canonical form. The first step in the analysis will be to determine the critical instant phasing, by identifying the blocking term and the multiply preemptive and singly preemptive sets. The second step is to obtain the number of jobs that have to be checked by evaluating the length of the task's busy period. Finally, the worst-case completion time for each job will be obtained by application of the schedulability equations, and checked against the deadlines or timing requirements.

3.2.1. Analysis of τ_1: The transformed canonical form of τ_1 is obtained by lowering the priority of its first subtask, and combining the resultant equal priority subtasks into a single segment with priority 7.

Step 1. The lowest priority level in task τ_1' is 7. We will classify the rest of the tasks according to this priority level to determine the critical instant phasing.

τ_2: It is an LHL task (type 4), and its only contribution to the critical instant may be a blocking segment.

τ_3: It is an LH task (also type 4). It can only contribute with a blocking segment.

τ_4: It is an HL task (type 2), so it is classified as a singly preemptive task.

τ_5: It is an L task (type 5) and has no effect on the critical instant.

Consequently, we have task τ_4 in the set of single preemptive tasks (SP_{11}) with its first subtask τ_{41} acting as the H segment, and $B_1 = C_{32}$ as the maximum of the blocking terms.

Step 2. The length of the busy period is:

$$L_1 = \min\left(t > 0 \;\middle|\; B_1 + \sum_{\tau_p \in MP_{11}} \lceil t/T_p \rceil C_p + \sum_{\tau_p \in SP_{11}} C_p^h + \lceil t/T_1 \rceil C_1 = t \right)$$

$$L_1 = \min\left(t > 0 \;\middle|\; 0 + C_{32} + C_{41} + \lceil t/T_1 \rceil C_1 = t \right)$$

$$L_1 = 0 + 12 + 10 + 1 \cdot 6 = 28$$

Therefore, there is only one job, $N_1 = \lceil 28/T_1 \rceil = 1$, of τ_1 in the busy period. As it can be seen, when the task has only one segment of constant priority and the deadline is before or at the end of the period, there can only be a single job in the busy period if it makes its deadline. This fact can be used as a shortcut for this step of the analysis.

Step 3. As the transformed canonical form has only one segment and there is only one job in the busy period, the completion time and the busy period expressions become identical. Consequently:

$E_1(1) = 28 \leq 40 = D_1$

Intermediate Deadline. Task τ_1 meets its final deadline, but it also has a deadline for its first subtask, which has to be checked. According to the schedulability rules, τ_1's critical instant is valid for all jobs of this first subtask, except for the first one. For this first job we have to create a different critical instant, according to its own priority level. In this case, the first subtask has the highest priority in the system; therefore, all the rest of the tasks can be classified as low priority (type 5), and the completion time of this first subtask will be:

$E_{11}(1) = C_{11} = 1 \leq 1 = D_{11}$

3.2.2 Analysis of τ_2: The transformed canonical form for τ_2 is a task with a single segment τ'_{21} of priority 4.

Step 1. For task τ_2 we will classify the rest of the tasks according to its lowest priority level, which is 4.

τ_1,τ_3:	They are H tasks (type 1) and have to be included among the multiply preemptive tasks.
τ_4:	It is an HL task (type 2), so it is classified into the singly preemptive set.
τ_5:	It is an LH task (type 4) and has a potential blocking effect on τ_2.

Thus, we have tasks τ_1 and τ_3 in the set of multiply preemptive tasks (MP_{21}), τ_{41} as a singly preemptive segment, and $B_2 = C_{53}$ as the only blocking term.

Step 2. The length of the busy period is:

$$L_2 = \min\left(t > 0 \,\middle|\, B_2 + \sum_{\tau_p \in MP_{21}} \lceil t/T_p \rceil C_p + \sum_{\tau_p \in SP_{21}} C_p^h \;+\; \lceil t/T_2 \rceil C_2 = t \right)$$

$$L_2 = \min\left(t > 0 \,\middle|\, C_{53} + \lceil t/T_1 \rceil C_1 + \lceil t/T_3 \rceil C_3 + C_{41} + \lceil t/T_2 \rceil C_2 = t \right)$$

$$L_2 = 10 + 3 \cdot 6 + 2 \cdot 20 + 10 + 1 \cdot 20 = 98$$

Finding the minimum t > 0 that satisfies the equation above can be accomplished by supposing an initial positive but very small value for t, obtaining the ceiling functions, and adding all the terms in the left-hand side of the equality. Using this result as the new value for t, we repeat the same process until we find a value of t that is the same as in the last iteration and, therefore, satisfies the equality. If the total utilization is less than or equal to 100%, then we know that the busy period will end (and so will the algorithm). In this particular case, we find that there is only one job of τ_2 in the busy period.

Step 3. As the transformed canonical task has only one segment of constant priority, the completion time has the same expression as the length of the busy period. Therefore:

$E_2(1) = 98 \leq 100 = D_2$

Task τ_2 also meets its final deadline.

3.2.3. Analysis of τ_3: Task τ_3 is already in canonical form.

Step 1. Task τ_3 has its basic priority at level 5, and the classification of the rest of the tasks is:

τ_1: It is an H task (type 1) and is included among the multiply preemptive tasks.

τ_2: It is an LHL task (type 4), so its H segment has a potential blocking effect.

τ_4: It is an HL task (type 2), so it is classified as a singly preemptive task.

τ_5: It is an LH task (type 4) and has a potential blocking effect on τ_2.

Thus, we have task τ_1 in the set of multiply preemptive tasks (MP_{31}), τ_{41} as a singly preemptive segment, and $B_3 = C_{53}$ as the maximum blocking term.

Step 2. The length of its busy period is:

$$L_3 = \min\left(t > 0 \mid B_3 + \sum_{\tau_p \in MP_{31}} \lceil t/T_p \rceil C_p + \sum_{\tau_p \in SP_{31}} C_p^h + \lceil t/T_3 \rceil C_3 = t\right)$$

$$L_3 = \min\left(t > 0 \mid C_{53} + \lceil t/T_1 \rceil C_1 + C_{41} + \lceil t/T_3 \rceil C_3 = t\right)$$

$$L_3 = 10 + 2 \cdot 6 + 10 + 2 \cdot 20 = 72$$

Consequently, there are two jobs, $N_3 = \lceil 72 / T_3 \rceil = 2$, of τ_3 in the busy period, which have to be checked.

