Technical Report Number 2007-536

Penalty Minimization in Scheduling a Set of Soft Real-Time Tasks*

Arezou Mohammadi and Selim G. Akl
School of Computing
Queen's University
Kingston, Ontario, Canada K7L 3N6
E-mail:{arezoum, akl}@cs.queensu.ca

June 13, 2007

Abstract

A soft real-time task is one whose completion time is recommended by a specific deadline. However, should the deadline be missed, such a task is not considered to have failed; only the later it finishes, the higher the penalty that is paid. For a set of soft real-time tasks that are to be scheduled on a single machine under overload conditions, our objective is to minimize the total penalty paid. This optimization problem is NP-hard. In this paper, we prove a number of properties of any optimal scheduling algorithm for the problem. Then, we provide a number of heuristic algorithms which satisfy the properties obtained herein. Numerical simulations are presented to compare the penalty to be paid by the algorithms. We also determine an upper bound for the optimal solution to the problem. Numerical results that compare the upper bound with the optimal solution and the heuristic algorithms are provided.

Keywords: Soft Real-Time Tasks, Optimal Scheduling Algorithm, Upper Bound, Heuristic Algorithms, Overload Conditions, Penalty Minimization.

*This work was supported in part by Natural Sciences and Engineering Research Council (NSERC) of Canada.

1 Introduction

The purpose of a real-time system is to produce a response within a specified time-frame. In other words, for a real-time system not only the logical correctness of the system should be satisfied, but also it is required to fulfill the temporal constraints of the system. Although missing deadlines is not desirable in a real-time system, *soft real-time tasks* could miss some deadlines and the system will still work correctly while certain penalties will have to be paid for the deadlines missed. In this paper, we focus our attention on scheduling of a set of soft real-time tasks.

The problem under study in this paper occurs in overload conditions where it can not be guaranteed that all tasks can meet their deadlines. In this case, it is unavoidable to miss some deadlines, while we aim to minimize the penalty that should be paid.

Consider a system that consists of a set of soft real-time tasks, $T = \{\tau_1, \tau_2, ..., \tau_n\}$. Task τ_i is a *soft real-time* task, meaning that the later the task τ_i finishes its computation after its deadline, the more penalty it pays. A release time r_i, an execution time e_i and a deadline d_i are given for each task $\tau_i \in T$ (see Section 2 for the definition of these terms). The finishing time of each task $\tau_i \in T$, denoted by F_i, depends on the scheduling algorithm which is used to schedule the execution of the tasks [19]. Suppose that the tasks are scheduled by some scheduling algorithm A. A penalty function $\varphi(\tau_i)$ is defined for the task. If $F_i \leq d_i$, $\varphi(\tau_i) = 0$; otherwise $\varphi(\tau_i) > 0$. The value of $\varphi(\tau_i)$ is a non-decreasing function of $F_i - d_i$. The penalty function of a given scheduling algorithm A for a given set T is denoted by $\varphi(T) = \sum_{i=1}^{l} \varphi(\tau_i)$.

The fact known about our problem, as is true for most of the problems in this class, is that it is NP-hard [11]. Recently, there has been a lot of progress in the design of approximation algorithms for a variety of scheduling problems in the aforementioned class [12, 2, 21, 3, 1, 7, 8, 14]. Also, in the real-time literature, several scheduling algorithms have been proposed to deal with overloads. For instance, one may refer to [6, Chapter 2] and the references therein. A relevant and recent work is [20], in which the problem is studied for the special case of non-preemptive tasks (see Section 2). In this paper, we address a more general problem; namely, the scheduling of a set of preemptive soft real-time tasks (see Section 2) where the objective function is to minimize the total penalties that should be paid for the deadlines missed.

In this paper, we formally define the problem and prove a number of properties of any optimal scheduling algorithm for the problem. Then, we derive a number of heuristic algorithms which hold the properties obtained herein. The heuristic algorithms differ in the way that the tasks priorities are assigned. These algorithms assign priorities by using functions of task execution times, penalty factors or deadlines. We present simulation results and compare the performances of the proposed algorithms. Finally, we derive a tight upper bound for the optimal solution. Since the running time of finding an optimal solution grows exponentially with the number of tasks, we compare the upper bound with the optimal solution for small sets of soft real-time tasks. We also compare the upper bound with the heuristic algorithms provided in the paper.

The remainder of this paper is organized as follows. We introduce the terminology in Section 2. In Section 3, we formally define the problem to be solved. In Section 4, we derive and prove some of the properties of any optimal scheduling algorithm for the problem. Then, in Section 5, we provide a class of heuristic algorithms, present simulation results and compare the performance of the proposed algorithms. In Section 6, we find an upper bound for the objective function, present the simulation results and compare the upper bound with the optimal solution. We also compare the upper bound with the best heuristic algorithm provided in the paper. Section 7 contains the conclusions.

