UNITE DE RECHERCHE
INRIA-ROCQUENCOURT

Institut National
de Recherche
en Informatique
et en Automatique

Domaine de Voluceau
Rocquencourt
B.P.105
78153 Le Chesnay Cedex
France
Tél.:(1) 39 63 55 11

Rapports Techniques

1992

ème
anniversaire

N° 138

*Programme 1*
*Architectures Parallèles, Bases de Données, Réseaux et Systèmes*

# D-OVER : AN OPTIMAL ON-LINE SCHEDULING ALGORITHM FOR OVERLOADED REAL-TIME SYSTEMS

Gilad KOREN
Dennis SHASHA

Février 1992

*RT_0138*

# D-Over: an optimal on-line scheduling algorithm for overloaded real-time systems

Gilad Koren, koren@cs.nyu.edu    Dennis Shasha, shasha@cs.nyu.edu

INRIA, Rocquencourt, projet Rodin, batiment 9
and Courant Institute, New York University
New York, NY 10012.

February 5, 1992

**Abstract**

*Every task in a real-time system has a deadline by which time it should complete. Each task also has a value that it obtains only if it completes by its deadline. The problem is to design an on-line scheduling algorithm (i.e., the scheduler has no knowledge of a task until it is released) that maximizes the obtained value.*

*When such a system is underloaded (i.e. there exists a schedule for which all tasks meet their deadlines), Dertouzos showed that the earliest deadline first algorithm will achieve 100% of the possible value. Locke showed that earliest deadline first performs very badly when the system is overloaded and proposed heuristics to deal with overload.*

*This paper presents an optimal on-line scheduling algorithm for overloaded systems. It is optimal in the sense that it gives the best competitive factor possible relative to an offline scheduler.*

**Mots Clés:** *temps réel, ordonnancement, surchargé.*

## D-Over: un algorithme d'ordonnancement actif et optimal pour systèmes temps réels surchargés

*Dans un système temps réel, toute tâche a une date limite à laquelle elle doit être terminée. De plus, chaque tâche a une valeur qu'elle n'obtient que si elle se termine à la date limite. Le problème est de concevoir un algorithme d'ordonnancement actif (on-line) qui maximise la valeur obtenue (i.e., tout en ayant aucune connaissance de la tâche a priori).*

*Lorsqu'un tel système est sous-chargé (i.e., il existe un ordonnancement par lequel toutes les tâches terminent à leur date limite), Dertouzos a montré que l'agorithme "earliest deadline first" atteint 100% de la valeur possible. Cependant, Locke a montré que cet algorithme avait de très mauvaises performances lorsque le système devennait surchargé. Il a proposé des heuristiques dans le cas de surcharge.*

*Ce papier présente un algorithme d'ordonnancement optimal pour systèmes temps réels surchargés. Il est optimal dans le sens où il offre le meilleur facteur de compétition possible relatif à un ordonnanceur passif (off-line).*

# 1 Introduction

In real-time computing systems, correctness may depend on the completion time of tasks as much as on their input/output behavior. Tasks in real-time systems have deadlines. If the deadline for a task is met, then the task is said to *succeed*. Otherwise it is said to have *failed*.

A system is *underloaded* if there exists a schedule that will meet the deadline of every task and *overloaded* otherwise. Scheduling underloaded systems is a well-studied topic, and several on-line algorithms have been proposed for the optimal scheduling of these systems on a uniprocessor [4, 11]. Examples of such algorithms include *earliest-deadline-first* (D) and *smallest-slack-time* (SL). However, none of these classical algorithms make performance guarantees during times when the system is overloaded. In fact, Locke has experimentally demonstrated that these algorithms perform quite poorly when the system is overloaded [10].

Practical systems are prone to intermittent overloading caused by a cascading of exceptional situations. A good on-line scheduling algorithm, therefore, should give a performance guarantee in overloaded as well as underloaded circumstances.

Researchers and designers of real-time systems have devised on-line heuristics to handle overloaded situations [1, 10, 12]. Locke proposed several clever heuristics as part of the CMU Archons project [10]. Unfortunately, those offer no performance guarantee. This paper proposes an algorithm with strong performance guarantees for a large portion of the parameters that Locke's algorithm considers.

## 1.1 Background

Real-time systems may be categorized by how they react when a task fails. In a *hard real-time system*, a task failure is considered intolerable. The underlying assumption is that a task failure would result in a disaster, e.g. a fly-by-wire aircraft may crash if the altimeter is read a few milliseconds too late.

A less stringent class of systems is denoted as *soft real-time systems*. In such systems, each task has a positive value. The goal of the system is to obtain as much value as possible. If a task succeeds, then the system acquires its value. If a task fails, then the system gains less value from the task [11]. In a special case of soft real-time systems, called a *firm* real-time system, there is no value for a task that has missed its deadline, but there is no catastrophe either. The first algorithm we present here applies to firm real-time systems. The paper then generalizes the algorithm to soft real-time systems.

An *on-line* scheduling algorithm is one that is given no information about a task before its release time. Different tasks models can differ in the kind of information (and its accuracy) given upon release. We assume the following: when a task is released, its value and deadline are known precisely, its computation time may be known either precisely, or, more generally, within some range. Also, preemption is allowed and task switching takes no time.

The *value density* of a task is its value divided by its computation time. The *importance ratio* of a collection of tasks is the ratio of the largest value density to the smallest value density. When the importance ratio is 1, the collection is said to have UNIFORM VALUE DENSITY, i.e., a task's value equals its computation time. We will denote the importance ratio of a collection by $k$.

A natural way to measure a performance guarantee of an on-line scheduler is to compare it with a *clairvoyant* scheduling algorithm. A clairvoyant scheduler has complete A PRIORI knowledge of all the parameters of all the tasks. A clairvoyant scheduler can choose a "scheduling sequence" that will obtain the maximum possible value achievable by any scheduler. (The problem of finding the maximum achievable value for such a scheduler, however, can be shown to be reducible from the knapsack problem[5]; hence is NP-hard.)

As in [2, 6, 13] we say that an on-line algorithm has a *competitive factor* $r$, $0 \leq r \leq 1$, *if and only if* it is guaranteed to achieve a cumulative value at least $r$ times the cumulative value achievable by a clairvoyant algorithm on *any* set of tasks.

Three years ago, Marc Donner introduced a group of us to realtime scheduling in a seminar at NYU. Inspired by that seminar, Koren, Mishra, Raghunathan and Shasha [2, 7] suggested the first on-line scheduling algorithm with a performance guarantee for an overloaded system. They assumed a simplified variation of the task model that assumes *firm deadline, preemption at no cost* and *uniform value density.* This algorithm is called *D-star* ($D^*$) since it behaves like *earliest-deadline-first* (D) in an underloaded situation.

$D^*$ executes to completion all the tasks with deadlines in underloaded intervals. $D^*$ also guarantees that all the tasks with a deadline in an overloaded interval will achieve a cumulative value of at least *one-fifth* of the *length* of the overloaded interval. However, $D^*$ has a competitive factor of zero.

Baruah et. al. [2, 3] demonstrated, using an adversary argument that, in the uniform value density setting, there can be NO on-line scheduling algorithm with a competitive factor greater than one-quarter.

Koren and Shasha describe in a technical report [8] an algorithm called *DD-star* ($DD^*$), that has the competitive factor of one-fourth in the uniform value density case and offers 100% of the possible value in the underloaded case. This showed that the one-quarter bound is tight in the uniform value density case. Wang and Mao [15] independently report a similar guarantee.

On the complexity side, Baruah et. al. [2, 3] showed for environments with an importance ratio $k$, a bound of $\frac{1}{(1+\sqrt{k})^2}$ on the best possible competitive factor of an on-line scheduler. This result and some pragmatic considerations reveal the following limitations of the algorithms with competitive factors invented to date:

1. The algorithms all assume a uniform value density, yet some short tasks may be more important than some longer tasks.

2. The algorithms all assume that there is no value in finishing a task after its deadline. But a slightly late task may be useful in many applications.

3. The algorithms all assume that the computation time is known upon release. However, a task program that is not straight-line may take different times during different executions.

## 2 The Main Results

In this paper we present an on-line scheduling algorithm called $D^{over}$ that has an optimal competitive factor of $\frac{1}{(1+\sqrt{k})^2}$ for environments with importance ratio $k$. Hence, we show that the bound in [2, 3] is tight for all $k$.

Furthermore, $D^{over}$ achieves 100% of the value of in an underloaded environment. In fact the performance guarantee of $D^{over}$ is even stronger: $D^{over}$ schedules to completion all tasks in underloaded periods and achieves at least $\frac{1}{(1+\sqrt{k})^2}$ of the value a clairvoyant algorithm can get during overloaded periods[1].

We also investigate two important extensions to the task model studied earlier:

- Gradual Descent:

  We relax the *firm deadline* assumption. Tasks that complete after their deadline can still have a positive value though less than their initial value. As in Locke [10], the task's value is given by a *value function* which depends on its completion time.

  We show that under a variety of value functions an appropriate version of $D^{over}$ has a competitive factor of $\frac{1}{(1+\sqrt{k})^2}$ for environments with importance ratio $k$.

- Situations in which the exact computation time of a task is not known

  Suppose the on-line scheduling algorithm does not know the exact computation time of a task upon its release. However, for every task $T$, an upper bound on its possible computation time denoted by $c_{max}$, is given. The actual computation time of $T$ denoted by $c$ satisfies:

  $$(1-\epsilon)\cdot c_{max} \leq c \leq c_{max}$$

  Where, $0 \leq \epsilon < 1$ is a given *error margin* which is common to all the tasks.

  We show that in that case $D^{over}$ has a competitive factor of $\frac{1}{(1+\sqrt{k})^2+(\epsilon\cdot k)(1+\sqrt{k})+1}$. We also show that, in this setting a competitive on-line scheduler can not guarantee 100% of the value for underloaded systems.

Finally, $D^{over}$ can be implemented using balanced search trees, and runs at an *amortized* cost of $O(\log n)$ time per task, where $n$ bounds the number of tasks in the system at any instant.

The rest of the paper is organized as follows: Section 3 introduces some notation and definitions used in the paper. Section 4 describes the main algorithm, $D^{over}$. Section 5 shows that $D^{over}$ has the optimal competitive factor as mentioned above. Section 6 defines the notion of *conflicting* and *conflict-free* tasks and proves the complete performance guarantee of $D^{over}$ with respect to underloaded and overloaded periods. Section 7 contains the gradual-descent result. Section 8 is devoted to the model in which the exact computation time of a task is not known. The paper ends with a brief conclusion section and a discussion of open problems.

# 3 Notation

Before we describe the full algorithm, we need some notation. We are given a collection of tasks $T_1, T_2 \cdots T_n$ denoted by $\Gamma$. For each task $T_i$, its value is denoted by $v_i$, its release time is denoted by $r_i$, its computation time by $c_i$ and its deadline by $d_i$. The value density of $T_i$ is denoted by $imp(T_i)$ and $k$ denotes the importance ratio of the collection.

[1] The definitions of underloaded and overloaded periods will be made precise in section 6.

**Definition 3.1**

- UNDERLOADED AND OVERLOADED SYSTEMS: A system is *underloaded* if there exists a schedule that will meet the deadline of every task and *overloaded* otherwise.
- EXECUTABLE PERIOD: The *executable period*, $\Delta_i$, of the task $T_i$ is defined to be the following interval:

$$\Delta_i = [r_i, d_i]$$

By definition, $T_i$ may be scheduled only during its executable period.

Suppose a collection of tasks is being scheduled by some scheduler $S$.

- COMPLETED TASK: A task (successfully) *completes* if before its deadline, the scheduler $S$ gives it an amount of execution time that is equal to its computation time.
- ABANDONED TASK: A task is *abandoned* if it did not complete and will never be scheduled again by $S$.
- PREEMPTED TASK: A task is *preempted* when the processor stops executing it, but then the task might be scheduled again and complete at some later point.
- A READY TASK:

  A task is said to be *ready* at time $t$ if its release time is before $t$, its deadline is after $t$ and it neither completed nor was abandoned before $t$ (the current executing task, if any, is always a ready task).

□

The earliest deadline first algorithm ( hereafter, D) can now be described as follows:

---

At any given moment,
schedule the ready task with the earliest deadline.

---

D THE EARLIEST DEADLINE FIRST SCHEDULING ALGORITHM.

Also, we shall make the following assumption:

**Assumption 3.2**

- TASK MODEL: Tasks may enter the system at any time; their computation times and deadlines are known exactly at their time of arrival (we weaken this assumption of exact knowledge later in section 8). Nothing is known about a task before it appears. □
- TASKS SWITCHING TAKES NO TIME: A task can be preempted and another one scheduled for execution instantly.

Suppose that a collection of tasks $\Gamma$ with importance ratio $k$ is given.

- NORMALIZED IMPORTANCE: Without loss of generality, assume that the smallest importance of a task in $\Gamma$ is 1. Hence if $\Gamma$ has *importance ratio* of $k$, the largest importance of a task in $\Gamma$ is $k$.