Step 3. The completion time of the first segment of the first job is:

$$E_{31}(1) = \min\left(t > 0 \mid B_3 + \sum_{\tau_p \in MP_{31}} \lceil t/T_p \rceil C_p + \sum_{\tau_p \in SP_{31}} C_p^h + C_{31} = t\right)$$

$$E_{31}(1) = \min\left(t > 0 \mid C_{53} + \lceil t/T_1 \rceil C_1 + C_{41} + C_{31} = t\right)$$

$$E_{31}(1) = 10 + 1 \cdot 6 + 10 + 8 = 34$$

For the second segment, we have to recalculate the sets of multiple and single preemptive tasks. MP_{32} will be empty, as there are no tasks with all their subtasks with priority higher than or equal to 8. SP_{32} can get elements from the tasks that were originally in MP_{31} and are not in MP_{32}, if their priority is first higher and then lower than P_{32}. Task τ_1 satisfies these conditions and therefore the analysis for this subtask is:

$$E_{32}(1) = \min\Big(t > 0 \mid E_{31}(1) + \sum_{\tau_p \in MP_{32}} \big[\lceil t/T_p \rceil - \lceil E_{31}(1)/T_p \rceil\big] C_p +$$
$$\sum_{\tau_p \in SP_{32}} \min\big(1, \big[\lceil t/T_p \rceil - \lceil E_{31}(1)/T_p \rceil\big]\big) C_p^h + C_{32} = t\Big)$$

$$E_{32}(1) = \min\Big(t > 0 \mid E_{31}(1) + \min\big(1, \big[\lceil t/T_1 \rceil - \lceil E_{31}(1)/T_1 \rceil\big]\big) C_{11} + C_{32} = t\Big)$$

$$E_{32}(1) = 34 + 1 \cdot 1 + 12 = 47 \le 50 = D_3$$

And now we have to check the second job, in the same way:

$$E_{31}(2) = \min\left(t > 0 \mid C_{53} + \lceil t/T_1 \rceil C_1 + C_{41} + C_3 + C_{31} = t\right)$$

$$E_{31}(2) = 10 + 2 \cdot 6 + 10 + 20 + 8 = 60$$

$$E_{32}(2) = \min\Big(t > 0 \mid E_{31}(2) + \min\big(1, \big[\lceil t/T_1 \rceil - \lceil E_{31}(2)/T_1 \rceil\big]\big) C_{11} + C_{32} = t\Big)$$

$$E_{32}(2) = 60 + 0 \cdot 1 + 12 = 72 \le 50 + 50 = T_3 + D_3$$

We can see that task τ_3 meets its deadlines through all of the busy period, thanks to the high priority of its last subtask. If the priority of this subtask had not been so high, let us say it remained at 5, then the first job would have missed its deadline.

3.2.4. Analysis of τ_4: The transformed canonical form for τ_4 is a task with two segments: τ'_{41} of priority 2 and τ'_{42} of priority 3.

Step 1. Task τ_4 has its lower priority subtask at level 2, so we will classify the rest of the tasks accordingly.

τ_1,τ_2,τ_3: They are H tasks (type 1) and their subtasks fall into the multiple preemptive set.

τ_5: It is an HLH task (type 3), so its last segment has a potential blocking effect on τ_4, and its first segment a one-time preemptive effect.

Thus, we have tasks τ_1, τ_2, and τ_3 in the set of multiply preemptive tasks (MP_{41}), τ_{51} as a singly preemptive segment and $B_4 = C_{53}$ as the only blocking term.

Step 2.The length of the busy period is:

$$L_4 = \min\left(t > 0 \,\middle|\, C_{53} + \lceil t/T_1 \rceil C_1 + \lceil t/T_2 \rceil C_2 + \lceil t/T_3 \rceil C_3 + C_{51} + \lceil t/T_4 \rceil C_4 = t \right)$$

$$L_4 = 10 + 5 \cdot 6 + 2 \cdot 20 + 4 \cdot 20 + 2 + 1 \cdot 33 = 195$$

Consequently, there is only one job of τ_4 in the busy period.

Step 3. We will now obtain the completion time of this job, starting with the first segment of its canonical form:

$$E'_{41}(1) = \min\left(t > 0 \,\middle|\, C_{53} + \lceil t/T_1 \rceil C_1 + \lceil t/T_2 \rceil C_2 + \lceil t/T_3 \rceil C_3 + C_{51} + C_{41} + C_{42} = t \right)$$

$$E'_{41}(1) = 10 + 5 \cdot 6 + 2 \cdot 20 + 4 \cdot 20 + 2 + 10 + 20 = 192$$

For the completion time of the second segment of the canonical form, tasks τ_1, τ_2, and τ_3 remain as multiple preemptive in the MP_{42} set. The analysis is:

$$\begin{aligned} E'_{42}(1) = \min\Big(t > 0 \,\Big|\, & E'_{41}(1) + \\ & \left[\lceil t/T_1 \rceil - \lceil E'_{41}(1)/T_1 \rceil \right] C_1 + \\ & \left[\lceil t/T_2 \rceil - \lceil E'_{41}(1)/T_2 \rceil \right] C_2 + \\ & \left[\lceil t/T_3 \rceil - \lceil E'_{41}(1)/T_3 \rceil \right] C_3 + C_{43} = t \Big) \end{aligned}$$

$$E'_{42}(1) = 192 + 0 + 0 + 0 + 3 = 195 \le 200 = D_4$$

Task τ_4 also meets its final deadline.

3.2.5. Analysis of τ_5: The transformed canonical form for τ_5 is a task with two segments: τ'_{51} of priority 1 and τ'_{52} of priority 6.

