WORST CASE ANALYSIS FOR ON-LINE SCHEDULING IN REAL-TIME SYSTEMS

Fuxing Wang and Decao Mao
Department of Computer and Information Science
University of Massachusetts
Amherst, MA 01003

COINS Technical Report 91-54
June 1991

Worst Case Analysis for On-Line Scheduling in Real-Time Systems *

Fuxing Wang Decao Mao

Department of Computer and Information Science
University of Massachusetts
Amherst, MA 01003

June 1991

Abstract

On-line scheduling in real-time environments has been studied by a number of researchers [8, 16, 13, 4, 10, 1]. If the system is not overloaded, there exist several optimal uniprocessor on-line scheduling algorithms for real-time tasks, such as Earliest-Deadline-First and Least-Laxity-First. However, it has been proven that there are no optimal multiprocessor on-line scheduling algorithms for real-time tasks [8]. On the other hand, if overload is allowed, no optimal on-line scheduling algorithms exist, even for uniprocessors. Many researchers have turned to approximation algorithms [8, 16, 13, 4]. Therefore, it is important to study the behavior of approximation algorithms.

A good on-line scheduling algorithm should have both good average performance and good worst case performance. If we know the performance range of an on-line scheduling algorithm, it will greatly help in designing predictable real-time systems. In this paper, we study the performance bounds for both uniprocessor and multiprocessor on-line scheduling. Specifically, we consider tasks with different values to the system and consider the performance bound to be the ratio of the value obtained by an on-line scheduling algorithm and the value obtained by an ideal optimal off-line "clairvoyant" algorithm.

If all tasks have the same value density, i.e. the value per unit computation time, we show that the tight upper bound of the uniprocessor on-line scheduling problem is $1/4$. More generally, if tasks have different value densities and the ratio between the highest and the lowest value density is γ, we show that the upper bound for the uniprocessor on-line scheduling problem is $1/(\gamma + 1 + 2\sqrt{\gamma})$. Two on-line scheduling algorithms, TD_1 and TD_1', are presented, which can reach the two upper bounds, respectively.

*This work is part of the Spring Project at the University of Massachusetts and is funded in part by the Office of Naval Research under contract N00014-85-K-0398 and by the National Science Foundation under grant CDA-8922572.

1 Introduction

The problem of on-line scheduling in real-time environments is to *dynamically* make a sequence of decisions by assigning system resources to real-time tasks. This decision must be made without *a priori* knowledge of future tasks. System resources are processors, memory, and shared data structures[1], and the tasks are independent and preemptable, and have arbitrary arrival times, computation times, deadlines, and importance values. Because a scheduling decision is made without *a priori* knowledge, the outcome of the decision is not fully predictable. So, the objective is to maximize the value accrued from tasks that complete on time.

With *a priori* knowledge of future tasks, scheduling is actually not on-line in nature, although the decisions are made on-line. For example, if overloads are impossible, then the Earliest-Deadline-First algorithm (EDF) can be applied, since it has beed proven that every task can finish before its deadline [11]. Intuitively, when a task is preempted, since we have a future knowledge that overloads will never occur, we have 100% confidence that the remaining portion of the task can be completed before its deadline. Many real-time systems do not have future knowledge. One example is robotics, which requires its control subsystem to adapt to a dynamic environment. It will be too costly to assume that overload will never occur and/or inefficient to construct a schedule *a priori* in such a system. Therefore, on-line scheduling is important in such real-time systems, and on-line scheduling is more practical than off-line scheduling because the overload *will* occur in many systems. Overload happens in many practical systems because

- the environment changes;
- there is a burst of task arrivals; or
- a part of the system fails.

Hence, on-line scheduling is necessary to shed task load. Without overload, simple algorithms, such as EDF and Least-Laxity-First (LLF), perform very well. However, with overload, it is more difficult to construct a good on-line algorithm to compete with a *clairvoyant* algorithm, as it will be clear from the following. A clairvoyant algorithm is an ideal optimal off-line algorithm with full knowledge of task parameters.

The *lower bound* on the performance of an on-line scheduling algorithm, A, can be defined in the following way: If over all task arrival sequences, the smallest value of the ratio of the

[1]Only processors are considered in this paper.

performance of A and that of the clairvoyant algorithm is B_A, then B_A is the *lower bound* on the performance of A. If, for a given scheduling problem, the largest value of B_A for all A is B, then B is the upper (performance) bound of the scheduling problem.

Dertouzos and Mok studied multiprocessor on-line scheduling of hard real-time tasks [8][2]. They showed that, in the case of uniprocessors, both EDF and LLF are optimal in the sense that, for any task request pattern, if there exists one feasible schedule, both EDF and LLF are guaranteed to find it. But both algorithms have difficulty if overloads occur. In the case of multiprocessors, they proved that no scheduling algorithm can be optimal without *a priori* knowledge of task deadlines, computation times, and arrival times.

Locke developed an efficient approximation algorithm called Best-Effort (BE) for multiprocessor on-line scheduling, by using time-dependent value functions to schedule real-time tasks [13]. For uniprocessor scheduling, BE behaves the same as EDF if the system is not overloaded. During overload, BE sheds tasks with the Lowest Value Density First[3], until the system becomes underloaded. Although BE has been shown to have a good average performance, it does not perform well in the worst case (see Example 1 of Section 2).

Biyabani, Stankovic, and Ramamritham proposed two on-line algorithms for uniprocessor scheduling in a real-time distributed environment and showed that the two algorithms perform well by simulation studies [4]. They assumed that tasks have both timing constraints and importance values. If the system is not overload, these two algorithms behave the same as EDF. If the system is overloaded, their algorithms shed tasks with less importance value. These two algorithms differ only in how they remove lower importance tasks. In the first algorithm, lower importance tasks are removed one at a time and in strict order from low to high importance. The second algorithm also removes tasks with the lower importance value, but does not follow the strict order found in the first algorithm. Again, the two algorithms do not perform well in the worst case (see Example 2 of Section 2).

Recently, Koren, Mishra, Raghunathan, and Shasha have been studying the uniprocessor on-line scheduling problem [10][4]. They proposed an algorithm, D^*, and showed that the D^* algorithm has the lower bound of 1/5, under the assumption that all tasks have the same value density, which they called *uniform value-density*.

[2]Their results first appeared in 1978 [15].

[3]Value density is defined as the ratio between task's value and its computation time, therefore, it measures the value per unit computation time.

[4]It appears that they have an revised version which we have not seen yet.

More recently, Baruah and Rosier showed that the uniprocessor on-line scheduling problem has a performance upper bound of 0.414 [1]. Given the results described in the following sections, the 0.414 performance bound does not appear to be correct.

Besides the real-time on-line scheduling problem, there are many other on-line problems. The recent theoretical development of on-line algorithms has established a *Theory of On-Line Algorithms* which compares relative power of on-line and off-line (or clairvoyant) algorithms. Sleator and Tarjan analyzed the list researching and paging problems [17]. Borodin, Linial, and Saks studied the metrical task system problem [5]. Manasse, McGeock, and Sleator presented results on the K-sever problem. Both the metrical task system and the K-sever are abstract models. Other work on the theory of on-line algorithms can be found in [9, 3, 7, 2].

From the above discussion, it is clear that on-line algorithms is an important research issue. On-line algorithms can be applied to many important applications, such as dynamic control and operations research.

In this paper, we study on-line scheduling in a real-time environment. We consider both uniprocessor on-line scheduling and multiprocessor on-line scheduling. Tasks have either the same value density or different value densities, where value density is defined as the value in per unit time, e.g., the ratio between the value of a task and its computation time.

In the case of uniprocessor on-line scheduling, if tasks have the same value density, we show that, the performance upper bound is 1/4. Another interpretation of the result is that, in the worst case, an on-line algorithm is only able to complete the amount of work which is 1/4 of an amount work completed by a clairvoyant algorithm. We also show that 1/4 is a tight bound by constructing an on-line algorithm, called Threshold-1 (TD_1) algorithm, to reach the bound. TD_1 guarantees no less than 1/4 of the value obtained by a clairvoyant algorithm for any different kind of task request sequences. This means that TD_1 has the best *lower bound* among all on-line algorithms. Therefore, TD_1 is an *optimal on-line* scheduling algorithm under overload, and furthermore, it has the same performance as LLF and EDF in case of non-overload.

The above result can be further extended to the cases in which tasks have different value densities. Let γ be the ratio of the highest and lowest value densities of tasks, we prove that the upper bound of the on-line scheduling problem is

$$\frac{1}{\gamma + 1 + 2\sqrt{\gamma}}.$$

As a special case, if γ is 1, the upper bound is 1/4, which is just the result mentioned above. If

γ is 2, the upper bound is 1/5.828.