2 Terminology

For a given set of tasks the *general scheduling problem* asks for an order according to which the tasks are to be executed such that various constraints are satisfied. For a given set of real-time tasks, we want to devise a feasible allocation/schedule to satisfy timing constraints. The release time, the deadline and the execution time of the tasks are some of the parameters that should be considered when scheduling tasks on a real-time system. The timing properties of a given task τ_j, where $\tau_j \in T$, refer to the following [16, 18, 22, 9]:

- *Release time* (or *ready time* (r_j)): Time at which the task is ready for processing.
- *Deadline (d_j)*: Time by which execution of the task should be completed.
- *Completion time (C_j)*: Maximum time taken to complete the task, after the task is started.
- *Finishing time (F_j)*: Time at which the task is finished: $F_j = C_j + r_j$.
- *Execution time (e_j)*: Time taken without interruption to complete the task, after the task is released.
- *priority function ($f_{a,j}$)*: Priority of task τ_j is defined as relative urgency of the task. In this paper, we derive a number of heuristic algorithms which differ in the way that the tasks priorities are assigned. For each algorithm, we consider a *priority*

function $f_{a,j}$, where a is the name of the algorithm and τ_j is a task in the task set that should be scheduled by the algorithm. In other words, the priority function is the relative urgency of a task in a given algorithm.

- *Penalty factor (P_j)*: Penalty that should be paid per time unit after the deadline of task τ_i.
- *Makespan factor (α_j)*: Ratio of C_j to e_j, i.e., $\alpha_j = C_j/e_j$, where e_j and C_j are respectively the execution time of task τ_j and the completion time of the task in the schedule. This factor depends on schedule.

Other issues to be considered in real-time scheduling are as follows.

- *Periodic/Aperiodic/Sporadic tasks*

  Periodic real-time tasks are activated (released) regularly at fixed rates (periods). A majority of sensory processing is periodic in nature. Aperiodic real-time tasks are activated irregularly at some unknown and possibly unbounded rate. The time constraint is usually a deadline. Sporadic real-time tasks are activated irregularly with some known bounded rate. The bounded rate is characterized by a minimum inter-arrival period; that is, a minimum interval of time between two successive activations. The time constraint is usually a deadline [17, 15, 13, 5].

- *Preemptive/Non-preemptive tasks*

  In some real-time scheduling algorithms, a task can be preempted if another task of higher priority becomes ready. In contrast, the execution of a non-preemptive task should be completed without interruption once it is started [17, 16, 13, 5].

- *Fixed/Dynamic priority tasks*

  In priority driven scheduling, a priority is assigned to each task. Assigning the priorities can be done statically or dynamically while the system is running [10, 22, 16, 17, 5].

3 Problem Definition

Consider a set of n soft real-time tasks. There exists one processor. The tasks are preemptive, independent and aperiodic. For each task τ_i, we assume that r_i, e_i, P_i, and d_i, which are respectively the release time, execution time, penalty factor and deadline of the task, are known.

We define the penalty function of task τ_i as

$$\varphi(\tau_i) = (F_i - d_i)^+ P_i, \tag{1}$$

where $F_i = r_i + \alpha_i e_i$ is the finishing time of task τ_i, α_i is the makespan factor ($\alpha_i \geq 1$), and

$$(F_i - d_i)^+ = \begin{cases} F_i - d_i & \text{if } F_i - d_i > 0 \\ 0 & \text{otherwise.} \end{cases}$$

A slot is the smallest time unit.

The objective is to minimize $\sum_{i=1}^{n} \varphi(\tau_i)$. Therefore, we can formally express the objective function as follows. Let us define

$$x_{i,t} = \begin{cases} 1 & \text{if the processor is assigned to task } \tau_i \\ & \text{at time slot } t \\ 0 & \text{otherwise} \end{cases}$$

Our goal is to minimize the objective function

$$\sum_{i=1}^{n} (r_i + \alpha_i e_i - d_i)^+ P_i, \tag{2}$$

subject to the following conditions

$$\sum_{i=1}^{n} x_{i,t} = 1,$$

which means only one processor is working at any given time t, and

$$\sum_{t=1}^{\infty} x_{i,t} = e_i,$$

meaning that the total time slots assigned to any given task i over time is equal to its execution time.

As mentioned earlier, the problem defined in this section is known to be NP-hard. Thus, the only known algorithm for obtaining an optimal schedule requires time that grows exponentially with the number of tasks. It is beneficial to know the behavior of any optimal scheduling algorithm when the optimal order of priorities is provided. Knowing a number of properties of any optimal schedule for the problem will lead us to designing heuristic algorithms which in some properties behave the same as the optimal schedule. Also, it is desired to find an upper bound for the objective function which, unlike the optimal algorithm, it would be computationally feasible. The upper bound may also be useful for design and comparison purposes.