Step 1. Task τ_5 has its lower priority subtask at level 1, which is the lowest priority in the system, so all the rest of the tasks are of type 1.

Step 2.The length of the busy period is:

$$L_5 = \min\left(t > 0 \;\middle|\; \lceil t/T_1 \rceil C_1 + \lceil t/T_2 \rceil C_2 + \lceil t/T_3 \rceil C_3 + \lceil t/T_4 \rceil C_4 + \lceil t/T_5 \rceil C_5 = t \right)$$

$$L_5 = 10 \cdot 6 + 4 \cdot 20 + 8 \cdot 20 + 2 \cdot 33 + 1 \cdot 24 = 390$$

Consequently, there is only one job of τ_4 in the busy period.

Step 3. We will now obtain the completion time of this job. The analysis of the first segment of the canonical form is:

$$E'_{51}(1) = \min\left(t > 0 \;\middle|\; \lceil t/T_1 \rceil C_1 + \lceil t/T_2 \rceil C_2 + \lceil t/T_3 \rceil C_3 + \lceil t/T_4 \rceil C_4 + C_{51} + C_{52} = t \right)$$

$$E'_{51}(1) = 5 \cdot 6 + 2 \cdot 20 + 4 \cdot 20 + 1 \cdot 33 + 2 + 12 = 197$$

For the analysis of the second segment τ_1 remains as a multiply preemptive task, τ_{41} is a singly preemptive segment, and the rest of the tasks have no influence. Therefore:

$$\begin{aligned} E'_{52}(1) = \min\Big(t > 0 \;\Big|\; & E'_{51}(1) + \left[\lceil t/T_1 \rceil - \lceil E'_{51}(1)/T_1 \rceil \right] C_1 + \\ & \min\left(1, \left[\lceil t/T_4 \rceil - \lceil E'_{51}(4)/T_4 \rceil \right]\right) C_{41} + C_{53} = t \Big) \end{aligned}$$

$$E'_{52}(1) = 197 + 1 \cdot 6 + 1 \cdot 10 + 10 = 223 \le 400 = D_5$$

Task τ_5 also meets its final deadline, so the total task set is schedulable.

4 Theoretical Analysis

In this section, we present the theoretical underpinnings that support the rules for schedulability analysis that were presented and used in Sections 2 and 3.

4.1 Busy Period Analysis

The focus of this section is finding the longest response time for a particular subtask τ_{ij}. Once this has been found, we check whether it is less than the subtask deadline D_{ij}. If so, then the subtask timing requirements will be met under all task phasings. We define P_i to be $\min(P_{ij}, 1 \le j \le m(i))$. If we select any particular phasing of the n tasks and consider the resulting processor execution sequence that evolves over time, we can partition this execution sequence into intervals of time of two types: τ_i-busy periods, as defined in Section 1, and instants of τ_i-idleness or intervals of τ_i-idleness, during which tasks of priority less than P_i (or no task at all) are processed. The execution of τ_i takes place solely within the τ_i-busy period segments. Consequently, when checking any timing requirements associated with τ_i, attention can be restricted to τ_i-busy periods.

The definition of a τ_i-busy period permits two such periods to be "back-to-back." In this case, we consider the interval of lower priority processing or idleness to exist, but to be of zero length. One could require that the τ_i-busy period have strictly positive intervals of lower priority processing (or idleness) on each side; however, this can lead to unnecessary deadline checking.

We begin our schedulability analysis by looking only at the final subtask deadlines, $D_{im(i)}$, $1 \le i \le n$. We later discuss the deadlines for the earlier subtasks in the task set.

4.2 Schedulability of the Canonical Form

We next show that for purposes of checking its timing requirements, τ_i can be reduced to its canonical form. Consider any task τ_i that has two consecutive subtasks τ_{ij} and τ_{ij+1} with strictly decreasing priorities $P_{ij} > P_{ij+1}$. We consider a modified version of τ_i, τ'_i, obtained by reducing the priority of τ_{ij} to $P'_{ij} = P_{ij+1}$. We next show that the completion times of τ_i and of the subtasks τ_{ik}, $j+1 \le k \le m(i)$, are unchanged in the modified version.

Theorem 1: Suppose τ_i has two consecutive subtasks τ_{ij} and τ_{ij+1} of strictly decreasing priority $P_{ij} > P_{ij+1}$. Then for any task set phasing, the completion times of a task τ_i and its subtasks τ_{ik}, $j+1 \le k \le m(i)$ are unchanged if the priority of τ_{ij} is reduced to P_{ij+1}, assuming all equal priority segments are executed in the same relative order.

Proof: Let the execution sequence for the task set be given. For any single job of τ_i (τ'_i) let b_{ik} (b'_{ik}) and f_{ik} (f'_{ik}) denote the start time and the completion time of τ_{ik} (τ'_{ik}). Notice that the completion time of τ_{ik} (τ'_{ik}) is the activation time of τ_{ik+1} (τ'_{ik+1}). The activation time is the instant at which a particular task or subtask becomes ready to execute. Preemption may cause the task to start executing at an instant later than the activation; we call this the start time. Assume that the two versions of this job of τ_i are activated at the same time. The execution sequences will then be identical up to f_{ij-1}. If we prove $b_{ij+1} = b'_{ij+1}$, then the execution sequences will be identical after b_{ij+1}, and the result will follow, assuming that segments with equal priority are processed in the same order in the original and in the modified task set. For the original task set, the interval $[f_{ij-1}, b_{ij+1}]$ consists of the execution of τ_{ij} and other tasks with priority higher than or equal to the priority of τ_{ij+1}. At time b_{ij+1}, all work of equal or higher priority than τ_{ij+1} will have just finished, and that subtask can begin. If we now reduce the priority of τ_{ij} to that of τ_{ij+1}, then exactly the same work will be done during this interval, although possibly in another order. At time b_{ij+1}, all work of equal or higher priority than τ_{ij+1} will also have just finished, and that subtask will begin execution. Consequently, $b_{ij+1} = b'_{ij+1}$, and the results hold. Moreover, since the activation time of both versions is the same at the beginning of any τ_i-busy period, this result holds for the entire busy period.