In the case of multiprocessor on-line scheduling, some important observations are provided, which will help to derive the tight bounds for more generalized on-line scheduling problem. The main strategy used in multiprocessor on-line scheduling is the careful coordination among processors.

The remainder of the paper is organized as follows. Section 2 presents some notations, assumptions, and examples. Section 3 presents a brief summary on several analytical results for the on-line scheduling problem. Section 4 presents some useful properties of a family of integer sequences. These are useful in deriving the upper bound of on-line scheduling problem. In Section 5 we study uniprocessor on-line scheduling by assuming that all tasks have the same value-density. This assumption is removed in Section 6. In Section 7, we discuss some important hints about how to derive the tight bounds for multiprocessor on-line scheduling. We conclude the paper in Section 8.

2 Assumptions and Example

A system consists of a set $\mathcal{P}$ of m application processors: $\mathcal{P} = \{P_1, P_2, ..., P_m\}$, which are identical. The system serves a sequence of tasks. Let R be an arbitrarily task request sequence, $\{T_1, T_2, ..., T_n\}$, where n can be arbitrarily large. Task T_i is defined by (a_i, c_i, d_i, v_i), where

- a_i — its arrival time,
- c_i — its computation time,
- d_i — its deadline,
- v_i — the value obtained by the system if the task completes its execution before its deadline.

We assume tasks are aperiodic, independent, and preemptable without penalty. A preempted task can be resumed on any available processor. We also assume that $a_i \leq a_{i+1}$, where $1 \leq i < n$. Further, the system obtains a zero value from a task if it misses its deadline.

Definition 1: Let R be an arbitrary task request sequence. A is an *on-line* scheduling algorithm if it knows T_i only at time a_i. A *clairvoyant* algorithm, C, is an ideal optimal off-line scheduling algorithm, which knows all tasks in R *a priori*.

Definition 2: Let R be an arbitrary task request sequence, A be an on-line scheduling algorithm, and C be a clairvoyant algorithm. $V_A(R)$ is the total value obtained by A. $V_C(R)$ is the total value obtained by C.

Definition 3: The *lower bound*, B_A, of an on-line scheduling *algorithm*, A, is defined as

$$\frac{V_A(R)}{V_C(R)} \geq B_A, \qquad \text{for all } R,$$

where $B_A \in [0,1]$ because $\forall R \{ V_A(R) \leq V_C(R)\}$.
The *tight* lower bound, TB_A, of an on-line scheduling *algorithm*, A, is

$$TB_A = \sup\{B_A\}.$$

The *upper bound*, B, of an on-line scheduling *problem* is defined as

$$B \geq TB_A, \qquad \text{for all } A.$$

The *tight* upper bound, TB, of an on-line scheduling *problem* is

$$TB = \sup\{TB_A : \text{ for all on-line algorithms } A\}.$$

For example, if A is an on-line scheduling algorithm with a tight lower bound of 0.2. then, 0.1 is also a lower bound for A, but it is not tight. On the other hand, if we assume all on-line scheduling algorithms have tight lower bounds in a range of [0, 0.25], then, 0.3 is an upper bound for the on-line scheduling problem and 0.25 is the tight upper bound. In the remainder of the paper, both the lower bound and the upper bound simply mean tight bounds. Now a couple of examples will illustrate these terms and ideas.

Example 1: Let A be an on-line scheduling algorithm in a uniprocessor system. A uses a simple strategy to make scheduling decisions: it uses EDF when the system is underloaded, and it favors a task with *larger value density* during overload. Let

$$R = \{T_1, T_2\},$$

with their parameters specified in the following table:

Tasks	a_i	c_i	d_i	v_i
T_1	0	2	2	3
T_2	1	100	101	100

At time 0, T_1 arrives and gets service. At time 1, T_2 arrives and the system is overloaded. Algorithm A favors an task with a larger value density, which is T_1. Hence, T_2 is rejected and is lost. The total value obtained by A is 3, and the total value obtained by a clairvoyant algorithm can be 100 (v_2). The performance ratio is

$$\frac{V_A(R)}{V_C(R)} = \frac{3}{100}.$$

If both computation time and value of T_2 increase at the same rate, then the ratio between $V_A(R)$ and $V_C(R)$ goes to zero.

Example 2: Let A be an on-line scheduling algorithm in a uniprocessor system. A uses a simple strategy to make scheduling decisions: it uses EDF when the system is underloaded, and it favors a task with *larger value* during overload. Let

$$R = \{T_1, T_1', T_2, T_2', T_3, T_3', T_4, T_4', T_5, T_5', T_6, T_6', T_7, T_7', T_8, \},$$

with their parameters specified in the following table:

Tasks	a_i	c_i	d_i	v_i	Tasks	a_i	c_i	d_i	v_i
T_1	0	10	10	10	T_1'	0	9	11	9
T_2	9	11	20	11	T_2'	9	10	21	10
T_3	19	12	31	12	T_3'	19	11	32	11
T_4	30	13	43	13	T_4'	30	12	44	12
T_5	42	14	56	14	T_5'	42	13	57	13
T_6	55	15	70	15	T_6'	55	14	71	14
T_7	69	16	85	16	T_7'	69	15	86	15
T_8	84	16	100	16					

Notice that the value densities of all tasks are the same, which is 1. The schedule of a clairvoyant algorithm is simply in the following order:

$$(T_1', T_2', T_3', T_4', T_5', T_6', T_7', T_8),$$

with the total value 100. The algorithm A works as follows: At time 0, the system is empty and T_1 and T_1' arrive. T_1 gets service and T_1' is discarded because A favors the larger valued task during overload. (A does not know that T_2 will arrive, otherwise it will choose T_1'.) At time 9, T_2 and T_2' arrive and the system is overloaded again, and T_2 gets service because it is the task with largest value among the current task sets. This pattern continues until

T_8 arrives at time 84. The current running task is T_7 with the same value as T_8, hence, algorithm A does not make the switch. The total value obtained by A is 16 because only T_8 makes its deadline and all other tasks are lost. The performance ratio is

$$\frac{V_A(R)}{V_C(R)} = \frac{16}{100}.$$

The above task pattern can be used to construct a scenario with a task arrival sequence with an arbitrarily number of tasks, such that, the ratio between $V_A(R)$ and $V_C(R)$ goes to zero.

From the above examples, we can observe a phenomenon which is common in on-line scheduling. That is, an on-line algorithm times makes some mistakes because it lacks *a priori* knowledge. This is unavoidable. However, an on-line algorithm may still provide a certain level of predictability on its performance, which is measured by the lower bound defined above. Further, researchers are searching for on-line scheduling algorithms with good lower bounds. The best one can reach the upper bound of the problem. One benefit from this kind of research is that, after we know the upper bound of the problem, we have a deeper insight into the behavior of on-line scheduling, and we may be able to avoid the worst cases during the design of real-time systems.

3 Overview of Results

There are three main results. The first two results concern uniprocessor on-line scheduling and the third result is about multiprocessor on-line scheduling.

The first result is the upper bound of the uniprocessor on-line scheduling problem for tasks with the same value density.

Theorem 1: If all tasks have the same value density, then the upper bound of uniprocessor on-line scheduling problem is 1/4.

Note:

- The assumption of the same value density on all tasks may not be practical. Nevertheless, it is the first step in studying the on-line scheduling problem. It provides a basis for analyzing more sophisticated models.

- If we interpret the task's computation time as its value, then this theorem says that, in the worst case, any on-line algorithm can only complete 1/4 of the work completed by a clairvoyant algorithm.

- The theorem has another implicit assumption, which is that the computation time of tasks can be arbitrarily small or large. This may not be true in practice. We are currently studying a case in which the ratio between the largest and the smallest computation time is bounded. When the ratio is arbitrarily close to 1, the upper bound is also close to 1. The upper bound decreases while the ratio increases. The upper bound converges to 1/4, as the ratio goes to infinity.

The next result is again about the upper bound of the uniprocessor on-line scheduling problem, except that the restriction on tasks' value density being the same is removed. Therefore, tasks are allowed to have arbitrary value densities.

Theorem 2: If γ is the ratio between the highest and the lowest value density of tasks, then the upper bound of the uniprocessor on-line scheduling problem is

$$\frac{1}{\gamma + 1 + 2\sqrt{\gamma}}.$$

Note:

- It is easy to verify that the upper bound is 1/4 when γ is 1, which is the same as the first result.

- When γ increases, the upper bound decreases. This means that a clairvoyant algorithm has more advantage over an on-line scheduling algorithm, because, in the worst case, the clairvoyant algorithm works on the highest value density tasks while the on-line algorithm works on the lowest value density tasks.