# 4 The Algorithm

In the algorithm described below, there are three kinds of *events* (each causing an associated interrupt) considered:

- **Task Completion**: successful termination of a task. This event has the highest priority.
- **Task Release**: arrival of a new task. This event has low priority.
- **Latest-start-time Interrupt**: the indication that a task must immediately be scheduled in order to complete by its deadline, that is the task's remaining computation time is equal to the time remaining until its deadline. This event has also low priority (the same as **Task Release**).

If several interrupts happen simultaneously they are handled according to their priorities. A **Task Completion** interrupt is handled before the **Task Release** and **Latest-start-time** interrupts which are handled in random order. It may happen that a **Task Completion** event suppresses a lower priority interrupt, e.g., the **Task Completion** handler schedules the next task, if this task had just reached its $LST$ then the **Latest-start-time Interrupt** is removed.

At any given moment, the set of ready tasks [2] is partitioned into two disjoint sets. *recently-preempted* tasks and other tasks. Whenever a task is preempted it becomes a *recently preempted* task. However, whenever some task is scheduled as a result of **Latest-start-time Interrupt** <u>all</u> the ready tasks (whether preempted or never scheduled) become other tasks.

$D^{over}$ maintains a special quantity called **availtime**. Suppose a new task is released into the system and its deadline is the earliest among all ready tasks. The value of **availtime** is the maximum computation time that can be taken by such a task without causing the current task or any of the recently-preempted tasks to miss their deadlines.

$D^{over}$ requires three data structures, called **Qrecent**, **Qother** and **Qlst**. Each entry in these data structures corresponds to a task in the system. **Qrecent** contains exactly the recently-preempted tasks and **Qother** contains the *other* tasks. These two structures are ordered by the tasks' deadlines. In addition, the third structure, **Qlst**, contains all tasks (again, not including the current task) but this time they are ordered by their latest-start-times ($LST$).

These data structures support **Insert**, **Delete**, **Min** and **Dequeue** operations.

- The **Min** operation for **Qrecent** or **Qother** returns the entry corresponding to the task with the earliest deadline among all tasks in **Qrecent** or **Qother**. For **Qlst** the **Min** operation returns the entry corresponding to the task with the earliest $LST$ among all tasks in the queue. The **Min** operation does not modify the queue.
- The **Dequeue** operation on **Qrecent** (or **Qother**) deletes from the queue the element returned by **Min**, in addition **Dequeue** deletes this element from **Qlst**. Likewise a **Dequeue** operation on **Qlst** will delete the corresponding element from either **Qrecent**, if it is a recently-preempted task or from **Qother**, if it is an *other* task.

[2] Excluding the currently executing task.

An entry of **Qother** and **Qlst** consists of a single task, whereas an entry of **Qrecent** is a 3-tuple ($T$, Previous-time, Previous-avail) where $T$ is a task that was previously preempted at time Previous-time. Previous-avail is the value of the variable **availtime** at time Previous-time.

All of these data structures are implemented as balanced trees (e.g. 2-3 trees).

---

In the following code, *Now*() is a function that returns the current time. *Schedule*($T$) is a function that gives the processor to task $T$. *Laxity*($T$) is a function that returns the amount of time the task has left until its deadline less its remaining computation time. That is, $laxity(T) = deadline(T) - (now() + remaining\_computation\_time(T))$. $\phi$ denotes the empty set.
This code includes lines manipulating *intervals*. The notion of an interval is needed for purpose of analysis only, so these lines are commented.

---

```
recentval  := 0     (* This will be the running value of recently-preempted tasks  *)
availtime  := ∞
                    (* Availtime will be the maximum computation time that can be taken
                    by a new task without causing the current task or the recently preempted
                    tasks to miss their deadlines. *)
Qlst       := φ     (* All ready tasks, ordered according to their latest start time. *)
Qrecent    := φ     (* The recently preempted tasks ordered by deadline order *)
Qother     := φ     (* All the other tasks ordered by their deadlines. *)
idle       := true  (* In the beginning the processor is idle *)
```

8 loop

Task Completion :

9 if (both Qrecent and Qother are not empty) then

10 *(* Both queues are not empty and contain together all the ready tasks. The ready task with the earliest deadline will be scheduled unless it is a task of* Qother *and it can not be scheduled with* <u>*all*</u> *the recently-preempted tasks. The first element in each queue is probed by the Min operation. *)*

11 $(T_{Qrecent}, t_{prev}, avail_{prev})$ := Min(Qrecent);

12 *(* Computes the current value of* availtime. *This is the correct value because $T_{Qrecent}$ is the task last inserted of those tasks currently in* Qrecent. *The available computation time has decreased by the time elapsed since this element was inserted to the queue. *)*

13 availtime := $avail_{prev} - (now() - t_{prev})$;

14 *(* Probe the first element of* Qother *and check which of the two tasks should be scheduled. *)*

15 $T_{Qother}$ := Min(Qother);

16 if $d_{Qother} < d_{Qrecent}$ and
availtime$\geq$ remaining_computation_time($T_{Qother}$) then

17 (* *Schedule the task from* Qother. *)

18 Dequeue(Qother);

19 availtime:= availtime − remaining_computation_time($T_{Qother}$) ;

20 availtime:= min(availtime, laxity($T_{Qother}$))

21 Schedule $T_{Qother}$;

22 else

23 (* *Schedule the task from* Qrecent. *)

24 Dequeue(Qrecent);

25 recentval := recentval − value($T_{Qrecent}$);

26 Schedule $T_{Qrecent}$;

27 end {if} (* *which task to schedule.* *)

28 else if (Qother is not empty) then

29 (* Qrecent *is empty. The current interval is closed here, $t_{close} = now()$. The first task in* Qother *is scheduled* *)

30 $T_{current}$ := Dequeue(Qother);

31 availtime:= laxity($T_{current}$);

32 (* *A new interval is created with $t_{begin} = now()$.* *)

33 Schedule $T_{current}$;

```
    else if (Qrecent is not empty)

        (* Qother is empty. The first task in Qrecent is scheduled *)

        ($T_{current}, t_{prev}, avail_{prev}$) := Dequeue(Qrecent);
        recentval := recentval - value($T_{current}$);
        availtime := $avail_{prev} - (now() - t_{prev})$;
        Schedule $T_{current}$;
    else

        (* Both queues are empty. The interval is closed here, $t_{close} = now()$. *)

        idle := true;
        availtime:= $\infty$;
    end {if}
end (* task completion *)


Task Release :
(* $T_{arrival}$ is released. *)

    if (idle ) then
        $T_{current}$ := $T_{arrival}$;
        Schedule $T_{current}$;
        availtime:= laxity($T_{current}$);
        idle := false;
          (* A new interval is created with $t_{begin} = now()$. *)
    else (* $T_{current}$ is executing *)
        if $d_{arrival} < d_{current}$ and
        availtime$\geq$ computation_time($T_{arrival}$) then

          (* No overload is detected, so the running task is preempted. *)

          Insert $T_{current}$ into Qlst;
          Insert ($T_{current}$, $now()$, availtime) into Qrecent;

          (* The inserted task will be, by construction, the task with the earliest deadline
          in Qrecent*)

          availtime:= availtime - remaining_computation_time($T_{arrival}$) ;
          availtime:= min(availtime, laxity($T_{arrival}$))
          recentval := recentval + value($T_{current}$);
          $T_{current}$ := $T_{arrival}$;
          Schedule $T_{current}$;
        else (* $T_{arrival}$ has later deadline or availtime is not big enough. *)
          Insert $T_{arrival}$ into Qlst and Qother;
        end {if}
    end {if} (* idle *)
end (* release *)
```

69 Latest-start-time Interrupt :

70 (* *The processor is not idle and the current time is the latest start time of the first task in* Qlst. *)

71 $T_{next}$ = Dequeue(Qlst);

72 if ($v_{next} > (1+\sqrt{k})$ ($v_{current}$ + recentval)) then
73 Insert $T_{current}$ into Qlst and Qother;
74 Remove all recently-preempted tasks from Qrecent and insert them into Qlst and Qother;
75 (* Qrecent $= \phi$ *)
76 recentval := 0;
77 availtime:= 0
78 Schedule $T_{next}$;
79 else (* $v_{next}$ *is not big enough; it is abandoned.* *)
80 Abandon $T_{next}$;
81 end {if}
82 end (* *LST* *)

83 end{loop }

D$^{over}$ : A COMPETITIVE OPTIMAL ON-LINE SCHEDULING ALGORITHM.

## 5 Analysis of D$^{over}$

In order to facilitate the analysis of D$^{over}$ it is convenient to introduce the notation of intervals.

**Definition 5.1** INTERVALS

- INTERVAL: The intervals are created (opened) and closed according to the scheduling decisions of D$^{over}$ and this process is depicted in the code of D$^{over}$ in section 4 above

  When an interval is created (comments 32 and 52 of D$^{over}$) it is considered *open*, meaning that it may be extended, it is closed when a task completes while **Qrecent** is empty (comments 29 and 41). A new interval would be opened when the next task is scheduled. Initially there is no open interval. Hence, the first interval is opened when the processor first becomes non-idle.

  The interval consists of the time between the point it was opened and the point it was closed. We will denote by $I = [t_{begin}, t_{close}]$ an interval $I$ that was opened at $t_{begin}$ and closed at $t_{close}$.

  **Note:** Two intervals may overlap only at their endpoints. Also, the pointwise union of all intervals is exactly the time in which D$^{over}$ was not idle.

□

Suppose that a collection of tasks $\Gamma$ with importance ratio $k$ is given. and $D^{over}$ schedules this collection. When a task is scheduled it can have zero or positive slack time. A task may be preempted and then re-scheduled several times. We will be mainly concerned with the last time a task was scheduled. For the purposes of analyzing $D^{over}$, we will partition the collection of tasks according to the question of whether the task had completed exactly at its deadline or before its deadline or failed.

- Let $F$ (for fail) denote the set of tasks that were abandoned.
- Let $S^p$ (for successful with *p*ositive time before the deadline) denote the set of tasks that completed successfully and that ended some positive time before their deadlines.
- Let $S^0$ (for successful with *0* time before the deadline) denote the set of tasks that completed successfully but ended exactly at their deadlines.

Call a task *order-scheduled* if it was scheduled by the **Task Completion** or **Task Release** handlers. Call a task *lst-scheduled* if it was scheduled as a result of a **Latest-start-time Interrupt**. (As mentioned above, a **Latest-start-time Interrupt** is raised on a waiting task when it reaches its latest start time (or $LST$), i.e. the last time when it can start executing and still complete by its deadline).

The first task in each interval is order-scheduled. The subsequent tasks (if any) in this interval may be order-scheduled or lst-scheduled. Proposition 5.1 shows that once a task is lst-scheduled all subsequent tasks of this interval must be lst-scheduled. During an interval several order-scheduled tasks may complete but only one lst-scheduled task can complete (this task will also be the last task that executes in the interval).

**Proposition 5.1** *According to the scheduling of $D^{over}$ once a task is lst-scheduled, then all subsequent tasks, in the current interval, are lst-scheduled.*
PROOF.

Suppose the current task, $T_{current}$, is lst-scheduled and a task, $T_{arrival}$, is released. $T_{arrival}$ will not be scheduled by the **Task Release** handler, because when the current task is lst-scheduled **availtime** equals zero (see statement 77 of $D^{over}$) hence no task can be scheduled by the **Task Release** handler (see statement 54 of $D^{over}$)
□

Let **recentval**($t$) denote [3] **recentval** at time $t$ (see statement 1) and **achievedvalue**($t$) denote the value achieved during the current interval before $t$. For an interval $I$, **achievedvalue**($I$) is the total value obtained during $I$.
We partition the value obtained during $I$ in two different ways:

- **ordervalue** vs. **lstvalue**:

[3]In the following only **recentval** is a variable explicitly manipulated by $D^{over}$. All the others: **zerolaxval**, **poslaxval**, **ordervalue** and **lstvalue** are introduced here to facilitate the analysis. This is way they do not reference algorithm statements.

**ordervalue**($I$) is the total value obtained by order-scheduled tasks that completed during $I$. The value obtained by lst-scheduled tasks is denoted by **lstvalue**($I$) (there is at most one such task in any interval $I$).

- **zerolaxval** vs. **poslaxval**:

  **zerolaxval**($I$) denotes the total value obtained by tasks that completed at their deadlines during $I$ (tasks in $S^0$). The value obtained by tasks that completed before their deadlines is denoted by **poslaxval**($I$).

Hence, for every interval

$$\text{achievedvalue}(I) = \text{ordervalue}(I) + \text{lstvalue}(I) = \text{zerolaxval}(I) + \text{poslaxval}(I)$$

When the index ($I$) is omitted we refer to the entire execution. For example **ordervalue** denotes the total value obtained by order-scheduled tasks summing over all intervals.