The above theorem can be used to simplify the determination of whether a particular task meets its final deadline. If any consecutive subtasks are of strictly decreasing priority, we can reduce the priority of the first to that of the second and merge these two into a single segment. By applying this theorem to all such consecutive subtasks, we can reduce the task to *canonical form*, as defined in Section 2.

4.3 Worst-Case Phasing

Recall that $P_i = \min(P_{ij}, 1 \le j \le m(i))$ and that the execution sequence arising from any choice of task phasings will create an alternating sequence of τ_i-busy periods and periods of less than priority level P_i execution. We select a τ_i-busy period and seek to determine the task phasings that will create the longest response time for subtask $\tau_{im(i)}$.

Theorem 2: The longest response time for $\tau_{im(i)}$ is found during a τ_i-busy period initiated by τ_i.

Proof: Suppose that τ_i does not initiate a τ_i-busy period, so there exists an interval of execution of length $A > 0$ prior to the first initiation of τ_i during which subtasks of priority level P_i or higher are continuously executed. If the initiation time of τ_i were moved back by A, then $\tau_{im(i)}$ could not start its execution any earlier than it did with its original initiation time, because all subtasks that were initiated during the period of length A would still need to be executed before $\tau_{im(i)}$ can begin execution. Thus, the response times for all jobs of $\tau_{im(i)}$ would be increased by A. Thus, the situation in which τ_i does not initiate its τ_i-busy period may not give the worst-case response time.

According to Theorem 1, one can reduce τ_i to its transformed canonical form. The resulting task, τ'_i, will consist of $m'(i)$ subtasks $\tau'_{i1}, \ldots, \tau'_{im'(i)}$ having priorities $P_i = P'_{i1} < \ldots < P'_{im'(i)}$. The transformed canonical form is useful for reasoning about the phasing of the rest of the tasks that creates the worst-case response. This phasing will depend on the priority levels of each task, compared to the priority of the first segment of the canonical form task, resulting in the classification of tasks by type that appeared in Section 2.

Clearly, type 5 tasks (lower priority) cannot be executed during a τ_i-busy period, and therefore cannot influence the response time of τ_i. We now seek the phasing of the 4 remaining task types that will maximize the longest response time for τ_i during the busy period.

4.3.1. Type 1 Tasks (H): The phasing of a type 1 task that will create the largest response time for any job of τ_i in a τ_i-busy period is to have such a task initiated at the same instant that τ_i is initiated. To see this, suppose that the first job of such a task within the τ_i-busy period is initiated at time $I > 0$. The cumulative processor demands from this task during $[0,t]$ for every $t \in [0,b)$ (where b represents the end of the busy period) monotonically increase as I decreases towards 0. Consequently, these demands are maximized uniformly over time by choosing I = 0; this maximizes the completion time of all subtasks and all jobs of τ_i in the busy period. Therefore, setting $I = 0$ for any type 1 task will lead to the longest response time of τ_i.

4.3.2. Type 4 Tasks ($(LH)^+L^0$): All H segments in type 4 tasks are preceded by an L segment that has priority lower than P_i. This means that, as tasks cannot suspend themselves, at most one of the H segments can be processed during the τ_i-busy period. Furthermore, only one segment of one of the type 4 tasks can be processed during this entire busy period. For τ_i to initiate its own busy period, and for a segment of a type 4 task to execute during the busy period, the type 4 task must be executing one of its high-priority segments when τ_i is initiated. It can finish that segment, but no further processing is possible until the busy period ends. Thus, to determine the worst-case phasing for all type 4 tasks, we must consider all of the H segments of all of these tasks and select the one with the longest processing requirement. That segment should be starting just as τ_i is initiated. The phasing of the other tasks is irrelevant. We let B' denote the length of this longest segment.

4.3.3. Type 2 ($(HL)^+$) and Type 3 ($(HL)^+H$) Tasks: Type 2 tasks are similar to type 4, in that only one segment of the task can be processed during the busy period. The difference is that by phasing each type 2 task to be initiated at time 0, each will contribute a processing requirement that must be finished during the busy period, in contrast to the type 4 tasks, which collectively contribute only a single processing requirement. However, one of the type 2 tasks may have a long blocking segment, so long that it contributes more processing requirements than its initial segment would contribute; we have to determine if this is the case.

Type 3 tasks are the most complicated to phase. Only part of a type 3 task can be processed during the τ_i-busy period, but that part can be an inner H segment, the last H segment, the first H segment, or the last followed by the first H segment. It should be noticed, however, that the inner H and final H segments are blocking terms and, as tasks do not suspend themselves, at most one blocking term chosen from the type 2, 3, and 4 tasks can be executed during the busy period.

To determine which task, if any, of types 2 and 3 acts with a blocking segment, and therefore has to be phased with τ_i at one of its inner or final H segments, we can follow the next procedure. For each type 2 and type 3 task, let W be the computational requirement of the initial H segment, U be the requirement of the longest inner H segment (if any) and V be the computation requirement of the final H segment (for type 3 tasks). If there is no inner blocking segment then set U=0, and if there is no final blocking segment then set V=0.

Assume that each type 2 or type 3 task is initiated at the same time as task τ_i. Now consider the two other possible phasings:

1. Initiating the longest inner H segment at the same time as τ_i will increase τ_i's response times by $U-W-B'$, which is the marginal gain in total computational requirement from using this inner H segment.
2. For type 3 tasks, initiating the final H segment at the same time as τ_i may increase τ_i's response times by $V-B'$. For type 2 tasks this value is negative or zero (because V=0), and will be disregarded in the next steps. Note that this is a pessimistic estimate in that it assumes that the final segment serves as a blocking segment and the initial segment is a preempting segment. However, the busy period may end before the initial segment can preempt.