4 Constant-Ratio Sequences

In this section, we study a particular family of integer sequences, which will give us some insight into the worst case behavior of on-line scheduling. In particular, the computations time of a task sequence will correspond to such an integer sequence. The properties of this family are

used in the proof of the upper bound of on-line the scheduling problem in the next section. One example of the sequence in this family is

$$1, 3, 8, 20, 48, 112, \cdots \tag{1}$$

which is defined by a *recurrence relation*, or *difference equation*:

$$c_{k+2} = 4(c_{k+1} - c_k)$$

with $c_0 = 1$ and $c_1 = 3$.

In general, this family of integer sequences has a generic form:

$$c_{k+2} = \beta(c_{k+1} - c_k) \tag{2}$$

with

$$c_0 = 1, \quad \text{and} \quad c_1 = \beta - 1.$$

When $\beta = 4$, it gives sequence (1). This family has some interesting properties, which, to our knowledge, have not been studied in literature, and it will be called the *Constant-Ratio* (CR) sequences because of the next property:

Property 1: [Constant-Ratio Property]

$$\frac{c_{k-1}}{\sum_{j=0}^{k} c_j} = \frac{1}{\beta}. \tag{3}$$

Proof.

$$\begin{aligned}
\sum_{j=0}^{k} c_j &= c_0 + c_1 + \sum_{j=2}^{k} \beta(c_{j-1} - c_{j-2}) \\
&= 1 + (\beta - 1) + \beta \sum_{j=2}^{k} c_{j-1} - \beta \sum_{j=2}^{k} c_{j-2} \\
&= \beta + (\beta c_{k-1} + \beta \sum_{j=1}^{k-2} c_j) - (\beta + \beta \sum_{j=1}^{k-2} c_j) \\
&= \beta c_{k-1}.
\end{aligned}$$

Hence,

$$\frac{c_{k-1}}{\sum_{j=0}^{k} c_j} = \frac{1}{\beta}.$$

□

We will use a CR sequence to construct a task request pattern, such that, in the worst case, a clairvoyant algorithm is able to obtain a value which is close to $\sum_{j=0}^{k} c_k$ while an on-line scheduling algorithm can only obtain a value c_{k-1}. This is the main reason for studying CR sequences.

Next, we show that a CR sequence is monotonically increasing if $\beta \geq 4$, by using a rather standard method in the study of recurrence relations [12].

Property 2: [Monotonicity Property]

If $\beta > 4$,

$$c_k = (\frac{1}{2} + \frac{\beta - 2}{2\sqrt{\beta(\beta-4)}})(\frac{\beta + \sqrt{\beta(\beta-4)}}{2})^k + (\frac{1}{2} - \frac{\beta - 2}{2\sqrt{\beta(\beta-4)}})(\frac{\beta - \sqrt{\beta(\beta-4)}}{2})^k \quad (4)$$

If $\beta = 4$,

$$c_k = (\frac{k}{2} + 1)2^k. \quad (5)$$

Proof. Given the recurrence relation (2):

$$c_{k+2} = \beta(c_{k+1} - c_k)$$

the corresponding characteristic equation is

$$x^2 - \beta x + \beta = 0.$$

Part 1: When $\beta > 4$, the characteristic equation has two distinct roots

$$x_1 = \frac{\beta + \sqrt{\beta(\beta-4)}}{2} \qquad \text{and} \qquad x_2 = \frac{\beta - \sqrt{\beta(\beta-4)}}{2}.$$

It follows that

$$c_k = A_0(\frac{\beta + \sqrt{\beta(\beta-4)}}{2})^k + A_1(\frac{\beta - \sqrt{\beta(\beta-4)}}{2})^k. \quad (6)$$

With the boundary conditions

$$c_0 = 1 \qquad \text{and} \qquad c_1 = \beta - 1,$$

the two constants can be determined as:

$$A_0 = \frac{1}{2} + \frac{\beta - 2}{2\sqrt{\beta(\beta - 4)}} \qquad \text{and} \qquad A_1 = \frac{1}{2} - \frac{\beta - 2}{2\sqrt{\beta(\beta - 4)}}.$$

Substitute A_0 and A_1 into (6) deriving (4).

Part 2: When $\beta = 4$, the characteristic equation has two identical roots:

$$x_1 = x_2 = 2.$$

It follows that

$$c_k = (A_0 k + A_1)2^k. \tag{7}$$

With the boundary conditions

$$c_0 = 1 \qquad \text{and} \qquad c_1 = \beta - 1,$$

the two constants can be determined as:

$$A_0 = \frac{1}{2} \qquad \text{and} \qquad A_1 = 1.$$

Substitute A_0 and A_1 into (7) deriving (5). Furthermore, Equation (5) is clearly monotonically increasing while k increases. If $\beta > 4$, the sequence defined by (2) will increase faster than the case in which $\beta = 4$. This can be seen in Figure 1. Therefore, the sequence is monotonically increasing while k increases, when $\beta > 4$. This completes the proof. □

Finally, we show that a CR sequence has an oscillation property if $\beta < 4$.

Property 3: [Oscillation Property]

If $\beta < 4$,

$$c_k = \frac{2}{\sqrt{4 - \beta}}(\sqrt{\beta})^{k+1} \cos(k\theta - \theta_1), \tag{8}$$

where

$$\theta = \tan^{-1}\sqrt{\frac{4 - \beta}{\beta}} \qquad \text{and} \qquad \theta_1 = \tan^{-1}\frac{\beta - 2}{\sqrt{\beta(4 - \beta)}}.$$

Proof. The corresponding characteristic equation is

$$x^2 - \beta x + \beta = 0,$$

which has two distinct roots

$$x_1 = \frac{\beta + i\sqrt{\beta(4-\beta)}}{2} \qquad \text{and} \qquad x_2 = \frac{\beta - i\sqrt{\beta(4-\beta)}}{2},$$

where $i = \sqrt{-1}$.

It follows that

$$c_k = A_0(\frac{\beta + i\sqrt{\beta(4-\beta)}}{2})^k + A_1(\frac{\beta - i\sqrt{\beta(4-\beta)}}{2})^k. \tag{9}$$

With the boundary conditions

$$c_0 = 1 \qquad \text{and} \qquad c_1 = \beta - 1,$$

we have

$$A_0 + A_1 = 1$$

and

$$A_0(\frac{\beta + i\sqrt{\beta(4-\beta)}}{2}) + A_1(\frac{\beta - i\sqrt{\beta(4-\beta)}}{2}) = \beta - 1.$$

The two constants can be determined as:

$$A_0 = \frac{1}{2} - \frac{i(\beta-2)}{2\sqrt{\beta(4-\beta)}} \qquad \text{and} \qquad A_1 = \frac{1}{2} + \frac{i(\beta-2)}{2\sqrt{\beta(4-\beta)}}.$$

But

$$(\frac{\beta + i\sqrt{\beta(4-\beta)}}{2})^k = (\sqrt{\beta}^k)(\cos k\theta + i \sin k\theta) \tag{10}$$

and

$$(\frac{\beta - i\sqrt{\beta(4-\beta)}}{2})^k = (\sqrt{\beta})^k(\cos k\theta - i \sin k\theta), \tag{11}$$

where

$$\theta = \tan^{-1}\sqrt{\frac{4-\beta}{\beta}}.$$

Substitute A_0, A_1, (10), and (11) into (9):

$$c_k = (\frac{1}{2} - \frac{i(\beta-2)}{2\sqrt{\beta(4-\beta)}})(\sqrt{\beta})^k(\cos k\theta + i \sin k\theta) + (\frac{1}{2} + \frac{i(\beta-2)}{2\sqrt{\beta(4-\beta)}})(\sqrt{\beta})^k(\cos k\theta - i \sin k\theta),$$