**Example 5.2** Before the detailed analysis, let us first study an example of D$^{over}$'s scheduling. Consider the following overloaded collection of six tasks. For notational convenience we will denote the tasks by their deadlines, hence for example $T_{20}$ is a task with deadline at time 20. In this example we assume uniform value density.

| Task | *Release-Time* | *Computation-Time* | *Deadline* | $\Delta_i$ |
|---|---|---|---|---|
| $T_{20}$ | 0 | 6 | 20 | $[0, 20]$ |
| $T_{34}$ | 1 | 26 | 34 | $[1, 34]$ |
| $T_{24}$ | 1 | 20 | 24 | $[1, 24]$ |
| $T_{18}$ | 2 | 5 | 18 | $[2, 18]$ |
| $T_{17}$ | 3 | 2 | 17 | $[3, 17]$ |
| $T_5$ | 4 | 1 | 5 | $[4, 5]$ |

Table 1: THE TASKS FOR EXAMPLE 5.2.

D$^{over}$ schedules the above collection as follows: In the beginning **availtime** is $\infty$ and **Qrecent** is empty.
First, D$^{over}$ schedules $T_{20}$ to run at time 0. **Availtime** is set to 14 since this is $T_{20}$'s laxity.
At time 1, $T_{34}$ is released into the system. Since $T_{34}$'s deadline is not earlier than the current task's ($T_{20}$), $T_{34}$ is inserted into **Qother** (and **Qlst** with $LST$ equal 8). Also at time 1, $T_{24}$ is released. Again, since its deadline is after 20 this task is inserted into **Qother** and **Qlst** with $LST$ equals 4.
At time 2, $T_{18}$ is released. This time the current task is preempted. $T_{20}$ is inserted into **Qrecent** and **Qlst** with $LST$ equals 16. **Availtime** is decremented by the computation time of $T_{18}$. Its new value is 9. The value of **recentval** is set to the value of $T_{20}$ (6).
$T_{18}$ executes for one time unit until time 3, when $T_{17}$ is released. $T_{17}$ is scheduled since its computation time (2) is smaller then **availtime** (9). **Availtime** is decremented by the computation

time of $T_{17}$. Its new value is 7. The value the value of $T_{18}$ (5) is added to **recentval** which becomes 11.
At time 4 two events occur: $T_{24}$ reaches its $LST$ and $T_5$ is released. These events can be handled in any order and we choose to handle the **Latest-start-time Interrupt** first. $T_{24}$ reaches its $LST$ but its value is smaller than twice $(1+\sqrt{k}=2)$ the value of the current task plus **recentval** $(2+11)$. Hence, $T_{24}$ is abandoned. $T_5$ is released and its deadline is earlier than the current task's ($T_{17}$). $T_5$ is scheduled since its computation time is smaller then **availtime**$(1<7)$. $T_5$ has laxity of zero which is smaller than the current **availtime** minus the computation time of $T_5$ (6). Hence, **availtime** is now set to 0 and **recentval** becomes $11+2=13$.
At time 5, $T_5$ completes and since $T_{17}$ is the task with the earliest deadline it is scheduled. **Availtime** is now 6 because this the value of **availtime** when $T_{17}$ was executing (7) minus the time elapsed since it was inserted to **Qrecent** (1). The value of $T_{17}$ is subtracted from **recentval** which becomes $13-2=11$.
The remaining computation time of $T_{17}$ is one unit, hence at time 6 it completes. The next task in **Qrecent** is $T_{18}$ which has a remaining computation time of 4 units. **Availtime** is set to 6 which is value of **availtime** when $T_{18}$ was executing (9) minus the time elapsed since it was inserted to **Qrecent** $((6-3)=3)$ (the value of $T_{18}$ is subtracted from **recentval** which becomes $11-5=6$). However, $T_{18}$ will execute only until 8 when $T_{34}$ reaches its $LST$. The value of $T_{34}$ is big enough to preempt the current task. All tasks from **Qrecent** are moved to **Qother** and **availtime** as well as **recentval** are reset to zero.
The $LST$ of $T_{18}$ is 16 and of $T_{20}$ (the only other task in **Qlst**) is 15. These tasks will generate **Latest-start-time Interrupt** in these respective times, both will be abandoned.
At time 34, $T_{34}$ completes its execution and D$^{over}$ finished scheduling this history. Table 2 summarizes the scheduling decisions of D$^{over}$.

| time | released | preempted ($LST$) | completed | scheduled | availtime | Qrecent | recentval | Qother | *comment* |
|---|---|---|---|---|---|---|---|---|---|
| 0 | | | | | $\infty$ | [] | 0 | [] | |
| 0 | $T_{20}$ | | | $T_{20}$ | $laxity(T_{20})=14$ | [] | 0 | [] | new interval |
| 1 | $T_{34}$ | | | | 14 | [] | 0 | $[T_{34}]$ | $LST$ of $T_{34}$ is 8 |
| 1 | $T_{24}$ | | | | 14 | [] | 0 | $[T_{24},T_{34}]$ | $LST$ of $T_{24}$ is 4 |
| 2 | $T_{18}$ | $T_{20}$ (16) | | $T_{18}$ | $min(14-5,13)$ | $[T_{20}]$ | 6 | $[T_{24},T_{34}]$ | |
| 3 | $T_{17}$ | $T_{18}$ (14) | | $T_{17}$ | $min(9-2,12)$ | $[T_{18},T_{20}]$ | $5+6$ | $[T_{24},T_{34}]$ | |
| 4 | | | | | $min(9-2,12)$ | $[T_{18},T_{20}]$ | $5+6$ | $[T_{34}]$ | $T_{24}$'s $LST$, it is abandoned |
| 4 | $T_5$ | $T_{17}$ (16) | | $T_5$ | $min(7-1,0)$ | $[T_{17},T_{18},T_{20}]$ | $2+5+6$ | $[T_{34}]$ | $T_5$ has no laxity |
| 5 | | | $T_5$ | $T_{17}$ | $7-(5-4)=6$ | $[T_{18},T_{20}]$ | $5+6$ | $[T_{34}]$ | |
| 6 | | | $T_{17}$ | $T_{18}$ | $9-(6-3)=6$ | $[T_{20}]$ | 6 | $[T_{34}]$ | |
| 8 | | $T_{18}$ (15) | | $T_{34}$ | 0 | [] | 0 | $[T_{18},T_{20}]$ | $T_{34}$'s $LST$ |
| 15 | | | | | 0 | [] | 0 | $[T_{18}]$ | $T_{20}$'s $LST$ |
| 16 | | | | | 0 | [] | 0 | [] | $T_{18}$'s $LST$ |
| 34 | | | $T_{34}$ | | 0 | [] | | [] | interval closed |

Table 2: D$^{over}$ SCHEDULING.

So, for this history, $S^0 = [T_5, T_{34}], S^p = [T_{17}]$ and $F = [T_{18}, T_{20}, T_{24}]$. Only three tasks completed their execution and the total value obtained by $D^{over}$ is 29. A clairvoyant scheduler can achieve a value of 34 by scheduling $T_{17}, T_{20}$ and $T_{34}$. Also notice that the system is already overloaded at time 1, but the first time an overload is "detected" by $D^{over}$ is at time 4.
□

Our goal is to show that $D^{over}$ has a competitive factor of $\frac{1}{(1+\sqrt{k})^2}$ for every collection of tasks with importance ratio of $k$. We will start by proving some lemmas about the behavior of $D^{over}$. Then we will try to estimate the best possible behavior of a clairvoyant algorithm by comparison to $D^{over}$. Our basic strategy is to bound from below what $D^{over}$ achieves during each interval. This will lead to a global lower bound over the entire execution. Then, we bound from above what a clairvoyant scheduler can achieve during the entire execution.

## 5.1 Some Lemmas about $D^{over}$'s Scheduling

In this section we present some technical lemmas about the behavior of $D^{over}$. These lemmas will be used in the next section when comparing $D^{over}$'s performance with that of a clairvoyant scheduler. These lemmas concern the relationship between the the interval length and the value achieved by $D^{over}$ in that interval (lemma 5.3). As well as the relationship between the computation time and value of tasks abandoned in an interval with respect to the value achieved in the interval (lemma 5.4 and 5.5).

**Lemma 5.2**

1. *For any task $T_i$ in $S^0$,*

$$\Delta_i = [r_i, d_i] \subseteq BUSY$$

2. *For any task $T_i$ in $F$. Suppose $T_i$ was abandoned at time $t_{aban}$, then*

$$[r_i, t_{aban}] \subseteq BUSY$$

PROOF.

A processor is idle, under $D^{over}$ scheduling, only if there is no *ready* task.

- A task $T_i$ of $S^0$ does not complete before its deadline hence it is a *ready* task during all its executable period. This implies that there is no idle time during the executable period of $T_i$.
- Similarly, a task of $F$ is a ready task from its release time to the point at which it is abandoned. Therefore there is no idle time between its release point and its abandonment point.

□

**Lemma 5.3** *For any interval $I = (t_{begin}, t_{close})$, the length of $I$, $t_{close} - t_{begin}$ will satisfy*

$$t_{close} - t_{begin} \leq \textsf{ordervalue}(I) + (1 + \frac{1}{\sqrt{k}}) \cdot \textsf{lstvalue}(I) = \textsf{achievedvalue}(I) + \frac{1}{\sqrt{k}} \cdot \textsf{lstvalue}(I)$$

*Recall that* ordervalue$(I)$ *and* lstvalue$(I)$ *are the values obtained from the order-scheduled and the lst-scheduled tasks respectively during $I$.*

PROOF.

An interval $I = [t_{begin}, t_{close}]$, has the following two sub-portions the second of which may be empty.

1. $[t_{begin}, t_{first\_lst}]$

   From the beginning of $I$ to the point in time, $t_{first\_lst}$, in which the first lst-scheduled task is scheduled. During this period all tasks are order-scheduled and some may complete their execution.

   If no task is lst-scheduled in $I$ then define $t_{first\_lst}$ to be $t_{close}$. In this case the second sub-portion is empty.

2. $[t_{first\_lst}, t_{close}]$

   During this period, all tasks are scheduled and preempted by **Latest-start-time Interrupt**. Only the last task to be scheduled completes.

If there are no lst-scheduled tasks in $I$ then all tasks that executed from $t_{begin}$ to $t_{close}$ completed successfully. The value achieved is ordervalue$(I)$ and is at least as big as the duration of execution [4]. Hence, the lemma is proved in this case

Otherwise, suppose that $T_1, T_2, \cdots, T_m$ $(m \geq 1)$ are the tasks that were lst-scheduled in $I$. Hence, $T_1$ was scheduled at $t_{first\_lst}$, later it was preempted (and abandoned) by $T_2$ and so forth. Eventually $T_m$ preempts $T_{m-1}$ and completes at $t_{close}$, its value $v_n$ is lstvalue$(I)$.

Denote by $l_i$ the length of the execution of $T_i$ during the process above.

$T_m$ preempted $T_{m-1}$ hence $v_m > (1 + \sqrt{k})v_{m-1}$. Which yields [5]

$$l_{m-1} < v_{m-1} < \frac{v_m}{(1+\sqrt{k})} = \frac{\textsf{lstvalue}(I)}{(1+\sqrt{k})}$$

Going backward along the chain of preemptions we get:

$$l_i < v_i < \frac{v_{i-1}}{(1+\sqrt{k})} < \frac{\textsf{lstvalue}(I)}{(1+\sqrt{k})^{m-i}} \quad \textit{for all } 1 \leq i \leq m-1 \tag{1}$$

[4] Recall that a value density is always equal or greater than 1, see assumption 3.2 above.

[5] Note that always $l_i \leq v_i$. However, for a task that was abandoned a strict inequality $l_i < v_i$. holds.