For each type 2 and type 3 task, we calculate the gain in latency as $max[\,(U-W-B'),(V-B')\,]$, and we choose the task with the largest value. If this value is negative, then all of these tasks should be initiated at the same time as τ_i, and the blocking term would be $B_i=B'$. If the largest value is positive, then either the inner blocking or final blocking segment should be selected. If $U-W \geq V$, then the inner segment is chosen, and $B_i=U$; in this case the initial H segment does not contribute a preemption effect. If $U-W < V$ then the final segment is chosen, we set $B_i=V$, and this task also remains as a singly preemptive task. All the other tasks should be initiated at the same time as τ_i.

4.4 Other Subtask Deadlines

The preceding analysis was carried out under the assumption that each task had only a single deadline at the end of its final subtask. We now wish to allow additional subtask deadlines, for example τ_{ij} having a deadline D_{ij}. Clearly, one must check the deadlines of each of these subtasks throughout the τ_i-busy period for the worst-case phasing developed earlier. However, this is not sufficient, because it will only ensure that jobs of subtasks after the first job in the busy period will meet their worst-case timing requirement. It does not guarantee the first job, because the phasing may not be worst for the first job of the subtask. The reason for this is that the phasing developed was based on the τ_i-busy period in which only activity of priority P_i or higher is executed, P_i being the minimum priority of all of the subtasks of τ_i. However, the appropriate priority level against which to create a longest response time phasing for the first job should be based upon a possibly higher level priority, $P_i(j) = \min(P_{ik}, 1 \leq k \leq j)$. This is the same as thinking of a τ_i^j-busy period, where task τ_i^j is a task composed of the first j subtasks of τ_i.

Theorem 3: The deadline, D_{ij}, of τ_{ij} will be met under all task phasings provided:

1. The deadline of the first job of τ_{ij} is met under the worst-case phasing for a τ_i^j-busy period, and
2. The deadline of all jobs of τ_{ij} after the first are met during a τ_i-busy period with worst-case phasing for the minimum priority level of that busy period.

Proof: For each subtask processed during the τ_i-busy period, we record the minimum priority of it and all subtasks of τ_i processed before it during this busy period. This minimum priority is defined as $P_i(j)$ for the first job of each subtask of τ_i and will be P_i for all subtasks of second and later jobs. The proof has two parts: finding the longest response time for jobs with minimum priority P_i and finding the longest response time for jobs with minimum priority $P_i(j) > P_i$.

The case of deadlines for subtasks with minimum priority P_i follows easily from the reasoning given in Section 4.3. Specifically, type 5 tasks can be ignored, type 1 tasks should be phased to maximize preemption as should any type 2 and type 3 tasks chosen for preemption. The largest blocking term derived from type 2, 3, and 4 tasks will also be the same, because a subtask of priority level P_i prior to the subtask in question is being blocked to the maximum extent, and this maximum blocking time will also delay all subsequent subtasks of τ_i. Consequently, one needs only to check the deadlines of all subtasks during the τ_i-busy period using the worst-case phasing derived in Section 4.3.

Subtasks τ_{ij} of the first job of τ_i with $P_i(j)$ greater than P_i must be handled separately, because their critical instant phasing can be different. One must now derive the worst-case phasing described in Section 4.3 for type 1 - 4 tasks, except the priority level P_i must be replaced by $P_i(j)$. Consider first the subtask τ_{ik} with the largest k among those for which $P_i(k) > P_i$. We wish to determine the longest response time for its first job. To do this, we ignore any later subtasks by considering only the task τ_i^k, and reduce this truncated task to its canonical form. Note that the minimum priority is now $P_i(k)$ and the other $n-1$ tasks must be reclassified into the five types with respect to $P_i(k)$. Once this is done, the arguments of Section 4.3 can be used to derive the worst-case phasing, and the longest response time of τ_{ik} can be computed. One now picks the subtask $\tau_{ik'}$, $k'<k$, with the next largest k' among those with $P_i(k') > \min(P_1, \ldots, P_k)$, truncates τ_i there to obtain $\tau_i^{k'}$, reduces it to canonical form, classifies the other tasks, and invokes the worst-case phasing to derive its worst-case response time. This process continues until all deadlines of first jobs have been checked. Thus conditions (1) and (2) are sufficient for all subtask deadlines to be met.

4.5 A Blocked-at-Most-Once Property

In Section 4.3 we described the procedure for calculating the latency due to internal and final H segments of other tasks, or in other words any H segment that is preceded by an L segment. Our procedure for finding the worst-case phasing implicitly used a "blocked-at-most-once" property for determining the effect of internal and final H segments. We will refer to internal and final H segments as blocking segments. We will refer to initial H segments as preempting segments. Restating this property more formally:

Lemma 4: There can be no more than one blocking segment within a τ_i-busy period.

Proof: Let task τ_i or a preempting segment (with a priority greater than or equal to P_i) start the τ_i-busy period. Since all blocking segments are preceded by an L segment and the L segment cannot preempt the τ_i-busy period, the blocking segment will not be able to execute until after the completion of the τ_i-busy period. Therefore the only way a blocking segment can be part of the τ_i-busy period is if the blocking segment starts the busy period.

Let a blocking segment start the τ_i-busy period. Once again, since blocking segments are preceded by L segments, a second blocking segment will not be able to execute until after the completion of the τ_i-busy period.

Two situations in which blocking segments naturally occur in practice are non-preemptible sections and emulation of the Priority Ceiling Protocol for task synchronization. These are described next.

4.6 Non-preemptible Sections

A non-preemptible section is typically implemented through system calls that start and later end the interval where preemption is to be disallowed. The system call to start the non-preemptible section is executed at the (base) priority of the subtask. The non-preemptible section can be modeled as a new subtask of extremely high execution priority. The non-preemptible section is ended with a system call that returns the priority to the original base priority. Therefore the original segment in which the non-preemptible section is embedded is transformed into three segments.

Given this model for non-preemptible sections, we would like to determine what effect they have on the schedulability of any given task. To understand this we must understand how non-preemptible sections affect each type of task discussed in Section 4.3. The effects of a non-preemptible section within a particular segment are different depending on the kind of segment:

1. When the non-preemptible section is embedded in an H segment it has no effect on the schedulability of the task being analyzed, because the resulting segments still behave as H segments.
2. When the non-preemptible section is embedded in an L segment, this segment is transformed into an LHL segment.