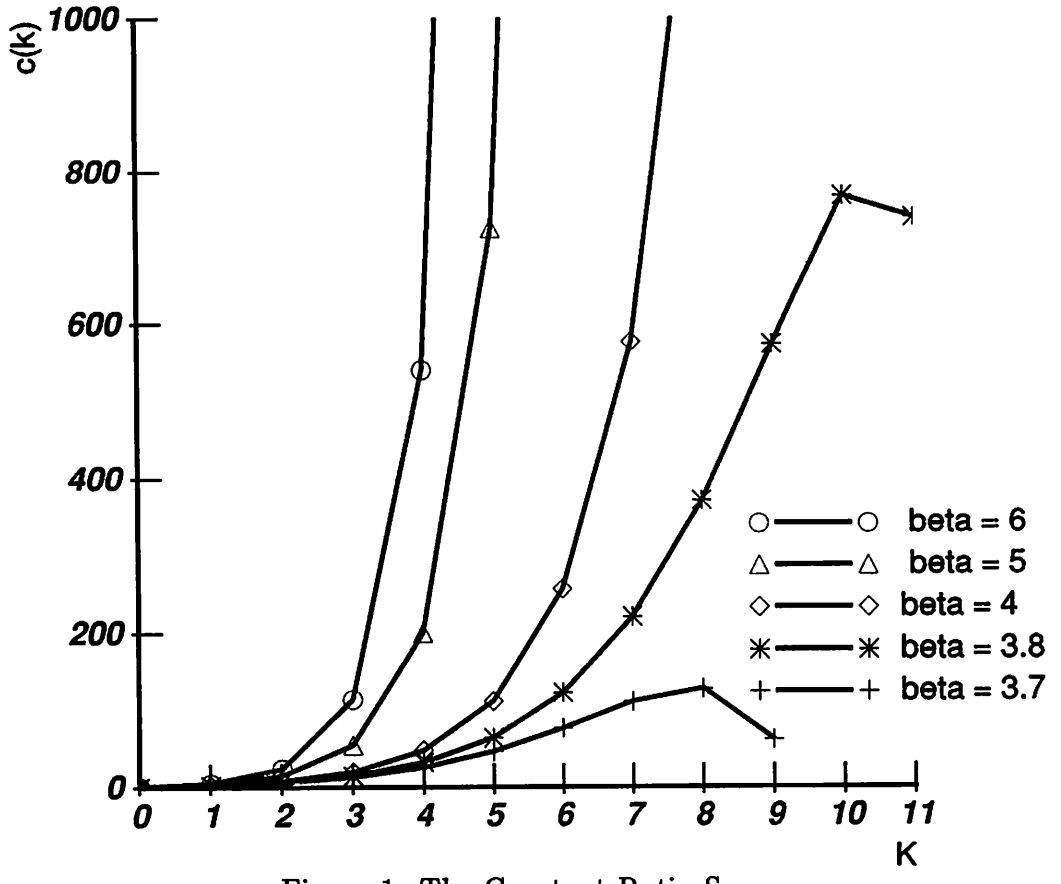

Figure 1: The Constant-Ratio Sequences.

or simply

$$c_k = (\sqrt{\beta})^k(\cos k\theta + \frac{\beta - 2}{\sqrt{\beta(4-\beta)}} \sin k\theta).$$

By defining

$$\theta_1 = \tan^{-1}\frac{\beta - 2}{\sqrt{\beta(4-\beta)}},$$

we have

$$c_k = \frac{2}{\sqrt{4-\beta}}(\sqrt{\beta})^{k+1} \cos(k\theta - \theta_1).$$

c_k oscillates while k increases, because there exists a factor of the *cos* function of k and θ while $0 < \theta < 2\pi$. □

Observation 1: As mentioned before, CR sequences are used to construct a task request pattern, such that, in the worst case, a clairvoyant algorithm is able to obtain a value which is close to $\sum_{j=0}^{k} c_k$ while an on-line scheduling algorithm can only obtain a value c_{k-1}. The ratio between c_{k-1} and $\sum_{j=0}^{k} c_k$ is $1/\beta$ according to Property 1. Hence, the ratio decreases

while β increases. But if $\beta \geq 4$, all sequences monotonically increase, which means that the value of tasks becomes larger and larger. An on-line scheduling algorithm will simply make a switch every time a more valuable task arrives. The ratio is likely to be much better than $1/\beta$ from the on-line scheduling algorithm point of view, and the sequence keeps on to infinity. The ratio will be much better than $1/\beta$ because the on-line scheduling algorithm gets the most valuable task. On the other hand, when β is less than 4, the sequence begins oscillating. Whenever $c_k \geq c_{k+1}$, the on-line scheduling algorithm can not make the switch because it is not worth it to switch to a less valuable task. Then, Property 1 can be used to measure the performance ratio. Intuitively, the worst case happens when β is very close to 4. Figure 1 shows the behavior of CR sequences with the different values of β (**beta** in Figure 1).

5 Uniprocessor On-Line Scheduling for Tasks with the Same Value Density

In this section, we assume that the value density of all tasks is a constant, and we will simply use the computation time of a task as its value. Under this assumption, we prove that the upper bound of the uniprocessor on-line scheduling problem is 1/4, that is, no on-line algorithm has its lower bound better than 1/4. Then we present a simple threshold algorithm with a guaranteed performance ratio of at least 1/4 compared to a clairvoyant algorithm. This is the best among all on-line scheduling algorithms with respect to the lower bound.

We first consider a general framework for studying on-line scheduling algorithms. The on-line scheduling problem can be considered as a "game" played by a *player* and an *adversary*. Whenever the adversary posts certain tasks, with different values and deadlines, the player examines these tasks and makes an on-line decision by applying an on-line policy or algorithm, A, to pick some tasks and to reject others, such that the total value obtained is as high as possible.

To show the upper bound of the uniprocessor on-line scheduling problem, it is sometimes necessary to consider the behavior of all algorithms on all possible input patterns according to Definition 3. This is a very difficult job because it is not practical to scrutinize all on-line scheduling algorithms. To avoid this, we use the following approach in our proof. We first show that there exists a task request sequence pattern from the adversary, such that, no on-line

scheduling algorithm can get a performance ratio higher than 1/4. Then we show that an on-line scheduling algorithm, TD_1, has its performance ratio at least 1/4 for all task request sequences. Consequently, the upper bound of the uniprocessor on-line scheduling problem is 1/4 by simply combining these two facts.

Lemma 1: There exists a task request sequence pattern, P, such that, no on-line scheduling algorithm can get a performance ratio higher than 1/4 compared to a clairvoyant algorithm.

Proof It is enough to prove that there exists P and an arbitrarily small δ, such that,

$$(\forall R \in P \quad \forall A \quad \frac{V_A(R)}{V_C(R)}) \leq \frac{1}{4} + \delta.$$

Let A be an arbitrary on-line scheduling algorithm used by the player of the "game". The adversary uses two types of tasks: τ-tasks and α-tasks with identical value density, represented by

(arrival-time, computation-time, deadline)

as follows:

$$\tau\text{-tasks} \quad : \quad T_t^\tau = (t, \tau, t+\tau),$$

and

$$\alpha\text{-tasks} \quad : \quad T_t^\alpha = (t, \alpha, t+\alpha),$$

where t is time, and α and τ are real to represent task size (computation time), while τ can be arbitrarily small and α will be specified in the following. These two task types have zero laxity, so the player is forced to make a decision immediately whenever a task arrives.

At time 0, the adversary posts a τ-task and an α-task:

$$T_0^\tau = (0, \tau, \tau), \quad \text{and} \quad T_0^1 = (0, 1, 1).$$

The player has only two choices, T_0^τ or T_0^1. If the player chooses T_0^τ, T_0^1 will be lost. The adversary will stop the game by not providing more requests. In contrast, the clairvoyant algorithm simply chooses T_0^1, and the ratio between the value obtained by the on-line scheduling algorithm, which is τ, and the value obtained by the clairvoyant algorithm, which is 1, is equal to $\tau/1$ which is far less than 1/4.

If the player chooses T_0^1, then the adversary posts another τ-task at time τ:

$$T_\tau^\tau = (\tau, \tau, 2\tau).$$

Again the player has only two choices: switch or not. If the player aborts T_0^1 for T_τ^τ, the adversary will stop the game and the ratio will be far less than 1/4. If the player keeps T_0^1, the game continues.

In general, while the player serves an α-task, τ-tasks will keep coming one after another. The adversary will stop the game whenever the player aborts the current α-task for a τ-task.

Now we specify the arrival pattern of other α-tasks. The second α-task will be $T_{1-\tau}^{3-\epsilon}$ at time $1-\tau$. At that time, if the player does not abort the current α-task, T_0^1, the game stops. There are no τ-tasks arriving after T_0^1 completes. Therefore, the player obtains the total value 1, while the clairvoyant algorithm gets the values from $T_{1-\tau}^{3-\epsilon}$ and all τ-tasks between time 0 and $1-\tau$. The ratio is

$$\frac{1}{4-\tau-\epsilon} \leq \frac{1}{4} + \delta,$$

where δ is a function of τ and ϵ, and δ goes to zero as both τ and ϵ go to zero. On the other hand, if the player aborts T_0^1 for $T_{1-\tau}^{3-\epsilon}$, the game continues. τ-tasks keep coming one after another during the time the player serves $T_{1-\tau}^{3-\epsilon}$.

In general, if the player does not abort the current α-task for a τ-task or if the player aborts the current α-task for a new α-task, the adversary will keep posting more τ-tasks and α-tasks. Each τ-task follows the previous τ-task, and each α-task is posted at the time just when the previous α-task can be completed.

The computation time of α-tasks are defined by the following recurrence relation:

$$c_0 = 1,$$

$$c_1 = \beta - 1,$$

$$c_{k+2} = \beta(c_{k+1} - c_k),$$

where

$$\beta = 4 - \epsilon$$

and c_0 and c_1 correspond to T_0^1 and $T_{1-\tau}^{3-\epsilon}$ respectively.