Figure 1: Status of processor in the proof of Lemma 1.

4 Properties of Optimal Solution

In this section, we prove a number of properties for any optimal schedule of the problem $\min \sum_{i=1}^{n}(r_i + \alpha_i e_i - d_i)^+ P_i$. For all of the following theorems we have one processor and a set of soft real-time tasks.

Lemma 1: In an optimal schedule, the processor is not idle as long as there exists a task to be executed.

Proof: We prove this lemma by contradiction. Suppose that in a given optimal schedule the processor is idle while all or a part of task τ_i is ready to be executed (see Figure 1).

There exists at least one other schedule which has a smaller value for the finishing time of task τ_i (i.e., $F_i^* \leq F_i$, where F_i^* denotes the new finishing time of task τ_i) and the finishing times of the other tasks in the system remain unchanged. This schedule will result in a smaller value for $\sum_{i=1}^{n}(F_i - d_i)^+ P_i$. The new schedule is achieved just by moving the starting point of the remaining part of task τ_i to the starting point of the idle time. This replacement does not affect the finishing times of the other tasks that are possibly in the system (see Figure 2). Therefore, the first algorithm (Figure 1) is not optimal. □

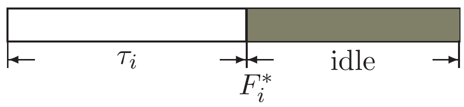

Figure 2: Status of processor for an optimal schedule in the proof of Lemma 1.

From the above lemma, it follows that if there is a task to be executed, the processor cannot be idle. In other words, if the processor is idle in any optimal schedule for our problem, there is no task to be executed during that time interval. Considering the fact that idle times are time intervals during which there is no task to be executed, it is enough that we prove the following theorems for time intervals during which there is no idle times (henceforth referred to as task-bursts). Since any schedule, including the optimal schedules, is a combination of idle times and task-bursts, the following theorems hold for any optimal schedule.

If the priorities of the tasks for the optimal scheduling are given, we could use the following theorems to achieve an optimal schedule whose running time grows polynomially with the number of tasks. Unfortunately, we do not have the priorities. However, we can define the concept of priority for any optimal schedule for the problem studied in this paper as follows. Suppose that task τ_j arrives while the processor is executing task τ_i. Task τ_j has a higher priority than task τ_i if preempting τ_i by task τ_j will result in a smaller value in $\sum_{i=1}^{n}(F_i - d_i)^+ P_i$. Therefore, a task with a higher priority than the current task will preempt the current task. In the following theorem, we prove that task τ_j preempts τ_i immediately after it arrives.

Theorem 1: Suppose that task τ_i is currently being executed by the processor. In an optimal schedule, when a higher-priority task τ_j arrives, task τ_i will be immediately preempted and task τ_j will begin execution.

Proof: We prove the theorem by contradiction. Suppose that in the optimal schedule, task τ_i can be preempted a while after task τ_j arrives (see Figure 3).

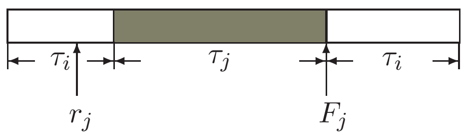

Figure 3: Status of processor in the proof of Theorem 1.

Because there exists at least one other schedule that has a smaller value for the finishing time of task τ_j (i.e., $F_j^* \leq F_j$) and the finishing time of the other tasks in the system remain unchanged, this schedule will result in a smaller value for $\sum_{i=1}^{n}(F_i - d_i)^+ P_i$ (see Figure 4 for an example). □

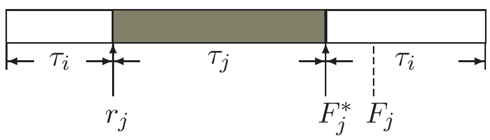

Figure 4: Status of processor for a more efficient schedule in the proof of Theorem 1.

Theorem 2: In an optimal schedule, if task τ_i is preempted by another task τ_j and the execution of task τ_j is not finished yet, task τ_j can not be preempted by task τ_i

Proof: We prove the theorem by contradiction. Suppose that in an optimal schedule, task τ_j can be preempted by task τ_i (see Figure 5).