$T_1$ preempted the last order-scheduled task hence (see statement 72 of $D^{over}$)

$$v_1 > (1+\sqrt{k})\{\textsf{recentval}(t_{first\_lst}) + value(current\ task\ at\ time\ t_{first\_lst})\} \quad (2)$$

Also,

$$t_{first\_lst} - t_{begin} \leq \textsf{ordervalue}(I) + \textsf{recentval}(t_{first\_lst}) + value(current\ task\ at\ time\ t_{first\_lst}) \quad (3)$$

This holds because the processor is not idle between $t_{begin}$ and $t_{first\_lst}$ (as part of $BUSY$) and the right hand side above represents the sum of the values of all the tasks that were scheduled between $t_{begin}$ and $t_{first\_lst}$. This sum must be greater than or equal to their period of execution by the normalized importance assumption (assumption 3.2). Inequalities 1, 2 and 3 imply

$$t_{first\_lst} - t_{begin} < \textsf{ordervalue}(I) + \frac{v_1}{(1+\sqrt{k})} < \textsf{ordervalue}(I) + \frac{\textsf{lstvalue}(I)}{(1+\sqrt{k})^m}$$

We have produced the following bound on the distance between $t_{begin}$ and $t_{close}$:

$$\begin{aligned}
t_{close} - t_{begin} &= (t_{first_lst} - t_{begin}) + (t_{close} - t_{first_lst}) \\
&= (t_{first_lst} - t_{begin}) + (l_1 + l_2 + \cdots + l_m) \\
&< \textsf{ordervalue}(I) + \textsf{lstvalue}(I) \cdot \left(1 + \frac{1}{(1+\sqrt{k})} + \frac{1}{(1+\sqrt{k})^2} + \cdots + \frac{1}{(1+\sqrt{k})^m}\right) \\
&< \textsf{ordervalue}(I) + \textsf{lstvalue}(I) \cdot \sum_{i=0}^{\infty} \frac{1}{(1+\sqrt{k})^i} \\
&= \textsf{ordervalue}(I) + \textsf{lstvalue}(I) \cdot (1 + \frac{1}{\sqrt{k}}) \\
&= \textsf{achievedvalue}(I) + \frac{1}{\sqrt{k}} \cdot \textsf{lstvalue}(I)
\end{aligned}$$

The last equality follows from the fact that achievedvalue(I) = ordervalue(I) + lstvalue(I) by definition. □

**Lemma 5.4** *Suppose $T_i$ was abandoned during the interval $I$. Then*

$$v_i \leq (1+\sqrt{k}) \cdot \textsf{achievedvalue}(I)$$

*Recall that* achievedvalue$(I)$ *is the total value obtained during $I$.*

PROOF.

Let $I = (t_{begin}, t_{close})$ be an interval. Define the *Prospective Value* map of $I$, $PV_I$, as follows:

$$PV_I(t) = \textsf{ordervalue}(t) + \textsf{recentval}(t) + value(current\ tasks\ at\ time\ t)$$
$$where\ t_{begin} \leq t \leq t_{close}$$

**Claim** For every interval, $I = [t_{begin}, t_{close}]$,

1. $PV_I$ is monotone non-decreasing.

2. $PV_I$ reaches, at the end of the interval, the total value obtained in $I$, i.e,

$$PV_I(t_{close}) = \textsf{achievedvalue}(I)$$

Note: $PV$ is not a function because it might have several values for one time instance since $D^{over}$ can make several scheduling decisions at one time instance (see assumption 3.2 ). However, as a map with the ordered sequence of scheduling decisions as its domain, $PV_I$ is a function.
**proof of claim.**

There are two cases.
The first applies when there is no lst-scheduled task in $I$, the other applies when such tasks exist.
Case 1: Suppose that there are no lst-scheduled tasks in $I$. Then every task that was scheduled does complete. Let $S(t)$ be the set of tasks that were scheduled (not necessary completed) up to $t$. One can verify by induction that

$$PV_I(t) = \sum_{T_i \in S(t)} v_i$$

The reason is that no scheduled task is abandoned hence at each moment a task is either the current task or in $\textsf{Qrecent}$ or had completed. At the closing of $I$ all tasks have completed. Hence,

$$PV_I(t_{close}) = \sum_{T_i \in S(t_{close})} v_i = \textsf{achievedvalue}(I)$$

$PV_I$ is monotone (when there are no lst-scheduled tasks) because $S(t)$ is a monotone increasing set of tasks.
Case 2: Suppose there were lst-scheduled tasks. Assume that the first lst-scheduled task, $T_1$, was scheduled at time $t_{first\_lst}$ . Let $t$ be a time instance just before the scheduling of $T_1$, then by definition:

$$PV_I(t) = \textsf{ordervalue}(t) + \textsf{recentval}(t) + value(current\ tasks\ at\ time\ t)$$

$T_1$ is scheduled only if

$$v_1 > (1+\sqrt{k}) \cdot (\textsf{recentval}(t) + value(current\ tasks\ at\ time\ t))$$

When $T_1$ is scheduled $\textsf{recentval}$ is set to zero hence we can conclude that

$$\begin{aligned}
PV_I(t_{first_lst}) &= \textsf{ordervalue}(t_{first_lst}) + \textsf{recentval}(t_{first_lst}) + value(T_1) \\
&= \textsf{ordervalue}(t) + 0 + value(T_1) \\
&> \textsf{ordervalue}(t) + (\textsf{recentval}(t) + value(current\ tasks\ at\ time\ t)) \\
&= PV_I(t)
\end{aligned}$$

Thus, $PV_I$ is monotone from $t_{begin}$ to $t_{first\_lst}$ (as in the case when there are no lst-scheduled tasks). It is left to show that $PV_I$ continues to be monotone. After $t_{first\_lst}$, $PV_I$ equals to

$$\textsf{ordervalue}(I) + value(current\ tasks\ at\ time\ t)$$

because recentval remains equal to zero. This is a monotone increasing value since ordervalue($I$) is fixed and a task $T$ will preempt the current task only if it has a larger value than the current task's value. In particular if $T_i$ is the last task to be scheduled in $I$ then

$$\begin{aligned} PV_I &= \mathsf{ordervalue}(I) + v_i \\ &= \mathsf{ordervalue}(I) + \mathsf{lstvalue}(I) = \mathsf{achievedvalue}(I) \end{aligned}$$

So, the claim is proved.

**end of proof of claim**

There is only one way a task, $T_i$, can be abandoned at time $t$:

- $T_i$ reaches its $LST$ at $t$. A Latest-start-time Interrupt is generated. However, $T_i$ has insufficient value to preempt the task executing at time $t$.

Hence if $T_i$ was abandoned then

$$\begin{aligned} v_i &< (1+\sqrt{k}) \cdot \{\mathsf{recentval}(t) + value(current\ \ task\ at\ time\ t)\} \\ &\leq (1+\sqrt{k}) \cdot PV_I(t) \qquad \text{, by definition of } PV \\ &\leq (1+\sqrt{k}) \cdot \mathsf{achievedvalue}(I) \qquad \text{, by the claim} \end{aligned}$$

□

**Lemma 5.5** *Suppose $T_i$ was abandoned at time $t$ in $I = [t_{begin}, t_{close}]$. Then,*

$$c_i \geq d_i - t_{close}$$

PROOF.

A task $T_i$, can be abandoned at time $t$ only when:

- It reaches its $LST$ at $t$. A Latest-start-time Interrupt is generated. However, the current task is not preempted.

$T_i$ reached its $LST$ hence its computation time is at least $d_i - t$. Also, $t \leq t_{close}$ by assumption. Hence the computation time of $T_i$ is at least $d_i - t_{close}$.

□

## 5.2 How Well Can a Clairvoyant Scheduler Do?

Given a collection of tasks $\Gamma$, our goal is to bound the maximum value that a clairvoyant algorithm can obtain from scheduling $\Gamma$. We do it by observing the way $D^{over}$ schedules $\Gamma$. From $D^{over}$'s scheduling we get the partitioning of the tasks to $S^0, S^p$ and $F$ we also take notice of the time periods in which the processor was not idle in this scheduling. The union of these periods is called $BUSY$.

**Definition 5.3** *BUSY*

Suppose $D^{over}$ schedules $\Gamma$. Let $BUSY$ denotes the time during which the processor is not idle during the execution of $\Gamma$. For simplicity, the length of $BUSY$ will also be denoted by $BUSY$.

Note that $BUSY$ equals the union of all intervals created when $D^{over}$ schedules $\Gamma$.

□

In order to bound the value that can be achieved from scheduling $\Gamma$, we will offer the clairvoyant algorithm two gifts that can only improve the value it can obtain. We will show an upper bound on the value the clairvoyant algorithm can get with these gifts hence bounding the value it can achieve from the original collection.

- As a first gift, we will give the clairvoyant algorithm the sum of the values of all tasks in $S^p$ at no cost to it (i.e. it will devote no time to these tasks). Then we will see what the clairvoyant algorithm can achieve on $F \cup S^0$.

- As a second gift, suppose that in addition to the value achieved from scheduling the tasks $F \cup S^0$ the clairvoyant scheduler can get an additional value called *granted value.* The amount of granted value depends on the scheduling chosen by the the clairvoyant scheduler: A value density of $k$ will be granted for every period of $BUSY$ that is not used for executing a task.

  The clairvoyant scheduler must consider that scheduling a task might reduce the granted value (since time in $BUSY$ is used). Of course, when this reduction is bigger than the value of a task then the task should not be scheduled. Suppose the clairvoyant algorithm had chosen a scheduling for $F \cup S^0$. We can assume that no task was scheduled entirely during $BUSY$ because the granted value lost would be greater or equal to the value gained from scheduling the task.

  We will show that tasks of $S^0$ can execute only during $BUSY$ hence this leaves only tasks of $F$ that were scheduled partially[6] outside $BUSY$. Executing $T$ results in a gain of $value(T)$, but entails a loss of the granted value for the time that $T$ executed in $BUSY$.

  The clairvoyant scheduler has now two options. It can schedule no task during the entire $BUSY$ period and get only (the whole) granted value or it can use some of $BUSY$ in order to schedule some of $F$ tasks. We will show that the maximal possible gain from choosing the second option is bounded by $(1 + \sqrt{k}) \cdot \textsf{achievedvalue}$. Putting this altogether will give the desired result (theorem 5.12).

**Example 5.4**

To see the possibilities opened to the clairvoyant algorithm by introducing the granted value consider the following example:

The length of $BUSY$ is 5 and the importance ratio, $k$, is 4. $F$ contains only one task, $T$, with computation length 3 and value density 2.

[6] When the computation time of a task is known precisely when it is released, a task $T \in F$ can not be scheduled completely outside $BUSY$ (see lemma 5.2). However, if the computation time of a task is not exactly known (section 8), then a failed task $T$ may be scheduled completely outside $BUSY$.

Without scheduling T the value obtained by the clairvoyant algorithm only from the granted value is $5 \times 4 = 20$. If $T$ could have been scheduled without using any of $BUSY$ time then its value will be *added* to give $20 + 2 \times 3 = 26$. However if the clairvoyant algorithm must use 2 units of $BUSY$'s time in order to schedule $T$ then the total value will be only $(5-2) \times 4 + 6 = 18$, hence it is better not to schedule $T$ in this case. As a matter of fact, whenever $T$ has to use more than $\frac{3}{2}$ units of $BUSY$'s time it should not be scheduled.
$\square$

Suppose a *clairvoyant scheduler* has to schedule a collection of tasks $A$. We can assume that it schedules a task only if that task eventually completes. Hence the work of a clairvoyant scheduler is first to choose the set of tasks $A' \subseteq A$ that will be scheduled and then to work out the details of the processor allocation among the tasks of $A'$. We will call all possible selections of $A'$ and processor allocation *a scheduling* of $A$.

In the scenario above the clairvoyant scheduler can achieve the maximal value of the sum in equation 4 below ranging over all possible schedulings of $F$.

$$\begin{array}{c}\textit{value obtained from those}\\ \textit{tasks of } F \textit{ that were scheduled}\end{array} + k \cdot \begin{array}{c}\textit{length of time in } BUSY \textit{ not utilized to}\\ \textit{schedule the tasks of } F\end{array} \qquad (4)$$

Denote by $C(\cdot)$ the value that a clairvoyant algorithm can achieve from a collection of tasks.

We would like to show that $C(F \cup S^0)$ can not be greater then this maximal value. This will then give us an upper bound on what a clairvoyant algorithm can achieve.

**Lemma 5.6**

$$C(F \cup S^0) \quad \leq \quad \max_{\substack{\textit{possible}\\ \textit{scheduling}\\ \textit{of } F}} \left\{ \begin{array}{c}\textit{value obtained by}\\ \textit{scheduling tasks of}\\ F\end{array} + k \cdot \begin{array}{c}\textit{length of time in } BUSY \textit{ not}\\ \textit{utilized by tasks of } F\end{array} \right\}$$

PROOF.

$$C(F \cup S^0) \leq \max \left\{ \begin{array}{c}\textit{value obtained from}\\ \textit{scheduling tasks of } F\end{array} + \begin{array}{c}\textit{value obtained from scheduling}\\ \textit{tasks of } S^0 \textit{ during the time not}\\ \textit{used by tasks of } F\end{array} \right\}$$

$S^0$ tasks can be scheduled only during $BUSY$ (lemma 5.2 ) hence,

$$\begin{array}{ccc} \textit{value obtained from} & + & \textit{value obtained from scheduling tasks of } S^0 \\ \textit{scheduling tasks of } F & & \textit{during the time not used by tasks of } F \end{array}$$

$$\leq \begin{array}{ccc} \textit{value obtained by} & + k \cdot & \textit{length of time in } BUSY \textit{ not utilized} \\ \textit{scheduling } F & & \textit{by tasks of } F \end{array}$$

The lemma is proved.

□

With the above gifts, the clairvoyant scheduler can maximize the sum in 4 above and hence obtain a value of at least $C(F \cup S^0)$ .