According to Property 1 and Property 3, we have

$$\frac{c_k}{\sum_{j=0}^{k+1} c_j} = \frac{1}{4-\epsilon}. \tag{12}$$

and

$$c_k = \frac{2}{\sqrt{4-\beta}}(\sqrt{\beta})^{k+1}\cos(k\theta - \theta_1),$$

where $\beta = 4 - \epsilon$,

$$\theta = \tan^{-1}\sqrt{\frac{4-\beta}{\beta}},$$

and

$$\theta_1 = \tan^{-1}\frac{\beta - 2}{\sqrt{\beta(4-\beta)}}.$$

Because c_k has a factor of *cos* function of k and θ, and $\theta \neq 0$, $\theta \neq 2\pi$, therefore, there exists k', such that

$$c_{k'} > c_{k'+1}.$$

Hence, the size of α-tasks does not monotonically increase. Whenever the size of the next α-task $c_{k'+1}$ is less than $c_{k'}$, the adversary changes $c_{k'+1}$ to be the same size as $c_{k'}$, and stops posting more τ-tasks and α-tasks. At this moment, the player can only choose one of the last two α-tasks with the same size, and obtains a total value c'_k. The clairvoyant algorithm gets the total value of

$$(\sum_{j=0}^{k'} c_j) + c'_k - k'\tau.$$

By Equation (12), we have

$$\frac{c'_k}{(\sum_{j=0}^{k'} c_j) + c'_k - k'\tau} < \frac{c'_k}{(4-\epsilon)c'_k - k'\tau} \leq \frac{1}{4} + \delta,$$

where δ is a function of τ and ϵ, and δ goes to zero as both τ and ϵ go to zero.

In summary, the adversary uses a task request pattern, P, such that any on-line scheduling algorithm used by the player has a performance ratio no more than 1/4. □

Next, we show the 1/4 bound is reachable, that is, there exists a particular on-line scheduling

algorithm, TD_1, which has a performance ratio at least 1/4 for all task request sequences, including the worst case pattern we used in the proof of Lemma 1.

To introduce the TD_1 algorithm, we define some notation. Then we consider several examples in order to understand the properties of TD_1. Finally, we present its pseudo code in three versions.

In the first version, we assume that all tasks have zero laxity. In the second version, we assume that tasks may have laxities but all preempted tasks are discarded. In the last version, the above two restrictions are removed.

Let t denote current time. Let T_i be an arbitrary task, $T_i = (a_i, c_i, d_i, v_i)$. a_i, c_i, d_i, and v_i are the arrival time, computation time, deadline, and value of T_i respectively as defined in Section 2. Let l_i be the *latest start time* of T_i:

$$l_i = d_i - c_i.$$

Definition 4: A *time interval*, or simply *interval*, at t is a time segment $[t_b, t_e)$ which consists of a busy subsegment $[t_b, t_f]$ followed by an optional idle subsegment $[t_f, t_e)$, where t_b ($t_b \leq t$) is the time the system transits from an idle state to a running state, t_f ($t \leq t_f$) is the time the system transits (or is expected to transit) back from a running state to an idle state because a task completes (or is expected to complete), and

$$t_e = \max(t_f, \max(\{dl_{discarded}\}))$$

where $\{dl_{discarded}\}$ are the deadlines of all tasks discarded during the time subsegment $[t_b, t_f]$.

Definition 5: An interval is *closed* at t whenever a task has completed in the interval, otherwise, it is *open*.

Intervals may be separate, cascade, or partially overlap each other. We present two examples to illustrate these terms and ideas.

Example 3: Figure 2 shows two intervals, one is closed and one remains open. The shadowed areas represent tasks which are either complete or running, and un-shadowed areas represent tasks that missed their deadlines. There are three tasks with zero laxity. T_0 arrives at time a_0 and is served at once, which opens Interval 1. The interval is closed when T_0 finishes at its deadline d_0. So Interval 1 is $[a_0, d_0)$. At tme a_1, the next task T_1 arrives

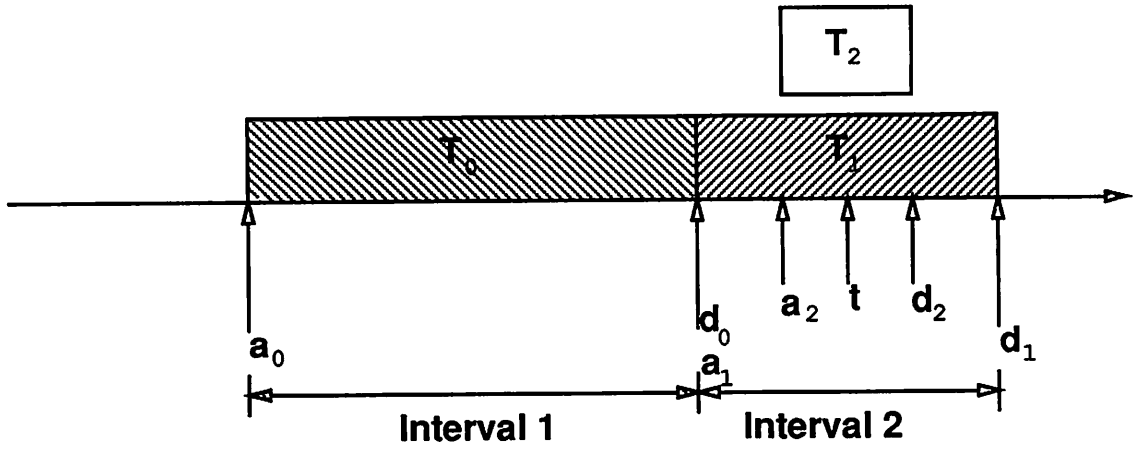

Figure 2: A closed interval and an open interval.

and it is also served at once. The system opens the second interval. At time a_2, T_2 arrives and is rejected. At current time t, T_1 is still running, so this interval remains open, and the expected end of the interval is d_1. Interval 2 immediately follows Interval 1.

Example 4: Figure 3 shows two closed intervals which partially overlap each other. In the first interval, T_1 is discarded and the processor becomes idle after T_0 finishes, so the interval is $[a_0, d_1)$. When T_2 arrives at time a_2, the system is idle, so T_2 gets service at once and the second interval starts before Interval 1 ends at d_1 according Definition 4. The new interval, $[a_2, d_3)$,involves T_2 and T_3. Therefore, the second interval overlaps the idle portion of the first interval.

In the proofs of the TD_1 algorithm, the following sequences are used. Let Δ be an arbitrary interval. The size of Δ depends on the tasks involved. The arrival pattern of these involved tasks determines how a sequence of intermediate open intervals grows to the final closed interval Δ. Let

$$(T_{a_1}, T_{a_2}, T_{a_3}, ..., T_{a_n}) \tag{13}$$

be a list of n tasks considered in Δ by the algorithm according the time sequence and

$$(\Delta_{a_1}, \Delta_{a_2}, \Delta_{a_3}, ..., \Delta_{a_n}) \tag{14}$$

be the corresponding interval list, where each Δ_{a_i} is an interval when T_{a_i} is considered by the algorithm and $\Delta_{a_n} = \Delta$. Let