Consequently, non-preemptible sections have no effect on type 1 (H) tasks. For type 2 ($(HL)^+$) tasks, an L segment with an embedded non-preemptible section will be transformed into an LHL sequence. Since this was already preceded by an H segment, the transformed task remains in this class. The same reasoning holds for type 3 ($(HL)^+H$) and type 4 ($(LH)^+L^0$) tasks. Type 5 (L) tasks will be transformed into type 4 tasks. Using Lemma 3, we can see that at most one non-preemptible section can delay the completion of any given task being analyzed.

4.7 Synchronization

When two or more tasks share a resource, such as data, that needs to be accessed in a mutually exclusive manner, then these tasks must be forced to synchronize by using the resource in a critical section. One effective synchronization protocol is the Priority Ceiling Protocol (PCP) [14]. The protocol dictates the priority at which a critical section executes and specifies conditions that must be true before a critical section can be entered. An approximation of the PCP can be easily implemented at the application-level by ensuring that a critical section executes at a sufficiently high priority. In the case where the entire task executes at a single priority level and also accesses data in a critical section, PCP emulation requires the critical section to be executed at a priority level slightly higher than any task that accesses the shared resource [12]. Effectively this changes the task from one that had a single segment to one that has three segments where the middle segment executes at a fixed but elevated priority. In the case where the original task has multiple segments, assume that the critical section is executed within a single segment of the original task. Divide that segment into three segments: one before, one during and one after the critical section. The critical section will execute at a higher priority than all subtasks that access the shared resource. Using the same reasoning that was used for non-preemptible sections, we can see that at most one critical section or one non-preemptible section can delay the completion of any given task being analyzed.

4.8 Sporadic Tasks and Aperiodic Servers

The theory in this paper is restricted to the case of periodic tasks which are composed of subtasks, each of which has a hard deadline. This theory can be easily extended to allow for sporadic tasks, tasks which do not necessarily arrive periodically but which have a minimum interarrival time, a fixed worst case computation time and a hard deadline. The sporadic tasks may be composed of serially executed subtasks. Sporadic tasks can be included into the scheduling analysis presented in this paper by treating each sporadic task as a periodic task with its period given by the minimum interarrival time. The analysis also permits the inclusion of aperiodic server algorithms, such as the *sporadic server* [16]. Aperiodic servers are used to provide high priority execution time to service aperiodic requests in a manner that is predictably invasive on the schedulability of lower priority tasks. The high priority of the server task can provide very fast response times to aperiodic tasks, especially when those tasks have relatively short execution times.

4.9 Assigning Subtask Priorities

In this section, we consider an important special case in which each task τ_i is composed of subtasks τ_{ij} whose deadlines D_{ij} satisfy $D_{i1} \le D_{i2} \le \ldots \le D_{im(i)} \le T_i$. Thus all subtask deadlines are at or before the end of the task period.

A good starting point for assigning priorities is to use a deadline monotonic priority assignment. That is, assign $P_{ij}>P_{kl}$ if and only if $D_{ij}<D_{kl}$ where ties between subtasks with equal deadline are broken arbitrarily. We will later prove that this priority assignment is optimal among all fixed priority assignments for task sets whose subtask priorities are required to be non-ascending. It is important to note that the canonical form for subtask τ_{ij} consists of a single subtask with priority P_{ij} and computation requirements:

$$C_i^j = \sum_{k=1}^{j} C_{ik}$$

The assumption $D_{km(k)} \le T_k$ allows one to infer that only the deadline of the first job of each subtask needs to be checked under its worst case phasing to ensure that all jobs of a subtask meet their deadlines. Assuming all subtask priorities are distinct, a τ_i^j-busy period ends with the completion of the first job of that subtask, thus eliminating the need for further checking.

For this case, we can most easily express the schedulability conditions if we relabel all the subtasks, writing them in order of decreasing priority. Thus the subtasks, become

$$\{\, s_k \mid k=1,N \,\},\ \textit{where}\ N = \sum_{i=1}^{n} m(i)\ .$$

Subtask s_k has priority P_k with $P_k \ge P_{k+1}$, computation requirement C_k, period T_k and deadline D_k. In this case, the necessary and sufficient condition for subtask s_k schedulability under all task phasing is:

$$\min\Big(\, 0 \le t \le D_k \;\Big|\; \sum_{j=1}^{k} \lceil t/T_j \rceil (C_j/t) \,\Big) \le 1\ .$$

The necessary and sufficient condition for task set schedulability is for the above inequality to hold for all s_k, $1 \le k \le N$. Equivalently,

$$max\left(\min\left(0 \le t \le D_k \,\Big|\, \sum_{j=1}^{k} \lceil t/T_j \rceil (C_j/t) \right)\right) \le 1, \quad 1 \le k \le N .$$

Since the precedence relations in the original task set are enforced by the priority assignment in the transformed task set, we can now use the result of Leung and Whitehead [7] to prove the optimality of the deadline monotonic priority assignment for this transformed task set $\{s_k\}$.

4.10 Deadline Monotonic Scheduling Conditions

We can use the inequalities in the previous subsection as exact schedulability tests; however, it is useful to develop sufficient conditions for a particular subtask to be schedulable. For example, Liu and Layland proved that if n periodic tasks have total utilization no greater than $n(2^{1/n}-1)$, then all deadlines will be met using the rate monotonic priority assignment. Lehoczky [5] extended the Liu and Layland analysis to allow arbitrary deadlines. Using Lehoczky's methods, one can prove that if n tasks are assigned fixed priorities, τ_n has lowest priority, period T_n and deadline D_n, and each (higher priority) task τ_i has a period T_i that is less than D_n, then the τ_n is schedulable if

$$\sum_{i=1}^{n} U_i \le n((2\Delta)^{1/n} - 1) + 1 - \Delta$$

$$where \;\; U_i = C_i/T_i, \; and \; \Delta = D_n/T_n$$

This idea can be used to develop sufficient conditions for task sets where subtasks are assigned deadline monotonic priorities and all deadlines are before the end the period. To check the schedulability of τ_{ij}, we reduce that subtask to canonical form. This results in a task with a single subtask of priority P_{ij}, period T_i, deadline D_{ij} and computation requirement C_i^j. All other tasks must be classified into one of five categories; however, in this case only types H, HL, and L are possible when τ_{ij} is reduced to canonical form. Now we have to further categorize tasks as being singly preemptive (SP) or multiply preemptive (MP). Singly preemptive tasks will be able to preempt the canonical subtask only once before its deadline. Multiply preemptive tasks will be able to preempt the canonical subtask more than once before its deadline. The type H tasks with period shorter than D_{ij} are MP tasks. The remainder of the H tasks are SP tasks. The HL tasks are all SP tasks. The L tasks can be ignored.