Suppose a task $T_f \in F$ is scheduled to completion. If $T_f$ executes entirely during $BUSY$ then the left hand factor of the sum is increased only by $v_i$ which is smaller than or equal to $k \cdot c_i$ while the right hand factor is decreased by $k \cdot c_i$ giving zero or negative net change. Thus we assume that $T_f$ executes (at least partially) outside $BUSY$.

**Lemma 5.7** *Suppose $T_f$ was abandoned (by $D^{over}$) at time $t_{aban}$ and that $I = [t_{begin}, t_{close}]$ is the the interval in which $T_f$ is abandoned. Then, if $T_f$ is to be executed (by the clairvoyant algorithm) anywhere outside $BUSY$ it must be after $t_{close}$.*

PROOF.

$\Delta_f = [r_f, t_{aban}] \cup [t_{aban}, d_f]$. The first portion of $\Delta_f$ is contained in $BUSY$ (lemma 5.2). $[t_{aban}, t_{close}] \subseteq I \subseteq BUSY$, hence if $T_i$ is to be scheduled anywhere outside $BUSY$ it must be after $t_{close}$ [7].

□

Now we are ready to give an upper bound on how much additional value can the clairvoyant algorithm achieve by scheduling tasks of $F$ compared with collecting only the granted value without scheduling any tasks. We make strong use of the fact that when a task $T$ is abandoned during $I$, $T$'s value can not be too large with respect to achievedvalue($I$). We believe the techniques in this lemma to be widely useful.

**Lemma 5.8** *With the above gifts, the total net gain obtained by the clairvoyant algorithm from scheduling the tasks abandoned during $I$ is not greater than*

$$(1 + \sqrt{k}) \cdot \mathsf{achievedvalue}(I)$$

PROOF.

[7] Note that parts of $[t_{close}, d_f]$ might be included in $BUSY$ as a new interval may be opened before $d_f$

Assume that a clairvoyant scheduler selected a scheduling for the tasks of $F$ considering the value that can be gained from leaving $BUSY$ periods idle. We can assume that a clairvoyant algorithm executes a task only if this task eventually completes. If the clairvoyant algorithm does not schedule any of the tasks abandoned during $I$ the lemma is proved. Hence, assume that of all the tasks abandoned in $I = [t_{begin}, t_{close}]$, the clairvoyant scheduler schedules $T_1, T_2, \cdots T_m$ (in order of completion). These tasks execute for $l_1, l_2, \cdots l_m$ time *after* $t_{close}$ (hence, maybe outside $BUSY$). We know that all the $l_i$'s are greater than zero (otherwise there is no net gain).

Lemma 5.4 ensures that the biggest possible value of a task to be abandoned during $I$ is $(1+\sqrt{k})\cdot\mathsf{achievedvalue}(I)$. If such a task has value density $k$ its execution time is $\frac{(1+\sqrt{k})\cdot\mathsf{achievedvalue}(I)}{k}$. Denote by $L$ the maximal value of this execution time and the length of $l_1$

$$L = \max\{\frac{(1+\sqrt{k})\cdot\mathsf{achievedvalue}(I)}{k}, l_1\} \tag{5}$$

Let $j$ be the index less than or equal to $m$ such that

$$\sum_{i\leq j} l_i \leq L < \sum_{i\leq j} l_i + l_{j+1}$$

If no such $j$ exists define $j$ to be $m$.

First, assume that we have an equality, $\sum_{i\leq j} l_i = L$. The $\sum_{i\leq j} l_i < L$ case is a little more complicated and will be treated later.

We will show that the net gain from scheduling tasks within a period of $L$ after the end of the interval can not be greater than $(1+\sqrt{k})\cdot\mathsf{achievedvalue}(I)$.

- Suppose that in 5, the maximum is the first term. Then the total net gain from $T_1, T_2, \cdots T_j$ is not greater than

$$k\cdot\sum_{i\leq j} l_i = k\cdot L = (1+\sqrt{k})\cdot\mathsf{achievedvalue}(I) \tag{6}$$

- If on the other hand the second term is maximal in 5 then the value obtained by scheduling $T_1$ is at most $(1+\sqrt{k})\cdot\mathsf{achievedvalue}(I)$ (lemma 5.4).

Now we will show that the net gain from scheduling tasks "after" $L$ is never positive.

Every task $T_i$ that executed at a time of at least $L$ after the end of the interval, where $j < i \leq m$, has an execution time of at least $d_i - t_{close}$ (see lemma 5.5).

$$\begin{aligned} d_i - t_{close} &\geq \text{"the point at which } T_i \text{ completes (according to the clairvoyant)"} - t_{close} \\ &\geq (t_{close} + \sum_{g\leq i} l_g) - t_{close} \\ &\geq l_i + \sum_{g\leq j} l_g = l_i + L \end{aligned}$$

For $i > j$, $T_i$ was scheduled by the clairvoyant scheduler but used only $l_i$ time after $t_{close}$. Hence, $T_i$ executed at least $L$ time before $t_{close}$ that is to say in $BUSY$ by lemma 5.7. The "loss" from scheduling $T_i$ during $BUSY$ is at least $k \cdot L$. The value obtained by scheduling $T_i$ is at most $(1+\sqrt{k}) \cdot \textsf{achievedvalue}(I)$ (lemma 5.4). Hence the net gain is less than or equal to

$$
\begin{aligned}
& (1+\sqrt{k}) \cdot \textsf{achievedvalue}(I) - k \cdot L \\
\leq\ & (1+\sqrt{k}) \cdot \textsf{achievedvalue}(I) - (1+\sqrt{k}) \cdot \textsf{achievedvalue}(I) \\
=\ & 0
\end{aligned}
$$

We conclude that the clairvoyant algorithm is better off not scheduling any task $T_i$, $\quad j < i \leq m$. Hence, the lemma is proved for the case that $\sum_{i \leq j} l_i = L$.

What if $L$ does not equal any of the partial sums? That is, if $\sum_{i \leq j} l_i < L < \sum_{i \leq j+1} l_i$. We will augment the total value given to the clairvoyant by some non-negative amount. Then we will show that even with this addition the net gain achieved by the clairvoyant algorithm is bounded by $(1+\sqrt{k}) \cdot \textsf{achievedvalue}(I)$, hence proving the lemma.

First we will take the value density of $T_j$ to be $k$. This move can only increase the overall value achieved by the clairvoyant algorithm. We will also "transfer" some execution time (and hence also value) from $T_{j+1}$ to $T_j$. We will transfer exactly $L - \sum_{i \leq j} l_i$ execution time. There will be a non-negative net increase of $(k - imp(T_{j+1})) \cdot (L - \sum_{i \leq j} l_i)$ in the overall achieved value of the clairvoyant algorithm and we are back in the case of $L = \sum_{i \leq j} l_i$. The total net gain from $T_1, \cdots, T_{j+1}$ is bounded by $(1+\sqrt{k}) \cdot \textsf{achievedvalue}(I)$ while the net gain from all other tasks is zero or negative.

□

Our strategy thus far has entailed partitioning the problem into what the clairvoyant can obtain with respect to a given interval compared to what $D^{over}$ obtains in that interval. We now compute an upper bound for what the clairvoyant algorithm can obtain over all intervals. This may overestimate what the clairvoyant algorithm obtains, because the time periods that the clairvoyant algorithm uses on the tasks of two neighboring intervals may overlap.

**Corollary 5.9** *With the above gifts, the total net gain (over the entire execution) obtained by the clairvoyant algorithm from scheduling the tasks of $F$ is not greater than*

$$(1+\sqrt{k}) \cdot \textsf{achievedvalue}$$

PROOF.

Lemma 5.8 measured the maximum net gain per interval. By construction, each task is accounted for in exactly one interval. Therefore, summing over all intervals we conclude that the total net gain during the entire execution is less than or equals to $(1+\sqrt{k}) \cdot \textsf{achievedvalue}$.

□

The previous corollary bounds the value the clairvoyant algorithm could obtain beyond the granted value. Now, we will estimate the granted value (by bounding the length of $BUSY$) to get an upper bound on $C(S^0 \cup F)$.

**Lemma 5.10**

$$\begin{aligned} C(S^0 \cup F) &\leq k \cdot (\text{achievedvalue} + \frac{1}{\sqrt{k}} \cdot \text{zerolaxval}) + (1+\sqrt{k}) \cdot \text{achievedvalue} \\ &= (k+1+\sqrt{k}) \cdot \text{achievedvalue} + \sqrt{k} \cdot \text{zerolaxval} \end{aligned}$$

PROOF.

Lemma 5.6 shows that $C(F \cup S^0)$ is bounded by the maximum, ranging over all possible schedulings of the tasks of $F$, of the following sum:

$(value\ obtained\ by\ scheduling\ F) + k \cdot (length\ of\ time\ in\ BUSY\ not\ utilized\ by\ F\ tasks)$.

Corollary 5.9 above, shows that this sum is less than or equal to

$$(1+\sqrt{k}) \cdot \text{achievedvalue} + k \cdot BUSY$$

Lemma 5.3, summed over all intervals yields:

$$BUSY \leq \text{achievedvalue} + \frac{1}{\sqrt{k}} \cdot \text{lstvalue}$$

$\text{lstvalue}(I) \leq \text{zerolaxval}(I)$ always holds because every task that is lst-scheduled must have completed at its deadline. This implies that

$$BUSY \leq \text{achievedvalue} + \frac{1}{\sqrt{k}} \cdot \text{zerolaxval}$$

Hence,

$$\begin{aligned} C(S^0 \cup F) &\leq k \cdot (\text{achievedvalue} + \frac{1}{\sqrt{k}} \cdot \text{zerolaxval}) + (1+\sqrt{k}) \cdot \text{achievedvalue} \\ &= (k+1+\sqrt{k}) \cdot \text{achievedvalue} + \sqrt{k} \cdot \text{zerolaxval} \end{aligned}$$

Which proves the lemma.

□

We gave the clairvoyant algorithm the value of all tasks in $S^p$. We also got a bound on $C(S^0 \cup F)$ . The following lemma shows that the sum of these two values bounds the value the clairvoyant can get from the entire collection.

**Lemma 5.11**

$$C(F \cup S^0 \cup S^p) \leq C(F \cup S^0) + C(S^p) \leq C(F \cup S^0) + \sum_{T_i \in S^p} v_i$$

PROOF.

$C(\cdot)$ is a sublinear function For every two collections of tasks $A$ and $B$ the value that a clairvoyant algorithm can get from scheduling $A \cup B$ is not greater than the sum of values from scheduling each collection separately. The reason is that executing tasks of $A$ might interfere with tasks of $B$ and vise versa.

$$C(A \cup B) \leq C(A) + C(B)$$

Hence,

$$C(F \cup S^0 \cup S^p) \leq C(F \cup S^0) + C(S^p)$$

$C(S^p)$ can not be greater than the sum of the values of all the tasks in $S^p$. That yields the desired result.

□

Given a collection of tasks $\Gamma$, lemmas 5.10 and 5.11 give an upper bound on the value the clairvoyant algorithm can obtain from $\Gamma$ in terms of the value obtained by $\mathrm{D}^{over}$ (**achievedvalue**, **zerolaxval** and **poslaxval**). The next theorem puts these results together.

**Theorem 5.12** *$D^{over}$ has a competitive factor of $\frac{1}{(1+\sqrt{k})^2}$. That is, $D^{over}$ obtains at least $\frac{1}{(1+\sqrt{k})^2}$ times the value of a clairvoyant algorithm given any task collection $\Gamma$.*
PROOF.

In the notation of the lemmas above, we got from lemma 5.10 that

$$C(S^0 \cup F) \leq (k + 1 + \sqrt{k}) \cdot \mathsf{achievedvalue} + \sqrt{k} \cdot \mathsf{zerolaxval}$$

We will bound $\sqrt{k} \cdot \mathsf{zerolaxval}$ in the above equation.

$$\begin{aligned} \sqrt{k} \cdot \mathsf{achievedvalue} &= \sqrt{k} \cdot \mathsf{zerolaxval} + \sqrt{k} \cdot \mathsf{poslaxval} \\ &\geq \sqrt{k} \cdot \mathsf{zerolaxval} + \mathsf{poslaxval} \\ \Rightarrow \sqrt{k} \cdot \mathsf{zerolaxval} &\leq \sqrt{k} \cdot \mathsf{achievedvalue} - \mathsf{poslaxval} \end{aligned}$$

Hence, replacing $(\sqrt{k} \cdot \mathsf{zerolaxval})$ by $(\sqrt{k} \cdot \mathsf{achievedvalue} - \mathsf{poslaxval})$ yields:

$$\begin{aligned} C(S^0 \cup F) &\leq (k + 1 + \sqrt{k}) \cdot \mathsf{achievedvalue} + \sqrt{k} \cdot \mathsf{achievedvalue} - \mathsf{poslaxval} \\ &= (1 + \sqrt{k})^2 \cdot \mathsf{achievedvalue} - \mathsf{poslaxval} \end{aligned}$$

Using lemma 5.11 we get:

$$\begin{aligned} C(F \cup S^0 \cup S^p) &\leq C(F \cup S^0) + C(S^p) \\ &= C(F \cup S^0) + \mathsf{poslaxval} \\ &\leq ((1 + \sqrt{k})^2 \cdot \mathsf{achievedvalue} - \mathsf{poslaxval}) + \mathsf{poslaxval} \\ &= (1 + \sqrt{k})^2 \cdot \mathsf{achievedvalue} \end{aligned}$$

□

### 5.3 The Running Complexity of $D^{over}$

In the previous section we analyzed the performance of $D^{over}$ in the sense of what value it will achieve from scheduling tasks to completion. In this section we study the cost of executing the scheduling algorithm itself.