$$(T_1, T_2, T_3, ..., T_k) \tag{15}$$

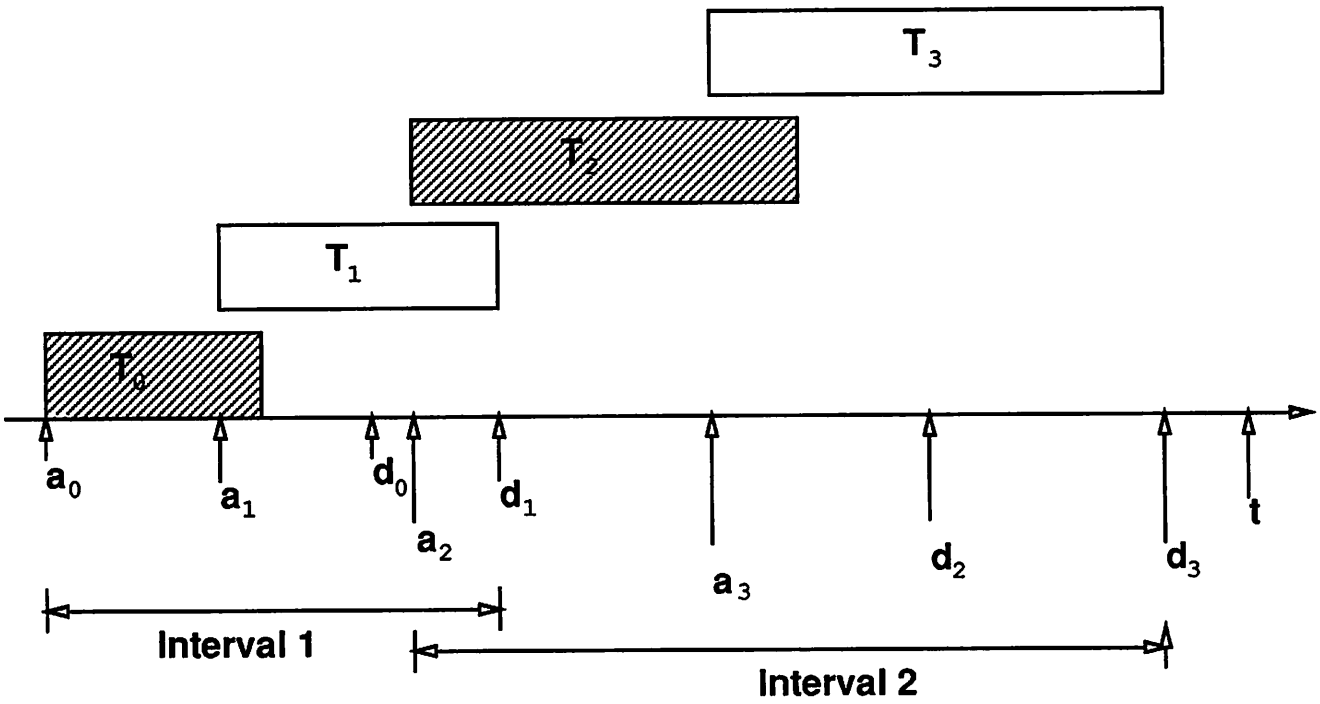

Figure 3: Two partially overlapping intervals.

be a list of tasks executed by the system where each T_i aborts another task, $1 < i \leq k$, and

$$(\Delta_1, \Delta_2, \Delta_3, ..., \Delta_k) \tag{16}$$

be the corresponding interval list. Both sequence in (14) and sequence in (16) are monotonically increasing.

Now we are ready to describe the TD_1 algorithm, which can guarantee a performance ratio of 1/4. TD_1 is used to prove that there is an on-line algorithm with a lower bound of 1/4, which, in turn, is used to show that 1/4 is the tight bound of the uniprocessor on-line scheduling problem. Hence, TD_1 is very simplified. But the algorithm can be easily expanded to further improve its average performance.

In the version 1 of TD_1, we assume that all tasks have zero laxity. Therefore, a scheduling decision must be made whenever a new task arrives. Its pseudo code is shown in Figure 4, where Δ_{run} records the current interval and v_{run} is the value of a running task. v_{run} is set to zero when the system is idle. The correctness of the version 1 of TD_1 guaranteeing a performance bound of 1/4 is based on the following lemma which shows that TD_1 obtains at least 1/4 of the value obtained by a clairvoyant algorithm in each interval.

```
whenever T_next arrives {
    update(Δ_run);
    if (v_run < Δ_run/4) { T_run = T_next; }
}
```

Figure 4: On-Line Scheduling Algorithm TD_1 (version 1) .

Lemma 2: In any interval, the version 1 of TD_1 obtains at least 1/4 of the value obtained by a clairvoyant algorithm.

Proof. Let Δ be an arbitrary interval. Let sequences in (13), (14), (15), and (16) be defined as before, where the sequence in (14) is corresponding to the values of Δ_{run} in the algorithm. We first use mathematical induction on the number of task involved in the task sequence in (15) to prove

$$v_k > \Delta_k/2. \tag{17}$$

1. *Basis of induction.* For $k = 1$, it is trivial, because $v_1 = \Delta_1$ $(> \Delta_1/2)$.
2. *Induction step.* Assume that $k = i$,

$$v_i > \Delta_i/2. \tag{18}$$

Because T_i is aborted by T_{i+1},

$$v_i < \Delta_{i+1}/4 \tag{19}$$

and

$$\Delta_{i+1} < \Delta_i + v_{i+1}. \tag{20}$$

Applying (20), (19), and (18) in order,

$$\begin{aligned} 2v_{i+1} &> v_{i+1} + (\Delta_{i+1} - \Delta_i) \\ &> v_{i+1} + 4v_i - \Delta_i \\ &> v_{i+1} + 2\Delta_i - \Delta_i \\ &= v_{i+1} + \Delta_i \\ &> \Delta_{i+1}. \end{aligned}$$

Thus the inequality (17) is true for any integer k.

To show that the version 1 of TD_1 obtains at least 1/4 of the value obtained by a clairvoyant algorithm in each interval, we consider two cases.

```
whenever (idle && not-empty(Q)) {
    T_run = dequeue(Q); p_loss = v_run;
}
whenever (running && alarm(Q)) {
    T_next = dequeue(Q); update(Δ_run);
    if (v_run < (Δ_run + p_loss)/4)
       {T_run = T_next; }
}
```

Figure 5: On-Line Scheduling Algorithm TD_1 (version 2).

- Case 1: $T_{a_n} = T_k$, which means that no other tasks arrive after T_k. By the inequality (17),

$$v_k > \Delta_k/2 = \Delta/2 > \Delta/4.$$

- Case 2: $T_{a_n} \neq T_k$, which means that there are tasks arrived after T_k and discarded. By the threshold rule,

$$v_k \geq \Delta_{a_n}/4 = \Delta/4.$$

□

In the next version, tasks may have laxities, but all preempted tasks are discarded. A queue, Q, is used to hold tasks waiting for service and they are sorted by latest start times in non-decreasing order. Figure 5 shows the pseudo code of the version 2 of TD_1. Whenever a new task arrives, it is inserted in Q first. If the system is idle, it will execute the first task in Q. Otherwise, a scheduling decision is made when the latest start time of the first task in Q is equal to the current time (alarm(Q) becomes true). Hence, the first task in any interval may have laxity at the time it is started. All other tasks have zero laxity because they are decided at the queue alarm time. If a non-zero-laxity task is completed by TD_1 and some other tasks (with zero laxity at the times they are considered) are discarded in the interval, then there exists a case such that a clairvoyant algorithm can execute this non-zero-laxity task outside the interval and obtain the value of other tasks by executing them inside the interval. To deal with this problem, the algorithm uses a variable, *potential_loss* (*p_loss*), as a compensation. *p_loss* records the value of the first task in any interval, which is the only task may have laxity. The correctness of the version 2 of TD_1 guaranteeing a performance bound of 1/4 is based on the following lemma which shows that TD_1 obtains at least 1/4 of the value obtained by a clairvoyant algorithm in each interval when a compensation added.

Lemma 3: The version 2 of TD_1 obtains at least 1/4 of the value obtained by a clairvoyant algorithm in any interval with considering the compensation.

Proof. As in the proof of lemma 2, we can show the following inequality is true by mathematical induction in a similar way:

$$v_k > (\Delta_k + p\_loss_k)/2, \tag{21}$$

We omit the detail proof here. To show that the version 2 of TD_1 obtains at least 1/4 of the value obtained by a clairvoyant algorithm in each interval when a compensation is considered. we again consider two cases.

- Case 1: $T_{a_n} = T_k$, which means that no other tasks compete with T_k. By the inequality (21),

$$v_k > (\Delta_k + p\_loss)/2 = (\Delta + p\_loss)/2 > (\Delta + p\_loss)/4.$$

- Case 2: $T_{a_n} \neq T_k$, which means that some tasks compete with T_k and are discarded. The threshold rule guarantees that

$$v_k \geq (\Delta_{a_n} + p\_loss)/4.$$

□

It is enough to use the version 2 of TD_1 to demonstrate the performance bound of 1/4 is a tight bound for the uni-processor on-line scheduling problem. However, when it is compared with EDF, there is still a small problem. If the system is underloaded, EDF has a performance bound of 1 while TD_1 has 1/4. It is better to design an algorithm having a performance bound of 1 under non-overloads and 1/4 under overloads. This is motivation for the design of the version 3 of TD_1.

The new algorithm uses the Earliest Deadline First (EDF) rule under non-overloads. Therefore, TD_1 guarantees a performance bound of 1 under non-overloads. Both Least Laxity First (LLF) and Least Latest Start Time First (LLSTF) can also be used although we only consider EDF here. A queue, Q, is used to hold tasks waiting for service and they are sorted by deadlines. Because the EDF rule, some tasks in Q may have been executed partially. These tasks are called *fragment tasks*, while other tasks are called *regular tasks*. The interval definition is expanded as following accordingly.