If we let U_{MP} denote the total utilization of all MP tasks; let C_{SP} denote the sum of the execution times of the initial H segments of all SP tasks; let $\Delta = D_{ij}/T_i$; and let m be one plus the number of MP tasks, then a sufficient condition for subtask τ_{ij} to meet its deadline is

$$U_{MP} + C_i^j/T_i + C_{SP}/T_i \le m((2\Delta)^{1/m} - 1) + 1 - \Delta$$

This test can be applied to every subtask. Those subtasks that satisfy the condition are schedulable. Those subtasks not satisfying the condition must be checked using the exact test in the previous section.

4.11 Improving Schedulability

The example presented in Figure 1 was designed to show the added scheduling complexity that can occur when tasks are composed of subtasks that are executed at different fixed priority levels. This example also shows a scheduling benefit that can accrue from such a task structure. Using ordinary rate monotonic scheduling, a task set with two tasks: $C_1 = 4$, $T_1 = 10$ and $C_2 = 6$, $T_2 = 14$, is schedulable, but fully utilizes the processor with total utilization of .829, essentially equal to the Liu and Layland bound of .828 for two tasks. However, the example in Section 1.2 modifies task two to have two subtasks of differing priority. This modified task set has a total utilization of .971 and is schedulable using a fixed priority algorithm. Thus, we see that we can increase the fixed priority schedulability of a task set by decomposing one or more of the tasks into subtasks that are executed at modified priority levels, and this can offer an increase in schedulability over that provided by the simple rate monotonic algorithm.

This paper is restricted to methods by which the schedulability of complex tasks can be determined for any fixed priority scheduling algorithm. The question of determining optimal priority assignments in the general case is beyond the scope of this paper. In the previous subsection we discussed an optimal priority assignment for a very special case. In this subsection we present a simple and very interesting result for two tasks; by dividing the one having the largest period into two subtasks, one with the lowest priority and one with globally highest priority, 100% schedulability can be attained. Specifically, given any two periodic tasks with $T_1 < T_2$ and utilizations $U_1 + U_2 = 1$, one should decompose τ_2 into two subtasks with $C_{21} = C_2 - (T_1 - C_1)$, $C_{22} = T_1 - C_1$. τ_1 has intermediate priority, τ_{21} has lowest priority and τ_{22} has highest priority. The resulting task set is schedulable, the C's and T's are arbitrary and the task set has 100% utilization.

Theorem 5: Given two periodic tasks τ_1 and τ_2, with the following characteristics:

- execution times are C_1 and C_2,
- periods are $T_1 \leq T_2$,
- $C_1/T_1 + C_2/T_2 = 1$

there exists a decomposition of $C_2=C_{21}+C_{22}$ with priorities $P_{21}<P_1<P_{22}$ such that the task set is schedulable

Proof: First notice that the τ_2 is being transformed from an L task into an LH. Therefore τ_1 will need to be able to tolerate blocking due to the newly created H segment. We know that τ_1 can tolerate blocking segment of length T_1-C_1, so we use that as C_{22}. We will show that the following decomposition results in a schedulable task set.

$$C_{21} = C_2 - (T_1 - C_1)$$

$$C_{22} = T_1 - C_1$$

First we want to establish that C_{21} and C_{22} are non-negative and then we will show that both tasks are schedulable.

It's obvious that C_{22} is non-negative. We will now show that C_{21} is non-negative. We will use $C_1/T_1 + C_2/T_2 = 1$ implies $T_2 = C_2[T_1/(T_1-C_1)]$.

$C_{21} \geq 0$ *iff*
$C_2/(T_1-C_1) \geq 1$ *iff*
$C_2T_1/(T_1-C_1) \geq T_1$ *iff*
$T_2 \geq T_1$, *which holds by assumption*.

Therefore $C_{21} \geq 0$.

Task τ_1 is clearly schedulable since C_{22} was chosen carefully to ensure that it remained schedulable. The only issue is the schedulability of τ_2.

We must show that each job of task τ_2 meets its deadline. The j^{th} job of task τ_2 will meet its deadline at time jT_2 if the following two conditions hold:

1. $\lfloor jT_2/T_1 \rfloor C_1 + jC_2 \leq jT_2$, *and*
2. $\lfloor jT_2/T_1 \rfloor C_1 + (j-1)C_2 + C_{21} \leq \lfloor jT_2/T_1 \rfloor T_1$

The value $i = \lfloor jT_2/T_1 \rfloor$ denotes the number of complete periods of task τ_1 that fit into the interval $[0,jT_2]$. Since τ_1 is already guaranteed to not overrun, then at least i jobs of τ_1 and j jobs of τ_2 must complete before jT_2 if job j of task τ_2 is to meet its deadline. This is stated in condition 1 above. The second condition states that i jobs from τ_1, (j-1) jobs of τ_2 and the first subtask of the j^{th} job of τ_2 should complete before the $(i+1)^{st}$ job of τ_1 is initiated. If this is true then the high priority of τ_{22} will allow it to finish before the $(i+1)^{st}$ job of τ_1 executes. Condition 1 then guarantees that it meets its deadline.