**Theorem 5.13** *If $n$ bounds the number of unscheduled tasks in the system at any instant then each task incurs an $O(\log n)$* **amortized** *cost.*
PROOF.
$D^{over}$ requires three data structures, called **Qrecent**, **Qother** and **Qlst**, all of them priority queues, implemented as balanced search trees, e.g. 2-3 trees. They support **Insert**, **Delete**, **Min** and **Dequeue** operations, each taking $O(\log n)$ time for a queue with $n$ tasks. The structures share their leaf nodes which represent tasks.

$D^{over}$ consists of a main loop with three "interrupt handlers" within it. The total number of operations is dominated by the number of times each of these handler clauses is executed and the number of data structure operations in each clause.

Suppose a history of $m$ tasks is given.

First, let us estimate the number of times each handler clause can be executed. A task during its lifetime causes exactly one **Task Release** event and at most one **Task Completion** event as well as at most one **Latest-start-time Interrupt** event. Hence, while scheduling $m$ tasks the total number of events is bounded by $3m$.

Now, we will bound the number of queue operations in each handler clause.

- In the handler for the **Task Release** event (statement 46), there is a constant number of queue operations. Hence, this contributes a total of $O(m)$ queue operations during the entire history.

- In the handler for the **Task Completion** event (statement 0) there is a constant number of queue operations. Hence, this contributes a total of $O(m)$ queue operations during the entire history.

- In the handler for **Latest-start-time Interrupt** event (see statement 69), the number of queue operations is proportional to the number of tasks in **Qrecent** plus a constant (because the recently-preempted tasks are all inserted into **Qother**, statement 74). How many tasks can be in **Qrecent** throughout the history? A task can enter **Qrecent** only as a result of **Task Release** event there are at most $m$ such events. Hence, the total number of tasks in **Qrecent** is at most $m$, which means that the total number of queue operations is $O(m)$ during the entire history.

We conclude that the total number of operations for the entire history is $O(m \log n)$ and the theorem is proved.
□

# 6 Conflicting Tasks

What if the collection of tasks to be scheduled is underloaded, that is to say that all tasks can be scheduled to completion? We would like the on-line scheduler to be optimal in this case.

D$^{over}$ is optimal for underloaded systems. In fact, it has an even stronger performance guarantee: We devise a procedure (*Remove_Conflicts*) to partition the tasks into two classes. The *conflict-free* tasks are those that can be scheduled to completion without preventing any other task from completing (in a sense to be made precise in the algorithm below). A task is *conflicting* otherwise.

We will show that D$^{over}$ schedule to completion all conflict-free tasks (in particular all tasks in an underloaded system) and also obtains at least $\frac{1}{(1+\sqrt{k})^2}$ the value a clairvoyant algorithm can get from the conflicting tasks.

The definitions of underloaded and overload systems in section 3 are natural and widely accepted. However, even when a system is overloaded it is possible that some periods are "underloaded" i.e., it is possible that some tasks will be scheduled to completion by all clairvoyant algorithms since they do not prevent any other task from completion. One can define the periods occupied by the aggregated tasks as overloaded *intervals*. We prefer this definition to that of [2, 7] because it does not depend on the behavior of D[8].

```
Function Remove_Conflicts ( Γ ) ;

if num_of_tasks(Γ) == 1 then
    return(Γ);
end {if} ;

collection_num_of_tasks :=2
repeat
    Select a collection of tasks S = T_{i_1}, T_{i_2}, ..., T_{i_{collection_num_of_tasks}} of size
    collection_num_of_tasks such that
    r = Min_{T_i ∈ S}{r_i} and d = Max_{T_i ∈ S}{d_i} and
    c_{i_1} + c_{i_2} + ... + c_{i_{collection_num_of_tasks}} > (d − r)
    if (such a collection is found) then
        mark all the tasks in S as conflicting tasks;
        Create a task T with release time r and deadline d
        and with no slack time;
        (* T is an aggregated task *)
        return( remove_conflicts( Γ − S + {T}));
        (* Starts again with the new collection of tasks with a smaller col-
        lection of tasks. When the recursive calls reach the bottom of the
        recursive the result is propagated upwards (tail recursion). *)
    else
        collection_num_of_tasks := collection_num_of_tasks + 1;
    end {if} ;
until collection_num_of_tasks > num_of_tasks(Γ)
```

THE REMOVE CONFLICTS ALGORITHM.[9]

[8] Also, in [2, 7] a task is "overloaded" if and only if its deadline is in an overloaded interval. This is not reasonable because even tasks that have enough slack time to complete "safely" before the overloaded interval starts will be considered as "overloaded".

[9] Another version of this algorithm is an iterative algorithm that at each iteration selects non-deterministicly

**Example 6.1** To see how **remove_conflicts** works consider the following example. Suppose we are given the following collection of tasks:

| Task | *Release-Time* | *Computation-Time* | *Deadline* |
|---|---|---|---|
| $T_1$ | 0 | 4 | 6 |
| $T_2$ | 2 | 4 | 6 |
| $T_3$ | 0 | 2 | 8 |
| $T_4$ | 6 | 2 | 8 |
| $T_5$ | 0 | 1 | 9 |

In the beginning **remove_conflicts** is invoked with the above collection. The algorithm seeks a conflicting collection $S$, starting with collections of size two. $S = \{T_1, T_2\}$ is such a collection since the computation time of these tasks (combined) is 8 but their combined execution periods has only a length of 6. Hence, these tasks are conflicting tasks and an aggregated task $T_a$ is created with release time 0, computation time 6 and deadline 6.

The aggregated task replaces $T_1$ and $T_2$ and **remove_conflicts** is invoked with the new collection. This time there is no conflicting collection of size 2 but there is one of size 3, namely $\{T_a, T_3, T_4\}$. This is true since the combined computation time is 10 while the length of the combined execution periods is only 8. These tasks are replaced by a new aggregated task $T_b$ which is created with release time 0 and computation time 8.

The new aggregated task replaces $T_a$, $T_3$ and $T_4$. **remove_conflicts** is invoked again but this time there are no conflicts. The process terminates. The following table summarizes the results:

| Task | *Release-Time* | *Computation-Time* | *Deadline* | *Final Status* |
|---|---|---|---|---|
| $T_1$ | | | | conflicted |
| $T_2$ | | | | conflicted |
| $T_3$ | | | | conflicted |
| $T_a$ | 0 | 6 | 6 | aggregated task |
| $T_4$ | | | | conflicted |
| $T_b$ | 0 | 8 | 8 | aggregated task |
| $T_5$ | | | | conflict-free |

**Definition 6.2**

- CONFLICTING AND CONFLICT-FREE TASKS: We are given a set $\Gamma$ of *original* tasks. A task $T$ is said to be *conflicting* if it was "marked" as such by the initial or any recursive

a minimal set of conflicting tasks and replace them by an aggregated task. A collection is minimal in the sense that removing any one task will make the remaining tasks schedulable. Our algorithm always selects a minimal collection with the smallest possible number of tasks. Note that the purpose of this algorithm is to define conflicting and conflict_free tasks. No scheduler needs ever to execute it.

call of **remove_conflicts** (statement 9). Conflicting tasks are merged into *aggregated* tasks. A task (original or aggregated) that is not conflicting is said to be a *conflict-free* [10] .

When all the tasks (original or aggregated) of a collection are conflict-free the collection is *conflict-free* and otherwise *conflicting.*

## 6.1 The Performance Guarantee of $D^{over}$

In the following assume that a collection $\Gamma$ is given.

**Lemma 6.1** $\Gamma$ *is overloaded if and only if it is conflicting.*
PROOF.

Assume $\Gamma$ is conflicting we will show that $\Gamma$ is overloaded. Let $T$ be the *first* aggregated task to be created by **remove_conflicts** when invoked with $\Gamma$ as its input. $T$ is an aggregate of original tasks. This means that the sum of the computation times needed for these tasks is greater than the time between their earliest release and latest deadline. Hence these tasks can not be all scheduled (see line 7 of **remove_conflicts**). We conclude that $\Gamma$ is overloaded.

Assume $\Gamma$ is overloaded we will show that $\Gamma$ is conflicting. Let $\tau$ be a minimal set of tasks in $\Gamma$ that can not be scheduled. $\tau$ is minimal in the sense that removing any one task will make the rest of the tasks in $\tau$ schedulable [11]. Let $r$ be the earliest release time and $d$ the latest deadline among all tasks in $\tau$. Let $\tau$ be scheduled by D.

**Claim**

When D schedules $\tau$, there is no idle time between $r$ and $d$.

**proof of claim.**

Suppose the system is idle time at time $t$, then at that time there is no ready task. This means that $\tau$ can be partitioned into two non-empty sets (one with all tasks with deadline before $t$ and the other with deadline after $t$). At least one of these sets can not be scheduled [12]. contradicting the minimality of $\tau$.

**end of proof of claim**

Since the claim shows that there is no idle time, and that D could not schedule all the tasks even while executing continuously, we conclude that the sum of computation times needed for the tasks of $\tau$ is greater than the time that can be possibly allotted to them.

**Remove_conflicts** must have found a conflict in $\Gamma$. To see this notice that as long as no conflict is found, the counter collection_num_of_tasks is advanced and is bound to reach the value of num_of_tasks($\tau$). At that point all the tasks of $\tau$ are still present (i.e. were not merged into an aggregated task) and satisfy the condition of statement 7.

Hence, $\Gamma$ is conflicting.

□

[10] $T$ is conflicting if there a collection of tasks $\tau$ (original or aggregated) such that all the tasks of $\tau$ can be scheduled but not all the tasks in $\tau \cup \{T\}$ can be scheduled.

[11] Such a minimal set must exists since the entire set of tasks, $\Gamma$, can not be scheduled but every singleton set of tasks can be scheduleed.

[12] Recall that D is an optimal scheduler for underloaded systems [9, 4].

**Lemma 6.2** *When scheduling* $\Gamma$*, D—the earliest-deadline-first algorithm— will schedule to completion all conflict-free tasks.*
PROOF.

Let $C$ be the time that can be "occupied" by the aggregated tasks, that is the pointwise union of all their executable periods.

$$C \;=\; \cup_{T_i \text{ is an aggregated task}}[r_i, d_i] \qquad (i)$$

where the union is a pointwise union. One can verify (see statement 10 of **remove_conflicts**) that

$$C \;=\; \cup_{T_i \text{ is an (original) conflicting task}}[r_i, d_i] \qquad (ii)$$

**Remove_conflicts**($\Gamma$) contains all the conflict-free tasks. It is conflict-free, otherwise **remove_conflicts** would not have halted. Hence, by lemma 6.1 all the tasks in **remove_conflicts**($\Gamma$) can be scheduled by D. The aggregated tasks of **remove_conflicts**($\Gamma$) can not be scheduled *outside* $C$. Moreover, all of $C$ must be occupied by aggregated tasks since they have no slack time. Hence the conflict-free tasks are scheduled by D using only time that lies outside $C$.

We showed that D schedules all the conflict-free tasks when the collection to be scheduled is **remove_conflicts**($\Gamma$), but does this hold when D schedules the original set of tasks, $\Gamma$? The answer is *yes*. When scheduling $\Gamma$, equation $(ii)$ above shows that all the time outside $C$ is available to the conflict-free tasks hence, by the previous paragraph, all conflict-free tasks complete their execution when $\Gamma$ is scheduled by D.
□

**Corollary 6.3** *D can schedule all the conflicting-free tasks using only time outside* $C$*.*
□

**Lemma 6.4** *Suppose* $T$ *is not the current executing task and is not in* **Qrecent**. *If* $T$ *has an earlier deadline than all the tasks in* **Qrecent** *and the current executing task (if any), then* $T$ *, the current executing task, and all the tasks in* **Qrecent** *can be scheduled by D.*
*if and only if*

$$\textsf{availtime} \geq \textit{remaining computation time}(T)$$

PROOF.

The proof is by induction on the scheduling decisions of $D^{over}$. The induction is done separately on each interval.
□

**Definition 6.3**

REAL $LST$ EVENT : According to $D^{over}$ scheduling, when a **Task Completion** event occurs the next task to be scheduled is the ready task, $T$, with the earliest-deadline. It is possible that the slack-time of $T$ reached zero exactly when a **Task Completion** event occurred, thus creating an $LST$ event for $T$. We will call this $LST$ event a *false* event since $T$ would have been scheduled even without the interrupt. All other $LST$ events will be called *real.*

In all of the following we ignore the false events. Only *real* $LST$ events are considered.