On the other hand, there exists at least one other schedule that has a smaller value for the finishing time of task τ_j (i.e., $F_j^* \leq F_j$) and the finishing time of the other tasks

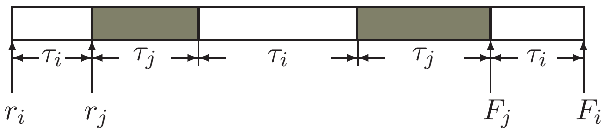

Figure 5: Task preemption in the proof of Theorem 2.

in the system remain unchanged. The new schedule will result in a smaller value for $\sum_{i=1}^{n}(F_i - d_i)^+ P_i$ (see Figure 6). $\square$

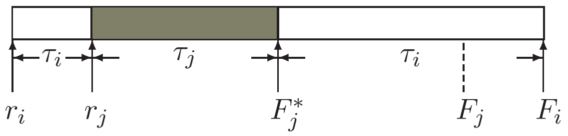

Figure 6: Example of a more efficient schedule for the proof of Theorem 2.

Theorem 3: In an optimal schedule, a task can not be preempted more than once by another task.

Proof: This is a direct consequence of Theorem 3. $\square$

5 Heuristic Algorithms

5.1 Algorithms Description

Although we can not find an optimal schedule in polynomial time, we have developed a set of heuristic algorithms for which the properties discussed in the theorems hold. *Class of algorithms*, which will hereafter be referred to as algorithm S, represents a set of algorithms which are the same, except in their *priority function*. For each algorithm in the class, we consider a *priority function* $f_{a,j}$, where a is the name of the algorithm and τ_j is a task in the task set that should be scheduled by the algorithm (refer to Table 1). We perform the scheduling based on the priority function. For an algorithm a and a pair of tasks τ_h and τ_j with respective priority functions $f_{a,h}$ and $f_{a,j}$, by $f_{a,j} \prec f_{a,h}$, we mean that task τ_h has a higher priority than task τ_j.

The priority function for each algorithm in the aforementioned class of algorithms can be found in Table 1. One of the data structures, which we consider for each algorithm, is a matrix called *Priorityq*. *Priorityq* contains all of the tasks that have arrived and either so far have not had a chance to be executed or preempted by a higher-priority task. The matrix is always kept sorted based on the priority of the tasks. For each task in the *Priorityq* we have the following information: release time, remaining execution time, penalty factor, deadline. An arbitrary algorithm a in the class of algorithms S has the following steps:

- When a new task τ_i arrives, we check if the processor is idle or busy. If it is idle, we load the task on it. Otherwise, we verify if the execution of the current task (τ_j) on the processor is finished.

  If execution of τ_j is finished, its termination time is computed. Then, the priority of task τ_i is compared with the task located on the top of the *Priorityq* matrix. If the priority of task τ_i is higher, the processor is allowed to execute τ_i. Otherwise, the position of task τ_i in *Priorityq* matrix is found and the task that has the highest priority in *Priorityq* matrix is taken.

  If execution of τ_j has not been finished yet, we check if $f_{a,j} \prec f_{a,i}$, where τ_j is the task that is running currently by the processor.

  - If $f_{a,j} \prec f_{a,i}$, then τ_i preempts τ_j. Task τ_j should be inserted into *Priorityq* at the appropriate position. Tasks are sorted in *Priorityq* based on their priorities. The position of task τ_j can be found by comparing its priority with the priority of the tasks in *Priorityq*.
  - If $f_{a,i} \prec f_{a,j}$ then the processor should continue executing task τ_j. Also, task τ_i should be inserted into its appropriate position in array *Priorityq*.

- When there in no other task to arrive, the algorithm computes the termination time of the current task. Then the task which is on the top of array *Priorityq* is popped up and its termination time is computed. The process continues until *Priorityq* becomes empty.

- Finally, the algorithm computes

$$\sum_{i=1}^{n}(F_i - d_i)^+ P_i,$$

  where F_i is the finishing time of task τ_i.

As mentioned earlier, we consider different priority functions each of which is used by one algorithm. For example, the priority function used in algorithm S_1 is $d_i P_i$ (see Table 1). This means the priority of task τ_i is higher than τ_j if $d_j P_j < d_i P_i$. The priority function used for each algorithm in this paper can be found in Table 1. All of the above algorithms are different versions of algorithm S, which we discussed earlier.

The running time of the algorithm is $O(n\log n)$, where n is the number of soft real-time tasks.

Table 1: Priority functions considered for each algorithm

Name of Algorithm	Priority Function
S_1	$d_i P_i$
S_2	$1/(d_i P_i)$
S_3	d_i
S_4	$1/d_i$
S_5	e_i
S_6	$1/e_i$
S_7	e_i/P_i
S_8	P_i/e_i
S_9	d_i/P_i
S_{10}	P_i/d_i
S_{11}	P_i
S_{12}	$1/P_i$
S_{13}	$e_i P_i$
S_{14}	$1/(e_i P_i)$

5.2 Simulation Results: Comparing Algorithms

We have implemented the algorithms for simulation purposes. Simulation conditions are as follows.

Each set of data includes n soft real-time tasks. For each task, we randomly generate r_i, e_i, d_i and P_i, which are the release time, execution time, deadline, and penalty factor, respectively. When randomly generating d_i, the condition that $e_i + r_i \leq d_i$ should hold.