Definition 6: A *time interval* or simply *interval* at t is a time segment $[t_b, t_e)$ and consists of a busy subsegment $[t_b, t_f]$ followed by an optional idle subsegment $[t_f, t_e)$. It satisfies all the following conditions:

- t_b ($t_b \leq t$) is the time the system transits from an idle state to a running state or the time the system switches from a running task to a regular task with both tasks feasible;
- In $[t_b, t_f]$, there is no such a switch from a running task to a regular task with both tasks feasible;
- t_f ($t \leq t_f$) is the time the system transits (or is expected to transit) back from a running state to an idle state, because some tasks complete (or is expected to complete);
- All fragment tasks in Q are feasible each other if they start after t_e; *and*
- $t_e = \max(t_f, \max(\{dl_{fragment}\}), \max(\{dl_{discarded}\}))$, where $\{dl_{fragment}\}$ are the deadlines of all fragment tasks which are either discarded or completed and $\{dl_{discarded}\}$ are the deadlines of all tasks discarded during the time subsegment $[t_b, t_f]$.

An interval is *closed* when a task completes and all involved fragment tasks are either completed or discarded, otherwise it is *open*.

There are some features in the new algorithm. Whenever a task from Q starts execution, the algorithm guarantees that remaining fragment tasks in the Q are feasible if they start after the end point of the current interval. If an interval is underloaded, only one task is actually involved in each interval. The tasks are completed in the order of their deadlines (the EDF rule). An open underloaded interval may be aborted when the system switches to a new arrival task with a smaller deadline and both tasks are feasible. The preempted task becomes a fragment task and is put back to Q. The time consumed in the aborted interval will be counted as the compensation in a future interval. If an interval is overloaded, there are more than one tasks involved in the interval. Let T_{run} be a running task and T_{next} be a regular task in the queue. If $l_{next} < t_f$, the algorithm makes a scheduling decision at l_{next}. Some fragment tasks, $\mathbf{T}_{frag}$, may also conflict with T_{next}, which means that $\mathbf{T}_{frag}$ is a minimum subset of the fragment tasks removed from Q such that the T_{next} and the remaining fragment tasks in Q are feasible. Therefore, the algorithm chooses either T_{run} and $\mathbf{T}_{frag}$ or T_{next} based on a threshold rule. If T_{next} is discarded, the algorithm is expected to complete T_{run} and $\mathbf{T}_{frag}$ in the current interval. If T_{next} wins, then T_{run} and $\mathbf{T}_{frag}$ are discarded. The algorithm maintains a conflict task set, $\mathbf{T}_{conflict}$, which consists of all $\mathbf{T}_{frag}$ involved in the interval and they have not yet been discarded

```
choose_one(T_run, T_next, Q)
{
    "compute T_conflict;"
    update(p_loss); update(Δ_run);
    if ("Rule in (22) is false")
       { T_run = T_next; discard(T_conflict); }
    return(T_run);
}
/* The event-triggered routines */
whenever (idle && not-empty(Q)) {
    T_next = remove_edf(Q);
    T_run = choose_one(Null, T_next, Q);
}
whenever (running && alarm(Q)) {
    T_next = remove_alarmed_task(Q);
    T_run = choose_one(T_run, T_next, Q);
}
whenever (finish) {
    if (empty(T_conflict))
       "set system state to idle";
    else T_run = remove_edf(T_conflict);
}
whenever (arrival) {
    if ( (d_run ≤ d_arr) || idle )
       insert(T_arr, Q);
    else if (feasible (T_run, T_arr, "all fragments")) {
       insert(T_run, Q);
       T_run = T_arr;
       "start a new interval, Δ_run";
    } else
       T_run = choose_one(T_run, T_arr, Q);
}
```

Figure 6: On-Line Scheduling Algorithm TD_1 (version 3).

or executed. When a running task completes, the system executes a fragment task from $\mathbf{T}_{conflict}$ immediately if it is not empty, otherwise, the interval is closed at this moment.

Figure 6 is the pseudo code of the version 3. It applies the following threshold rule under overloads:

$$\frac{gain + v_{run} + Value(\mathbf{T}_{conflict})}{\Delta_{run} + p\_loss} \geq \frac{1}{4}, \tag{22}$$

where *gain* is the value obtained from the tasks which have been completed in the interval, $Value(\mathbf{T}_{conflict})$ is the value obtained by executing $\mathbf{T}_{conflict}$, Δ_{run} is the current interval size,

and *p_loss* is the sum of the previously aborted intervals which relates to all involved fragment tasks in the current interval.

The correctness of the version 3 of TD_1 guaranteeing a performance bound of 1/4 under overloads and 1 under non-overloads is based on the following lemma.

Lemma 4: In any interval, the version 3 of TD_1 has a performance bound 1 under non-overloads and 1/4 under overloads with considering the compensation.

Proof. Let Δ be an arbitrary interval. Let sequences in (13), (14), (15), and (16) be defined as before, where the sequence in (14) is corresponding to the values of Δ_{run} in the algorithm. It is possible that there are more than one task completed in the task sequence in (15). We use mathematical induction on the number of task involved in the task sequence in (15) to prove:

$$v_k > (\Delta_k + p\_loss_k)/2, \tag{23}$$

1. *Basis of induction.* For $k = 1$, it is trivial, because $v_1 = \Delta_1 + p\_loss_1$.
2. *Induction step.* Assume that $k = i$,

$$v_i > (\Delta_i + p\_loss_i)/2. \tag{24}$$

Because the system switches to T_{i+1}, the threshold rule in (22) is false, which implies:

$$v_i + V_{i+1} < (\Delta_{i+1} + p\_loss_{i+1})/4, \tag{25}$$

where V_{i+1} is corresponding to $Value(\mathbf{T}_{conflict})$ in (22). With the definition of the interval and the property of the compensation, we have

$$\Delta_{i+1} < \Delta_i + v_{i+1} + V_{i+1} \tag{26}$$

and

$$p\_loss_i + V_{i+1} \geq p\_loss_{i+1} \tag{27}$$

Applying (26), (25), (24) and (27) in order,

$$\begin{aligned}
2v_{i+1} &> v_{i+1} + (\Delta_{i+1} - \Delta_i - V_{i+1}) \\
&> v_{i+1} + (4(v_i + V_{i+1}) - p_loss_{i+1}) - \Delta_i - V_{i+1} \\
&> v_{i+1} + 2(\Delta_i + p_loss_i) - p_loss_{i+1} - \Delta_i + 3V_{i+1} \\
&= (\Delta_i + v_{i+1} + V_{i+1}) + 2(p_loss_i + V_{i+1}) - p_loss_{i+1} \\
&\geq \Delta_{i+1} + 2p_loss_{i+1} - p_loss_{i+1} \\
&= \Delta_{i+1} + p_loss_{i+1}.
\end{aligned}$$

Thus the inequality (23) is true for any integer k.

To show that the version 3 of TD_1 obtains at least 1/4 of the value obtained by a clairvoyant algorithm in each interval with the compensation p_loss, we consider two cases.

- Case 1: $T_{a_n} = T_k$, which means that no other tasks arrive after T_k. By the inequality (23),

$$v_k > (\Delta_k + p\_loss_k)/2 > (\Delta_k + p\_loss_k)/4.$$

- Case 2: $T_{a_n} \neq T_k$, which means that the system does not make any preemption after T_k in this interval. The performance bound of 1/4 is guaranteed by the threshold rule in (22). □

Combining the above results, we prove Theorem 1.

Theorem 1: If all tasks have the same value density, then the upper bound of the uniprocessor on-line scheduling problem is 1/4.

6 Uniprocessor On-Line Scheduling for Tasks with Arbitrary Value Densities

In this section, we generalize the result of the last section to the case in which tasks do not have the same value density.

Let γ be the ratio between the highest value density and the lowest value density of tasks. The actual value densities of tasks can be mapped to $[1, \gamma]$. As in the proof of Lemma 1 in last section, an adversary uses both α-tasks and τ-tasks to build a worst case pattern of task request sequence. The adversary assigns a value density 1 to all α-tasks and a value density γ to all τ-tasks. The computation times of the α-tasks are defined by the following recurrence relation:

$$c_{k+2} = (\beta - \gamma + 1)c_{k+1} - \beta c_k \tag{28}$$

with the boundary conditions as

$$c_0 = 1$$

and

$$c_1 = \beta - \gamma.$$

The constant ratio property mentioned in Section 4 can be generalized to the following form:

Property 1: [Constant Ratio Property]

$$\frac{c_{k-1}}{\gamma \sum_{j=0}^{k-1} c_j + c_k} = \frac{1}{\beta}. \tag{29}$$

Proof. We use mathematical induction to prove

$$\gamma \sum_{j=0}^{k-1} c_j + c_k = \beta c_{k-1}.$$

1. *Basis of induction.* For $k = 1$, we have

$$\gamma c_0 + c_1 = \gamma + (\beta - \gamma) = \beta = \beta c_0.$$

2. *Induction step.* Assume that

$$\gamma \sum_{j=0}^{k-1} c_j + c_k = \beta c_{k-1}.$$

We have

$$\begin{aligned}
\gamma \sum_{j=0}^{k} c_j + c_{k+1} &= (\gamma \sum_{j=0}^{k-1} c_j + \gamma c_k) + ((\beta - \gamma + 1)c_k - \beta c_{k-1}) \\
&= (\gamma \sum_{j=0}^{k-1} c_j + c_k) + \beta c_k - \beta c_{k-1} \\
&= \beta c_{k-1} + \beta c_k - \beta c_{k-1} \\
&= \beta c_k.
\end{aligned}$$

Hence, Equation (29) is true for all $k \geq 1$. □

Lemma 5: Given γ, there exists task request sequence pattern, P', such that, no on-line scheduling algorithm can get a performance ratio higher than $1/(\gamma + 1 + 2\sqrt{\gamma})$ compared to a clairvoyant algorithm.