**Lemma 6.5**

1. *Let $C$ be the time that can be occupied by the aggregated tasks,*

$$C = \cup_{T_i \text{ is an aggregated task}}[r_i, d_i]$$

*then, outside $C$, $D^{over}$ schedules according to earliest-deadline-first (D).*

2. *Under $D^{over}$ scheduling a conflict-free task will never generate a (real)* **Latest-start-time Interrupt**.

3. *Let $A$ be an aggregate task in* **remove_conflicts**$(\Gamma)$ *with parameters $(r_a, d_a)$, then $D^{over}$ will complete on or before $r_a$ all conflict-free tasks with deadline on or before $d_a$.*

PROOF.

Recall that the aggregated tasks in **remove_conflicts**$(\Gamma)$ are those tasks that were created "from" conflicting tasks. List all the aggregated tasks according to deadline order

$$T_{a_1}, T_{a_2}, T_{a_3}, \cdots$$

By the construction of these tasks we know that

$$r_{a_1} < d_{a_1} < r_{a_2} < d_{a_2} < r_{a_3} \cdots$$

(Actually, from **remove_conflicts** one can infer only that $d_{a_1} \leq r_{a_2}$ but if it happens that $d_{a_1} = r_{a_2}$ we can, for the purpose of the the following proof merge $T_{a_1}$ and $T_{a_2}$ into one aggregated task with parameters $r_{a_1}$ and $d_{a_2}$)

$D^{over}$ departs from the earliest-deadline-first scheduling policy only when one of the following events occurs:

- The current task is *lst-scheduled* i.e., it was scheduled as the result of a **Latest-start-time Interrupt**.
- At a **Task Release** event or at a **Task Completion** event, the task with the earliest deadline among all ready tasks is not scheduled because **availtime** is too small (see statement 54 and 65 of $D^{over}$).

D$^{over}$ starts to schedule according to earliest-deadline-first. Before $r_{a_1}$ there is no conflict hence by lemma 6.1 there is no overload. This means that neither of the above conditions occurs (lemma 6.4). Hence, before the first aggregated task (up to $r_{a_1}$), D$^{over}$ schedules in the same way as D. Also, from corollary 6.3 we conclude that all conflict-free tasks with deadline on or before $d_{a_1}$ completed on or before $r_{a_1}$.

Between the first and second aggregated task, i.e., between $d_{a_1}$ and $r_{a_2}$ there can not be any ready conflicting tasks because all conflicting tasks have their deadlines before $d_{a_1}$ or release time after $r_{a_2}$. So, during this time only conflict-free tasks are scheduled. Moreover, they will be scheduled according to earliest-deadline-first. We will show this by showing that neither of the two cases above can hold. A conflict-free task would not create a real $LST$ event ( corollary 6.3 [13]). Also, a task with the earliest-deadline will be immediately scheduled. This holds because, if it is delayed, then D encounters an overloaded situation while executing the conflict-free tasks outside $C$. This contradicts corollary 6.3.

We conclude that up to $r_{a_2}$, D$^{over}$ acts like D and all the conflict-free tasks with deadline before $d_{a_2}$ complete before $r_{a_2}$. The induction can proceed through the entire list of aggregated task and the lemma is proved.

□

**Corollary 6.6** *D$^{over}$ will schedule to completion all conflict-free tasks.*

**Lemma 6.7** *Let $A$ be an aggregate task in* remove_conflicts($\Gamma$) *with parameters $(r_a, d_a)$, then during $(r_a, d_a)$ a conflict-free task will be scheduled by D$^{over}$ only if there are no ready conflicting tasks.*

PROOF.

Lemma 6.5 states that a conflict-free task with deadline on or before $d_a$ would complete before $r_a$. So, if any conflict-free task $T$ with release time $r$ and deadline $d$ is to be scheduled during $A$, it must satisfy $d > d_a$.

Suppose at time $t \in (r_a, d_a)$ there is a ready conflicting task $T_i$. Then $d_i < d_a$ must hold, because $T_i$ must be a part of the aggregated task $A$ [14] [15].

Hence, at time $t$ all ready conflicting tasks have deadlines before the deadline of any conflict-free task. A conflict-free task can be scheduled, in these circumstances, only by a Latest-start-time Interrupt. This can not occur because a conflict-free task will not generate a (real) Latest-start-time Interrupt (lemma 6.5)

□

**Theorem 6.8** *D$^{over}$ schedules to completion all conflict-free tasks and obtains at least $\frac{1}{(1+\sqrt{k})^2}$ the value a clairvoyant algorithm gets from all other (i.e., conflicting) tasks.*

[13] As a matter of fact the conflict-free tasks might have even used some of the time of $C$ (when scheduled by D$^{over}$).

[14] We say that, a task $T$ is a part of an aggregated task $A$ if it is one of the tasks that were "merged" to create $A$.

[15] $T_i$ is a conflicting task hence it is a part of an aggregated task, $B$, if this task is not $A$ then the two aggregated tasks should be merged contradicting the fact that $A$ is a task in remove_conflicts($\Gamma$).

PROOF.

The first part of this lemma is merely a repetition of corollary 6.6. From lemma 6.7, we conclude that $D^{over}$ schedules the conflicting tasks regardless the presence of the conflict-free tasks. Suppose the clairvoyant algorithm has to schedule only the conflicted-tasks. It can schedule this tasks only during $C$. But we have just shown that $D^{over}$ schedules the conflicted tasks as if the conflict-free tasks do not exist. Since $D^{over}$ has a competitive factor of $\frac{1}{(1+\sqrt{k})^2}$ it is guaranteed to achieve at least this fraction of what a clairvoyant algorithm can achieve from all conflicting tasks.
$\square$

# 7 Gradual Descent

In the previous sections we assumed *firm* deadlines. That is, a task has zero value if it misses its deadline. We would like to generalize to *soft* deadlines, which means that a task may have some value even after its deadline.

We assume here a soft deadline scheme called *gradual descent* and show that a suitable variant of $D^{over}$ is $\frac{1}{(1+\sqrt{k})^2}$ competitive in this case. $D^{over}$ is also $\frac{1}{(1+\sqrt{k})^2}$ competitive in some possible generalizations of this scheme. We discuss these generalizations at the end of this section.

## 7.1 Exponential Gradual Descent

Consider the following *"exponential"* value assignment for gradual descent. If a task $T_i$ with computation time $c_i$ and value $v_i$ does not complete by its deadline $d_i$ (we call this deadline, the *zero'th* deadline and denote it by $d_i^0$) then a value of $\frac{v_i}{2}$ can be obtained if it completes by $d_i + \frac{c_i}{2}$. This "deadline" is denoted by $d_i^1$. In general a value of $\frac{v_i}{2^y}$ is obtained the task completes by its $y$'th deadline, $d_i^y = d_i + \frac{c_i}{2} + \frac{c_i}{4} + \cdots + \frac{c_i}{2^y}$. We keep the list of deadlines finite by postulating that a task's value density can not go below 1. This means that the index of the last deadline after which the tasks has zero value is $\lfloor log_2(imp(T_i))\rfloor - 1 \; = \lfloor log_2(\frac{v_i}{c_i})\rfloor - 1$.

For notational convenience any task $T_i$ will have *associated descending* tasks denoted by

$$T_i^0, T_i^1, T_i^2, \cdots, T_i^{\lfloor log_2(imp(T_i))\rfloor - 1}$$

where the release times and the computation times of all these tasks are equal to the release time and the computation time of $T_i$. $T_i^y$ has a firm deadline at $d_i^y$ and a value of $\frac{v_i}{2^y}$. Only one of the tasks associated with $T_i$ can possibly complete. That is, if we say that an algorithm executes $T_i^y$, we mean that $T_i$ completes by deadline $d_i^y$, but after deadline $d_i^{y-1}$.

## 7.2 A Variant of $D^{over}$ for Gradual Descent

We modify the **Latest-start-time Interrupt** handler of $D^{over}$ in such a way that when $T_i^0$ is to be abandoned because it reached its *LST* but does not have enough value to be scheduled (see statement 80 of $D^{over}$), $T_i^0$ is indeed removed from all the data structures but in addition a task release for $T_i^1$ is simulated. $T_i^1$'s remaining computation time is set to the remaining computation time of $T_i^0$. In the same way, if $T_i^1$ is to be abandoned then a third task is "released". This process continues as long as the value density does not go below 1.

### 7.3 Analysis of $D^{over}$ in the Gradual Descent Model

The analysis is similar to one in section 5. We will discuss the differences only. Suppose that a collection of tasks $\Gamma$ with importance ratio $k$ is given. and $D^{over}$ schedules this collection. We partition the collection of tasks according to the question of which associated tasks (if any) completed.

- Let $S^p$ denote the set of tasks that completed successfully and that ended some positive time before their <u>zero'th</u> deadline.
- Let $S^0$ denote the set of tasks that completed successfully but ended exactly at their <u>zero'th</u> deadline.
- For $1 \leq y \leq \lfloor log_2 k \rfloor - 1$, let $S^y$ denote the set of tasks that completed successfully after their $(y-1)$'th deadline but not after their $y$'th deadline (i.e., the $y$'th associated task completed).
- Let $FAIL$ denote the set of tasks that never completed.

We will start by modifying the technical lemmas of subsection 5.1 to the new setting.

### 7.4 Lemmas about $D^{over}$'s Scheduling

For notational convenience we define a minus one deadline $d_i^{-1}$ which equals to the zero'th deadline $d_i^0$.

- In this setting lemma 5.2 reads

**Lemma 7.1**

1. *For any task $T_i$ in $S^y$ (with $y \geq 0$). Suppose $T_i^y$ completed at time $t_{complete} \leq d_i^y$, then*

$$[r_i, d_i^{y-1}] \subseteq [r_i, t_{complete}] \subseteq BUSY$$

2. *For any task $T_i$ in $FAIL$. Suppose $T_i$ was abandoned at time $t_{aban}$, then*

$$[r_i, t_{aban}] \subseteq BUSY$$

PROOF.
The proof is similar to that of lemma 5.2.
□

• Lemma 5.3 holds without change. Note that we continue to make the normalized importance assumption, because we never allow the value density to fall below 1.
• Lemma 5.4 holds without change.
• Lemma 5.5 reads:

**Lemma 7.2** *Suppose $T_i^y$ was abandoned at time $t$ in $I = [t_{begin}, t_{close}]$. Then,*

$$c_i \geq d_i^y - t_{close}$$

PROOF.

The proof is the same as the proof of lemma 5.5.
□

### 7.5 How Well Can a Clairvoyant Scheduler Do?

As in subsection 5.2, given a collection of tasks $\Gamma$, our goal is to bound the maximum value that a clairvoyant algorithm can obtain from scheduling $\Gamma$. We observe the scheduling of $\Gamma$ by $D^{over}$ which gives rise to the definition of $S^p$, the $S^y$'s and $FAIL$. As before, $BUSY$ is defined to be the union of the periods in which the processor was not idle (under $D^{over}$'s scheduling).

The clairvoyant algorithm is offered the same two gifts as before. The first is the sum of the values of all tasks in $S^p$ at no cost to it. The second gift is the *granted value.* That is, in addition to the value obtained from scheduling

$$LATE = (S^0 \cup S^1 \cup \cdots S^{\lfloor log_2 k \rfloor - 1} \cup FAIL)$$

a value density of $k$ will be granted for every period of $BUSY$ that is not used for executing a task of $LATE$. By a similar argument to lemma 5.6 we can see that[16]

$$C(LATE) \leq \max_{\substack{\textit{possible}\\ \textit{scheduling}\\ \textit{of LATE}}} \left\{ \begin{array}{c} \textit{value obtained by} \\ \textit{scheduling tasks of} \\ LATE \end{array} + k \cdot \begin{array}{c} \textit{length of time in BUSY not} \\ \textit{utilized by tasks of LATE} \end{array} \right\}$$

In lemma 5.8 we bounded the net gain that the clairvoyant algorithm could get from scheduling tasks of $F$ [17]. This was done by examining each interval separately. If $T \in F$ is scheduled

[16] Recall that $C(FAIL)$ denotes the value that a clairvoyant algorithm can achieve from scheduling (any subset of) $LATE$.

[17] Note that in section 5 the clairvoyant scheduler could not make any net gain from tasks of $S^0$ that completed in $I$ because they can be executed only during $BUSY$. This is not the case here because if $T_i^y$ completed in $I$, the clairvoyant algorithm could choose a different completion point for $T_i^y$ or even to abandon it in favor of another associated task $T_i^z$ with $z \neq y$.

then its value is *accounted* for in the interval in which $T$ was abandoned by $D^{over}$. Here, the method of relating the value of a task $T \in LATE$ to the interval in which it is *accounted* for is more complicated. Suppose the clairvoyant algorithm chose to execute the $z$'th task of $T_i$ to completion. $D^{over}$ could have chosen to complete any of the associated tasks of $T_i$ ($T_i \in S^y$ for some $y$) or none ($T_i \in FAIL$). In the first case we account for $T_i^z$ in the interval in which $D^{over}$ completed $T_i$; in the second case, in the interval during which $T_i^z$ was abandoned.