We generate 20 different data sets with size n. We execute each of the aforementioned algorithms on each data set. We compute the average of the aggregations of the termination times of the 20 simulations for data set with size n. The simulation is done for the algorithms for $n = 1$ to 500, with step 10.

Each algorithm computes

$$\sum_{i=1}^{n} (F_i - d_i)^+ P_i,$$

where F_i is the finishing time of task τ_i.

As mentioned earlier, the heuristic algorithms diverge in the way that the task priorities are assigned. We observe in Figure 7 that in the algorithms where the priority assigned to each task is in non-decreasing order of P_i, non-increasing order of e_i, or non-increasing order of d_i, the total penalty to be paid is decreased. Algorithms S_3 and S_4 are those which assign the priority to each task in non-decreasing and non-increasing order of d_i, respectively. As a matter of fact, S_4 is the Earliest Deadline First (EDF) algorithm [17]. A scheduler for a set of tasks is said to be *feasible* if every execution of the program meets all its deadlines. It is shown in [17] that on uni-processor systems, if the EDF algorithm cannot feasibly schedule a task set on a processor, there is no other scheduling algorithm that can do so. However, we study the problem under overload conditions. Therefore, there is no algorithm that can feasibly schedule the task set. As we observe in Figure 7, there are better algorithms as compared with S_4. As represented in the figure the best performance belongs to algorithm S_8 whose priority function is in non-decreasing order of P_i/e_i. The optimal solution is smaller than the output of algorithm S_8.

One may suggest to derive an algorithm whose priority function is in non-decreasing order of $P_i/(e_i d_i)$. We refer to the new algorithm as S_{15}. In Figure 8, we observe that the results of S_{15} is not better than S_8.

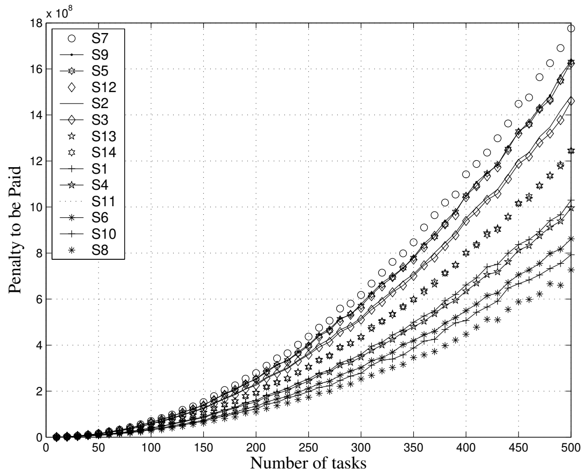

Figure 7: Comparing different priority functions

6 An Upper Bound

6.1 Deriving an Upper Bound

We observe, in Figure 7, that algorithm S_8 has the best solution as compared with the other algorithms discussed in Chapter 5. In other words, when the number of tasks grows, the penalty that should be paid by the schedule provided by the S_8 algorithm is

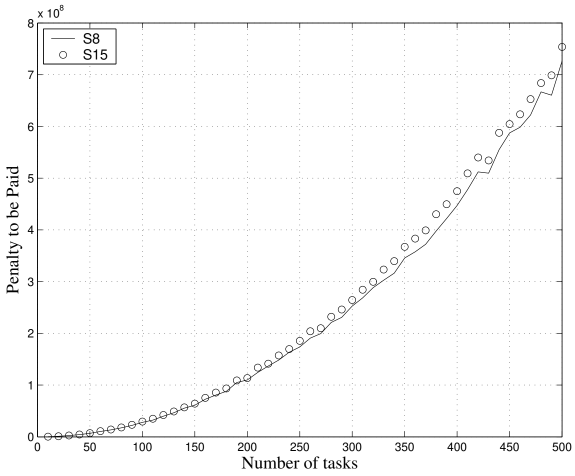

Figure 8: Comparing S_8 and S_{15}

smaller than the other algorithms. The priority assigned to each task in algorithm S_8 is non-decreasing in P_i/e_i. The optimal solution should perform equally or better than algorithm S_8. Therefore, we take the priority rule of algorithm S_8 to find the best upper bound on penalty amongst the heuristic algorithms in Chapter 5. As a matter of fact, corresponding to each heuristic algorithm in this paper, we can find an upper bound for the problem. Each of the upper bounds can be computed in $O(n^2)$ time, where n is the the number of tasks.