Proof. The adversary uses a sequence of the α-tasks defined by the recurrence relation (28) and presents them to the on-line scheduling algorithm in the similar way as before, so that whenever the "game" is stopped, the performance ratio will never be larger than $1/\beta$.

The adversary wants the α-task sequence to have the following two features:

1. the "game" always stops; and

2. β is as large as possible.

The first feature requires that the value of the α-task sequence should oscillate. The characteristic equation for (28) is

$$x^2 - (\beta - \gamma + 1)x + \beta = 0.$$

The oscillation occurs when the characteristic roots are complex numbers, which happens if

$$(\beta - \gamma + 1)^2 - 4\beta < 0. \tag{30}$$

The inequality of (30) can be written as

$$(\beta - (\gamma + 1 + 2\sqrt{\gamma}))(\beta - (\gamma + 1 - 2\sqrt{\gamma})) < 0,$$

which gives

$$\gamma + 1 - 2\sqrt{\gamma} < \beta < \gamma + 1 + 2\sqrt{\gamma} \tag{31}$$

and

$$\gamma + 1 - 2\sqrt{\gamma} > \beta > \gamma + 1 + 2\sqrt{\gamma}. \tag{32}$$

(32) is a contradiction and is discarded.
Therefore the largest possible value of β is $(\gamma + 1 + 2\sqrt{\gamma}) - \epsilon$, which will be used by the adversary to construct the α-task sequence. Thus, the adversary has a strategy to force every on-line algorithm to stop with a performance ratio not higher than $1/(\gamma + 1 + 2\sqrt{\gamma})$ compared to a clairvoyant algorithm. □

Lemma 6: Given γ, there exists an on-line scheduling algorithm, TD_1', for all input sequences of tasks, it has a performance ratio at least $1/(\gamma+1+2\sqrt{\gamma})$ compared to a clairvoyant algorithm.

Proof. Similar to the proofs of Lemma 2, 3, and 4 in last section, we may construct an algorithm TD_1' from TD_1 by using a new threshold, which is

$$\frac{1}{\gamma + 1 + 2\sqrt{\gamma}}.$$

Using the same argument as before, it is easy to show that TD_1' has the desired lower bound. □

Combining the above two lemmas, we prove Theorem 2.

Theorem 2: If γ is the ratio between the highest and the lowest value density of tasks, then the upper bound of the uniprocessor on-line scheduling problem is

$$\frac{1}{\gamma + 1 + 2\sqrt{\gamma}}.$$

7 Multiprocessor On-Line Scheduling for Tasks with the Same Value Density

We may further generalize the previous results from uniprocessor to multiprocessor. Some important hints are provided in this section while the details will be report in another place. The case for dual-processors is considered first, followed by the case for multiprocessors.

Comparing to uniprocessor on-line scheduling, dual-processor on-line scheduling is a very interesting problem. The reason is that two processors can cooperate with each other to guarantee that at least one processor does some productive work. This means that, under the cooperations, the performance bound is at least 1/2, comparing 1/4 for uniprocessor, which translates into a 200% performance bound improvement. For the designers of real-time systems, this will be an important reason to choose a dual-processor based system instead of a uniprocessor based system.

Turning our attention to multiprocessors, our main strategy is to group processors. If the number of processors is even, we may simply combine every two processors to form a group. Therefore, it is easy to see that the upper bound is the same as in dual-processor on-line scheduling. If the number of processors is odd, one processor is left after the others are combined into dual-processor groups. Each group can guarantee its performance bound to be 1/2. The remaining processor is only able to guarantee its performance bound to be 1/4. The upper bound is derived by simply combining these two facts.

8 Conclusions

In this paper we discuss the upper bound for any on-line scheduling algorithm in a real-time environment, in which the overload must be handled quickly and effectively. If all tasks have the same value density, the upper bound for the uniprocessor on-line scheduling problem is 1/4. If tasks have different value densities and the ratio between the highest and the smallest value density is γ, the upper bound for the uniprocessor on-line scheduling problem is $1/(\gamma+1+2\sqrt{\gamma})$.

We have also presented the on-line scheduling algorithms, TD_1 and TD_1', to reach these two upper bounds respectively. TD_1 and TD_1' use a simple threshold rule to make on-line decisions during the system overload periods. They can be easily implemented and further optimized. The upper bound is doubled from 1/4 in uniprocessors to 1/2 in dual-processors, which means that, in the worst case, the value obtained from a dual-processor system is twice of the value obtained from two separate uniprocessor systems. For the designers of real-time systems, this will be an important reason to choose a dual-processor based system instead of a uniprocessor based system.

References

[1] S. K. Baruah and L. E. Rosier. Limitations concerning on-line scheduling algorithms for overloaded real-time systems. *8th IEEE Workshop on Real-Time Operating Systems and Software*, pages 128–132, 1991.

[2] S. Ben-David, A. Borodin, R. M. Karp, G. Tardos, and A. Wigderson. On the power of randomization in online algorithms. In *Proceedings of the 22nd Annual ACM Symposium on Theory of Computing*, pages 379–386, 1990.

[3] M. Bern, D. H. Greene, A. Raghunathan, and M. Sudan. Online algorithms for locating checkpoints. In *Proceedings of the 22nd Annual ACM Symposium on Theory of Computing*, pages 359–368, 1990.

[4] S. Biyabani, J. A. Stankovic, and K. Ramamritham. The integration of deadline and criticalness in hard real-time scheduling. In *Proc. Real-Time Systems Symposium*, 1988.

[5] A. Borodin, N. Linial, and M. Saks. An optimal online algorithm for metrical task system. In *Proceedings of the 18th Annual ACM Symposium on Theory of Computing*, pages 373–382, 1987.

[6] E. G. Coffman, M. R. Garey, and D. S. Johnson. Dynamic bin packing. *SIAM J. Comput.*, 12(2):202–208, 1983.

[7] D. Coppersmith, P. Doyle, P. Raghavan, and M. Snir. Random walks on weighted graphs, and applications to on-line algorithms. In *Proceedings of the 22nd Annual ACM Symposium on Theory of Computing*, pages 369–378, 1990.

[8] M. L. Dertouzos and A. K. Mok. Multiprocessor on-line scheduling of hard-real-time tasks. *IEEE Trans. on Software Engineering*, SE-15(12):1497–1505, 1989.

[9] R. M. Karp, U. V. Vazirani, and V. V. Vazirani. An optimal algorithm for on-line bipartite matching. In *Proceedings of the 22nd Annual ACM Symposium on Theory of Computing*, pages 352–369, 1990.

[10] G. Koren, B. Mishra, A. Raghunathan, and D. Shasha. A competitive on-line algorithm for overloaded real-time systems. *Unpublished, 20 pages*, March 1991.

[11] C. Liu and J. Layland. Scheduling algorithms for multi-programming in a hard-real-time environment. *J. ACM*, 20:46–61, 1973.

[12] C. L. Liu. *Introduction to Combinatorial Mathematics.* McGraw-Hill, Inc., 1968.

[13] C. D. Locke. *Best-effort decision making for real-time scheduling.* PhD thesis, Carnegie Mellon University, 1986.

[14] M. S. Manasse, L. A. McGeock, and D. Sleator. Competitive algorithms for on-line problems. In *Proceedings of the 20th Annual ACM Symposium on Theory of Computing*, pages 322–333, 1988.

[15] A. K. Mok and M. L. Dertouzos. Multiprocessor scheduling in a hard real-time environment. In *Proceedings of the Seventh Texas Conference on Computing System*, 1978.

[16] K. Ramamritham and J. A. Stankovic. Dynamic task scheduling in distributed hard real-time systems. *IEEE Software*, 1(3), July 1984.

[17] D. Sleator and R. Tarjan. Amortized efficiency of list update and paging rules. *CACM*, 28(2):202–208, 1985.