Showing the first condition to be true follows:

$$\lfloor jT_2/T_1 \rfloor C_1 + jC_2 \le jT_2(C_1/T_1) + jC_2$$

$$= jT_2(\,(C_1/T_1) + jC_2/jT_2\,) = jT_2$$

Showing the second condition to be true follows:

$$iC_1 + (j-1)C_2 + C_{21}$$

$$= iC_1 + (j-1)C_2 + C_2-(T_1-C_1)$$

$$= (i+1)C_1 + jC_2 - T_1$$

$$= (i+1)T_1\Big[\,((i+1)C_1)/((i+1)T_1) + jC_2/((i+1)T_1)\,\Big] - T_1$$

$$\le (i+1)T_1[\,C_1/T_1 + jC_2/jT_2\,] - T_1 = iT_1$$

Thus, the j^th job of τ_2 meets its deadline and τ_2 is schedulable.

As an example, consider the task set depicted in Figure 1 with C_2 increased to 8.4. The resulting task set has 100% utilization, yet it is schedulable if the second task is broken into two subtasks with C_{21}=2.4, C_{22}=6, and τ_{22} is given highest priority. Ordinary rate monotonic scheduling can meet the timing requirements only if $C_2 \le 6$.

5 Conclusions

Even when application-level tasks are assigned fixed priorities, the actual priority structure of a realistic system can be much more complex. Characteristics of the operating system and underlying hardware impact the priority structure and consequently the timing behavior of the system. For example, the task dispatching mechanism of the operating system, the interrupt architecture of the processor, synchronization protocols, and intertask communication mechanisms all contribute to the system's actual timing behavior. In order to accurately predict system behavior, these effects should be included in the schedulability equations that model the system's behavior.

This paper offers a generalized model of fixed priority scheduling that provides a theoretical framework for analyzing task sets scheduled through a fixed priority preemptive scheduler, where each task is comprised of a number of subtasks, each executing at a different priority level. For a set of tasks with a complex priority structure, we offer a method of analysis with an underlying theoretical foundation. For simple task sets, our contribution is a formalization of techniques currently being used [3]. Furthermore, the method shows that an increase in schedulability can be achieved by taking advantage of the high-priority execution of the final subtasks of a task.

Another very important highlight of this method is that it provides some simple techniques for reasoning about time in systems with varying priorities. The fact that tasks can be reduced to a canonical form simplifies analysis and allows one to easily reason about worst-case phasing. It also simplifies analysis by allowing the classification of tasks according to the priority of their first subtask. We feel that in practical problems the algorithm described in Section 2 is easily implemented and runs efficiently; however, the worst case complexity of this algorithm is an open question.

References

1. Borger, M. W., Klein, M. H., and Veltre, R. A. "Real-Time Software Engineering in Ada: Observations and Guidelines". *Software Engineering Institute Technical Review* (1988).

2. Goodenough, J. B., and Sha, L. "The Priority Ceiling Protocol: A Method for Minimizing the Blocking of High Priority Ada Tasks". *Proceedings of the 2nd International Workshop on Real-Time Ada Issues* (June 1988).

3. Klein, M. H., and Ralya, T. An Analysis of Input/Output Paradigms for Real-Time Systems. Tech. Rept. CMU/SEI-90-TR-19, Software Engineering Institute, July 1990.

4. Lehoczky, J. P., and Sha, L. "Performance of Real-Time Bus Scheduling Algorithms". *ACM Performance Evaluation Review, Special Issue 14*, 1 (May, 1986).

5. Lehoczky, J.P. "Fixed Priority Scheduling of Periodic Task Sets with Arbitrary Deadline". *IEEE Real-Time System Symposium* (1990).

6. Lehoczky, J. P., Sha, L., Strosnider, J.K., Tokuda, H. Fixed Priority Scheduling for Hard Real-Time Systems. In *Foundations of Real-Time Computing: Scheduling and Resource Management*, van Tilborg, Andre and Koob, Gary M., Ed., Kluwer Academic Publishers, 1991, pp. 1-30.

7. Leung, J. and Whitehead, J. "On Complexity of Fixed-Priority Scheduling of Periodic Real-Time Tasks". *Performance Evaluation 2, 237-50* (1982).

8. Liu, C.L., and Layland, J.W. "Scheduling Algorithms for Multi-Programming in a Hard Real-Time Environment". *Journal of the Association for Computing Machinery Vol. 20*, 1 (January 1973), pp. 46-61.

9. Rajkumar, R., Sha, L., and Lehoczky, J.P. "Real-Time Synchronization Protocols for Multiprocessors". *IEEE Real-Time Systems Symposium* (December 1988).

10. Rajkumar, R. "Real-Time Synchronization Protocols for Shared Memory Multi-Processors". *Proceedings of The 10th International Conference on Distributed Computing* (1990).

11. Sha, L., Rajkumar, R., Lehoczky, J. and Ramamritham K. "Mode Change Protocols for Priority-Driven Preemptive Scheduling". *The Journal of Real-Time Systems Vol. 1* (1989), pp. 243-264.

12. Sha, L. and Goodenough, J. B. "Real-Time Scheduling Theory and Ada". *IEEE Computer Vol. 23, No. 4* (April 1990).

13. Sha, L., Rajkumar, R., and Lehoczky, J. P. "Real-Time Scheduling Support in Futurebus+". *IEEE Real-Time Systems Symposium* (1990).

14. Sha, L., Rajkumar, R., and Lehoczky, J. P. "Priority Inheritance Protocols: An Approach to Real-time Synchronization". *IEEE Transactions on Computers* (Sept. 1990).

15. Sha, L., Klein, M. H., and Goodenough, J. B. Rate Monotonic Analysis for Real-Time Systems. In *Foundations of Real-Time Computing: Scheduling and Resource Management*, van Tilborg, Andre and Koob, Gary M., Ed., Kluwer Academic Publishers, 1991, pp. 129-155.

16. Sprunt, B., Sha, L., and Lehoczky, J.P. "Aperiodic Task Scheduling for Hard Real-Time Systems". *The Journal of Real-Time Systems* , 1 (1989), pp. 27-60.

Table of Contents

List of Figures

List of Tables