We find the upper bound as follows. As mentioned in Section 2, $C_i = \alpha_i e_i$ is maximum time taken to complete the task, after the task is started. Therefore,

$$\alpha_i e_i = e_i + \sum_{\substack{k=1,\\ P_i/e_i<P_k/e_k, r_k<r_i<F_k}}^{n} e_k + \sum_{\substack{l=1,\\ P_i/e_i<P_l/e_l, r_i<r_l<F_i}}^{n} e_l, \qquad (3)$$

where τ_k is any task which has arrived before τ_i, has a higher priority than τ_i, and has not been finished when τ_i arrives, and τ_l is any task which arrives after τ_i and has a higher priority than τ_i, and finishes before F_i. From Theorem 3, recall that a task cannot be preempted more than once by another task.

It can be verified that

$$\sum_{\substack{k=1,\\ P_i/e_i<P_k/e_k, r_k<r_i<F_k}}^{n} e_k + \sum_{\substack{k=1,\\ P_i/e_i<P_k/e_k, r_i<r_k<F_i}}^{n} e_k$$

$$\leq \sum_{\substack{j=1,\\ P_i/e_i<P_j/e_j}}^{n} e_j. \tag{4}$$

Therefore, from (3) and (4), we obtain the following inequality

$$\alpha_i e_i \leq e_i + \sum_{\substack{j=1,\\ P_i/e_i<P_j/e_j}}^{n} e_j.$$

Therefore, we conclude that

$$\sum_{i=1}^{n}(r_i+\alpha_i e_i-d_i)^+P_i$$
$$\leq \sum_{i=1}^{n}\left(r_i+e_i+\sum_{\substack{j=1,\\ P_i/e_i<P_j/e_j}}^{n} e_j-d_i\right)^+ P_i.$$

We hence obtain the following upper bound for the optimal penalty function

$$\min\sum_{i=1}^{n}(r_i+\alpha_i e_i-d_i)^+P_i$$
$$\leq \sum_{i=1}^{n}\left(r_i+e_i+\sum_{\substack{j=1,\\ P_i/e_i<P_j/e_j}}^{n} e_j-d_i\right)^+ P_i. \tag{5}$$

Note on the right hand side of (5) that all of the parameters in this upper bound are known before scheduling and it is not needed to run a scheduling algorithm to find them. Also, the upper bound can be calculated in $O(n^2)$ time, where n is the number of tasks, while finding $\min\sum_{i=1}^{n}(r_i+\alpha_i e_i-d_i)^+P_i$ is an NP-hard problem.

We need to find the optimal solution to compare it with the results of the upper bound and do not claim that the following algorithm is the best possible optimal algorithm for the problem. In order to find the optimal solution, we use the following steps: we find all of the $n!$ possible permutations of order of priorities, which are assigned to a set of n soft realtime tasks. Then, we call algorithm A for any individual permutation of priorities, which computes $\Sigma_{i=1}^{n}(F_i-d_i)^+P_i$ for each of them separately. Finally, we find the minimum of $\Sigma_{i=1}^{n}(F_i-d_i)^+P_i$ that corresponds to the optimal schedule. The running time of the optimal scheduling algorithm proposed in this section is $O(n!)$.

6.2 Simulation Results: The Upper Bound

We have implemented the optimal algorithm and computed the upper bound for $n = 1, 2, \cdots, 8$ for simulation purposes and comparison. Simulation conditions in this section are the same as those used in Section 5.2, except that the simulation is done for the algorithms for $n = 1$ to 8.

The optimal algorithm finds $\min \Sigma_{i=1}^{n}(F_i - d_i)^+ P_i$, where F_i is the finishing time of task τ_i. Figure 9 compares the results of the simulations by plotting the penalty to be paid versus the number of tasks.

As mentioned in Section 5.2, for any given task τ_i, d_i is randomly generated under the condition that $e_i + r_i \leq d_i$ holds. When there exists only one task to be scheduled on the processor, there is no possibility to miss the deadline of the task. Therefore, the upper bound and the optimal value of the penalty to be paid are zero. In other words, the upper bound coincides with the optimal solution at $n = 1$. When there exists only one task to be scheduled, the upper bound and the optimal value of the penalty are not shown in the logarithmic figure (Figure 9). In general, we may have a set of one or more tasks that can be feasibly scheduled by some algorithms on the processor without missing any deadlines. In this case, the penalty to be paid by the algorithm is equal to zero.

Also, we observe that the ratio of the upper bound to the optimal solution is less than 1.09. We have also computed and plotted, in Figure 10, the upper bound for the penalty to be paid versus the number of tasks for $n = 1, 2, \cdots, 500$. Note that while it is computationally infeasible to find the optimal penalty for a large number of tasks, our upper bound can be easily calculated in a polynomial time.

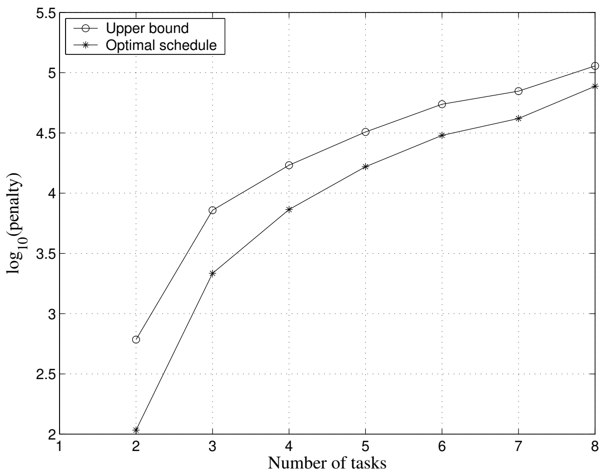

Figure 9: The total penalty of the optimal solution and the upper bound in (5).

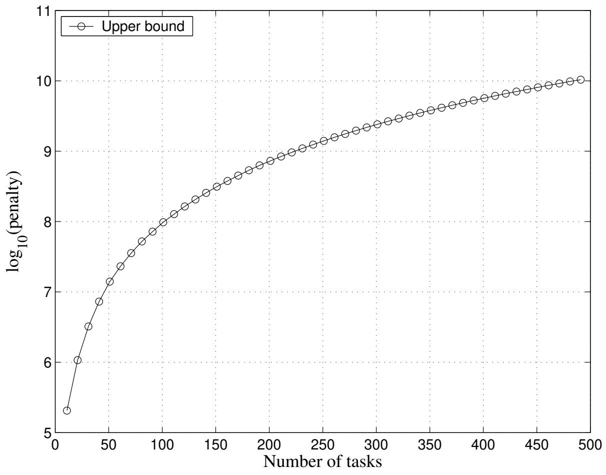

Figure 10: The upper bound of the total penalty for a large number of tasks.

We have compared the upper bound with the algorithm S_8. As we observe in Figure 11, the upper bound of the optimal penalty to be paid is smaller than the penalty paid by algorithm S_8. However, the upper bound just determines an upper bound for the penalty to be paid by the optimal solution and it does not determine a schedule.

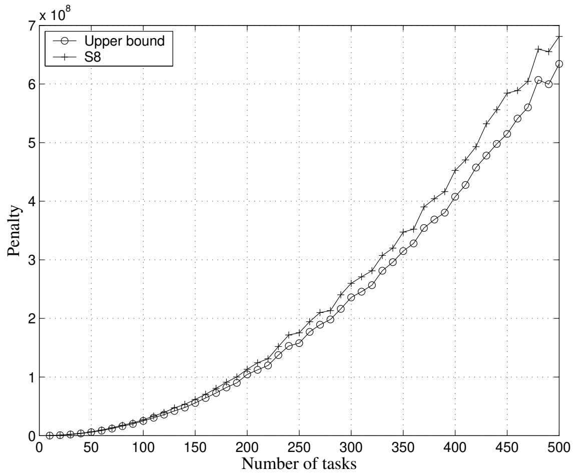

Figure 11: Comparing the upper bound of the total penalty and S_8.

7 Conclusions

In this paper, we study the problem of scheduling a a set of soft real-time tasks under overload conditions on a single processor, where our objective is to minimize the total penalty paid. Since the problem is NP-hard, it is not known whether an optimal schedule can be found in polynomial time. In spite of that, we are interested to know the behavior of any optimal algorithm for the problem. Hence, we prove a number of properties of any optimal scheduling algorithm for the problem. Then, we provide a number of heuristic algorithms which satisfy the properties obtained herein. The heuristic algorithms differ in the way that the tasks priorities are assigned. We compare the penalty to be paid by various algorithms derived in this paper. We observe that in the algorithms where the priority assigned to each task is non-decreasing in P_i, we pay a smaller total penalty as compared with the algorithms whose priority assignment is performed inversely. Similarly, we pay a smaller total penalty for the algorithms whose priority assignment is non-increasing in e_i or non-increasing in d_i, as compared with the algorithms whose priority assignment is performed inversely.

We conclude from the simulation results that algorithm S_8 has the best solution as compared with the other algorithms discussed in this paper. The priority assigned to each task in algorithm S_8 is non-decreasing in P_i/e_i. The optimal solution should be equal to or smaller than the output of algorithm S_8. We try another heuristic algorithm named S_{15}.

We also provide a tight upper bound for the objective function. The running time of computing the upper bound is $O(n^2)$, where n is the number of tasks. Therefore, it is feasible to compute the upper bound for a set of large real-time tasks in a short time. In order to determine the upper bound, we select the priority function of an algorithm which has the best solution as compared with the other algorithms.

An optimal scheduling algorithm is given in the paper to compare the penalty to be paid by the optimal solutions and its upper bound. The optimal algorithm grows exponentially with the number of tasks. Therefore, the comparison among the upper bound is done for small sets of real-time tasks.

The upper bound derived here is closer to the optimal as compared with even the best heuristic algorithm provided in this paper. However, the upper bound does not provide a schedule. Therefore, in practical problems, where we have to provide a schedule, algorithm S_8 is recommended.

Future work may include computing the ratio of our upper bound to the optimal solution. Moreover, one may study finding a lower bound. For a given set of real-time tasks, considering the fact that the problem is NP-hard, it is not known whether an optimal schedule can be found in polynomial time. However, it would be worthy if one can model the problem as a Linear Programming (LP) problem. In this case, some polynomial time algorithms, such as those that lie into the category of the Interior Point method [4], can be applied. These algorithms can find the value of the total penalties of the optimal solution which grows in polynomial time with the number of constrains of the LP problem.

References

[1] J. P. M. Arnaout and Gh. Rabadi, "Minimizing the toal weighted completion time on unrelated parallel machines with stochastic times," *Proceedings of Simulation Conference, 2005*, December 4-7, 2005.

[2] I. D. Baev, W. M. Meleis, and A. Eichenberger , "An experimental study of algorithms for weighted completion time scheduling," *Algorithmica*, Vol. 33, No. 1, pp. 34-51, 2002.

[3] I. D. Baev, W. M. Meleis, and A. Eichenberger, "Algorithms for total weighted completion time scheduling," *Proceedings of the tenth annual ACM-SIAM symposium on Discrete algorithms, Symposium on Discrete Algorithms*, pp. 852-853, January 17-19, 1999, Baltimore, Maryland, USA.

[4] D. Bertsimas and J. N. Tsitsiklis , *Introduction to linear optimization*, Athena Scientific, February 1997.

[5] G. C. Buttazzo, *Hard Real-Time Computing Systems: predictable scheduling algorithms and applications*, Springer, September 2006.

[6] G. Buttazzo, G. Lipari, L. Abeni, and M. Caccomo *"Soft Real-Time Systems, Predictability vs. Efficiency, "* Springer, 2005, NY, USA.

[7] C. Chekuri and R. Motwani, "Precedence constrained scheduling to minimize weighted completion time on a single machine," *Discrete Applied Mathematics*, Vol. 98, pp. 29-39, 1999.

[8] C. Chekuri, R. Motwani, B. Natarajan, and C. Stein, "Approximation techniques for average completion time scheduling," *SIAM Journal on Computing*, Vol. 31, No. 1, pp. 146-166, 2001.

[9] W. Fornaciari, P. di Milano, *Real-time operating systems scheduling lecturer*, www.elet elet.polimi polimi.it/ fornacia it/ fornacia.

[10] K. Frazer, *Real-time operating system scheduling algorithms*, 1997.

[11] M. R. Garey and D. S. Johnson, "Computers and intractability: a guide to the theory of NP-completeness," W. H. Freeman, January 15, 1979.

[12] M. X. Goemans, M. Queyranne, A. S. Schulz, M. Skutella, and Y. Wang, "Single machine scheduling with release dates," 2001.

[13] W. A. Halang and A. D. Stoyenko, *Real time computing*, NATO ASI Series, Series F: Computer and Systems Sciences, Volume 127, Springer-Verlag company, 1994.

[14] L. A. Hall, A. S. Schulz, D. B. Shmoys, and J. Wein, "Scheduling to minimize average completion time: off-line and on-line approximation algorithms," *Mathematics of Operations Research*, Vol. 22, pp. 513-544, August 1997.

[15] D. Isovic and G. Fohler, *Efficient scheduling of sporadic, aperiodic and periodic tasks with complex constraints*, in *Proceedings of the 21st IEEE RTSS*, Florida, USA, November, 2000.

[16] M. Joseph, *Real-time systems: specification, verification and analysis*, Prentice Hall, 1996.

[17] C. M. Krishna and K. G. Shin, *Real-time systems*, MIT Press and McGraw-Hill Company, 1997.

[18] P. A. Laplante, *Real-time systems design and analysis, an engineer handbook*, IEEE Computer Society, IEEE Press, 1993.

[19] A. Mohammadi and S. Akl, *Scheduling algorithms for real-time systems*, Technical Report 2005-499, School of Computing, Queen's University, 2005.

[20] M. W. P. Savelsbergh, R. N. Uma, and J. Wein, "An experimental study of LP-based approximation algorithms for scheduling problems," *INFORMS Journal of Computing*, Vol. 17, No. 1, pp. 123-136, Winter 2005.

[21] M. Skutella, "List scheduling in order of alpha-points on a single machine," in *Efficient Approximation and on-line algorithms*, editted by E. Bampis, K. Jansen and C. Kenyon, 2002.

[22] J. A. Stankovic and K. Ramamritham, *Tutorial on hard real-time systems*, IEEE Computer Society Press, 1988.