Assume that a clairvoyant scheduler selected an optimal scheduling for the tasks of $LATE$ considering the value that can be gained from leaving $BUSY$ periods idle. The execution of a task can give a positive net gain only if the task executed (at least partially) outside $BUSY$. The following lemma shows that such execution may take place only after $t_{close}$.

**Lemma 7.3** *Suppose the associated task $T_i^z$ of $T_i \in LATE$ is scheduled to completion by the clairvoyant algorithm. Suppose that $T_i$ is accounted for in $I = [t_{begin}, t_{close}]$. Then, if $T_i$ is to be executed (by the clairvoyant algorithm) anywhere outside $BUSY$ it must be after $t_{close}$.*

PROOF.

There are two cases:

- $D^{over}$ never completed $T_i$ ($T_i \in FAIL$). In this case let t be the time when $D^{over}$ abandoned $T_i^z$.

  $T_i^z$ can be executed only during $\Delta_{T_i^z}$ which is is $[r_i, t] \cup [t, d_i^z]$. The first portion of $\Delta_{T_i^z}$ is contained in $BUSY$ (lemma 7.1). The second portion is contained in $I$. Hence $[r_i, t_{close}] \subseteq BUSY$.

- $D^{over}$ completed $T_i^y$ for some $y$. Let $t$ be the completion time of $T_i^y$.

  A similar argument as above for $\Delta_{T_i^y} = [r_i, t] \cup [t, d_i^y]$ shows that $[r_i, t_{close}] \subseteq BUSY$.

Hence in both cases, if $T_i^z$ is to be executed outside $BUSY$ it must be after $t_{close}$.
□

- Lemma 5.8 has to be replaced by the following,

**Lemma 7.4** *With the above gifts, the total net gain obtained by the clairvoyant algorithm from scheduling the (associated) tasks* accounted for *in $I$ is not greater than*

$$(1 + \sqrt{k}) \cdot \mathsf{achievedvalue}(I)$$

PROOF.

Let $T_1, T_2, \cdots T_m$ be those tasks that are accounted for in $I = [t_{begin}, t_{close}]$ and that the clairvoyant algorithm scheduled after $t_{close}$ (in order of completion). These tasks execute for $l_1, l_2, \cdots l_m$ time *after* $t_{close}$ (hence, maybe outside $BUSY$ by the above lemma).

Denote by $L$ the following value

$$L = \max\{\frac{(1+\sqrt{k})\cdot \mathsf{achievedvalue}(I)}{k}, l_1\} \tag{7}$$

Let $j$ be the index less than or equal to $m$ such that

$$\sum_{i\leq j} l_i \leq L < l_{j+1} + \sum_{i\leq j} l_i$$

If no such $j$ exists define $j$ to be $m$.

First, assume that we have an equality, $\sum_{i\leq j} l_i = L$.

The proof now has two parts.

$\diamondsuit$*Part 1 :*

We will show that the net gain from scheduling tasks within a period of $L$ after the end of the interval can not be greater than $(1+\sqrt{k})\cdot \mathsf{achievedvalue}(I)$.

- Suppose that in 7, the maximum is the first term. Then the total net gain from $T_1, T_2, \cdots T_j$ is not greater than

$$k \cdot \sum_{i\leq j} l_i = k \cdot L = (1+\sqrt{k})\cdot \mathsf{achievedvalue}(I) \tag{8}$$

- Suppose the second term is maximum in 7 and that the $z$'th associated task of $T_1$ was scheduled by the clairvoyant algorithm. If $T_1^z$ was abandoned in $I$ (by $D^{over}$) then lemma 5.4 ensures that its value is bounded by $(1+\sqrt{k})\cdot \mathsf{achievedvalue}(I)$. The other possibility is that $D^{over}$ completed $T_1^y$ in $I$. If $z \geq y$ then $value(T_1^z) \leq value(T_1^y)$ but $value(T_1^y)$ is a component of $\mathsf{achievedvalue}(I)$ so must be less or equal to it. $z < y$ implies that $T_1$ executed to completion before $t_{close}$, since $d_i^z < d_i^y \leq t_{close}$ — a contradiction.

  Hence, in any case, the value obtained by scheduling $T_1$ is at most $(1+\sqrt{k})\cdot \mathsf{achievedvalue}(I)$.

$\diamondsuit$*Part 2 :*

Now we will show that the net gain from scheduling a task $T_i$ $(j < i \leq m)$ $L$ time after the end of $I$ is never positive. Here we have to distinguish between two cases depending on whether $D^{over}$ completed or abandoned $T_i$ in $I$.

- $D^{over}$ completed $T_i$

  Suppose that $D^{over}$ completed $T_i^y$ at $t_{complete} \in I$ and that the clairvoyant algorithm chose to schedule $T_i^z$.

  There are two possible cases:

  - $z < y$:

    Lemma 7.1 shows that

    $$[r_i, d_i^z] \subseteq [r_i, t_{complete}] \subseteq BUSY$$

    This means that $T_i^z$ executes during $BUSY$, a contradiction.

- $z \geq y$:
  The gradual descending scheme ensures that,

$$\begin{aligned} d_i^z &= d_i^0 + \frac{c_i}{2} + \frac{c_i}{4} + \cdots + \frac{c_i}{2^z} \\ &= d_i^0 + (c_i - \frac{c_i}{2^z}) \end{aligned}$$

  From lemma 7.1 we see that

$$d_i^0 \leq d_i^{y-1} \leq t_{complete} \leq t_{close} \in BUSY$$

  Hence we conclude that

$$d_i^z \leq t_{close} + (c_i - \frac{c_i}{2^z})$$

  $T_i^z$ must complete at or before $d_i^z$ implying that the clairvoyant algorithm schedules $T_i^z$ for at least $\frac{c_i}{2^z}$ time before $t_{close}$ hence in $BUSY$. The loss from the execution during $BUSY$ is at least $\frac{c_i}{2^z} \times k$ while the value of $T_i^z$ is at most $\frac{c_i \times k}{2^z}$. Hence the net gain is not positive.

- $T_i \in FAIL$

  Suppose that the $y$'th associated task of $T_i$ was scheduled by the clairvoyant algorithm and that $T_i^y$ was abandonded by D$^{over}$ in $I = [t_{begin}, t_{close}]$. $T_i^y$ has an execution time of at least $d_i^y - t_{close}$ by lemma 7.2.

$$\begin{aligned} d_i^y - t_{close} &\geq \text{"the point at which } T_i^y \text{ completes (according to the clairvoyant)"} - t_{close} \\ &\geq (t_{close} + \sum_{g \leq i} l_g) - t_{close} \\ &\geq l_i + \sum_{g \leq j} l_g = l_i + L \end{aligned}$$

$T_i^y$ was scheduled by the clairvoyant scheduler but used only $l_i$ time after $t_{close}$. Hence, $T_i$ executed at least $L$ time before $t_{close}$ that is to say in $BUSY$ (lemma 7.3). The "loss" from scheduling $T_i$ during $BUSY$ is at least $k \cdot L$. The value obtained by scheduling $T_i$ is at most $(1+\sqrt{k}) \cdot \mathsf{achievedvalue}(I)$ (lemma 5.4). Hence the net gain is less than or equal to

$$\begin{aligned} & (1+\sqrt{k}) \cdot \mathsf{achievedvalue}(I) - k \cdot L \\ \leq\ & (1+\sqrt{k}) \cdot \mathsf{achievedvalue}(I) - (1+\sqrt{k}) \cdot \mathsf{achievedvalue}(I) \\ =\ & 0 \end{aligned}$$

What if $L$ does not equal any of the partial sums? That is, if $\sum_{i \leq j} l_i < L < \sum_{i \leq j+1} l_i$. As in the proof of lemma 5.8, we augment the total value given to the clairvoyant by some non-negative amount. Even with this addition the net gain achieved by the clairvoyant algorithm is bounded by $(1 + \sqrt{k}) \cdot \textsf{achievedvalue}(I)$, hence proving the lemma.
□

• Corollary 5.9 holds with $LATE$ replacing $F$.

Before we continue we must clarify the meaning of **poslaxval** and **zerolaxval** in this setting. **poslaxval** denotes the value obtained by tasks that completed before their zero'th deadline (tasks in $S^p$). **zerolaxval** denotes the total value obtained by tasks that completed at or after that deadline (i.e., tasks in $S^0 \cup S^1 \cup \cdots S^{\lfloor log_2 k \rfloor - 1}$).

• Lemma 5.10 holds without change given these new definitions of **poslaxval** and **zerolaxval**.
• Lemma 5.11 holds with $LATE$ replacing $F \cup S^0$.

**Theorem 7.5** *In the exponential gradual descent model,*
$D^{over}$ *has a competitive factor of* $\frac{1}{(1+\sqrt{k})^2}$.
PROOF.

Proof as in theorem 5.12.
□

## 7.6 Inherent Bounds

The inherent bound given by Baruah et. al. [2, 3] can not be directly applied here. Hence, it is not clear whether $D^{over}$ is optimal in this setting. It might very well be that the introduction of descending value schemes helps the on-line scheduler more then it helps the clairvoyant one. Thus, the question of finding the inherent bounds in this case is open.

## 7.7 Performance Guarantee for Underloaded Periods

In the gradual descent model we define an underloaded collection of tasks as a collection such that all its tasks can be scheduled by the zero'th deadline (i.e., with their full value). It is clear that $D^{over}$ will get 100% of the value for such a collection since it will execute according to earliest deadline first scheduling.

## 7.8 Other Gradual Descent Schemes

In this section we presented a specific scheme of gradual descent. In fact, the current argument can provide the same result for more general schemes of descending value.

All schemes must have the following properties:

- The value density of a task must not go below 1 (used in lemma 5.3).
- For every possible associated task $T_i^z$ of $T_i$,

$$d_i^z < d_i^0 + c_i$$

and

$$(d_i^z - d_i^0) \times k \geq \text{“the value of of } T_i^z\text{”}$$

(used in part 2 of lemma 7.4)

Within these constraints, many schemes are possible. Some tasks can have firm deadlines; others may have descending values. The base of the exponent (2 was an arbitrary choice) can be different for different tasks.

# 8 Situations in Which The Exact Computation Time of A Task Is Not Known

Suppose the on-line scheduling algorithm is not given the exact computation time of a task upon its release. However, for every task $T_i$, an upper bound on its possible computation time denoted by $c_{i,max}$, is given. Also, the actual computation time of $T_i$, denoted by $c_i$, satisfies:

$$(1-\epsilon)\cdot c_{i,max} \leq c_i \leq c_{i,max}$$

Where, $0 \leq \epsilon < 1$ is a given *error margin* which is common to all the tasks. We make the following additional assumptions:

**Assumption 8.1**

- THE ACTUAL COMPUTATION TIME IS ENVIRONMENT-INVARIANT: The actual computation time of a task does not depend on the point in time in which it was scheduled, the number of times it was preempted and rescheduled etc.
- THE ACTUAL COMPUTATION TIME IS NOT KNOW BEFORE THE COMPLETION POINT: An on-line scheduler can not know the exact computation time of a task until it completes.

□

Some terms has to be redefined in the new set up:

**Definition 8.2**

- UNDERLOADED COLLECTION OF TASKS: A collection of tasks is *underloaded* (in this setting) if the actual computation times enable execution of all the tasks to completion.
- IMPORTANCE RATIO: The *importance ratio*, $k$, of a collection with an error margin of $\epsilon$ is defined to be the ratio of the largest *possible* value density to the smallest *possible* value density.

$$k = \frac{\max_i \frac{v_i}{(1-\epsilon)\cdot c_{i,max}}}{\min_i \frac{v_i}{c_{i,max}}} = \frac{1}{(1-\epsilon)}\cdot\frac{\max_i \frac{v_i}{c_{i,max}}}{\min_i \frac{v_i}{c_{i,max}}} \qquad (9)$$

Here, the *normalized importance* assumption (assumption 3.2) means that $\min_i \frac{v_i}{c_{i,max}} \geq 1$.

## 8.1 An Inherent Bound On The Competitive Factor

The inherent bound proof given in [2, 3] can be applied here as well. In the notation of those references, all the *major tasks* execute at their longest possible computation time with an actual value density 1 while all the *associated tasks* execute at their shortest possible computation time and value density $k$. This argument shows that no on-line scheduler can achieve a competitive factor greater than $\frac{1}{(1+\sqrt{k})^2}$.

## 8.2 Underloaded Systems

**Example 8.3** Suppose we are given the following collection of two tasks:

| Task | *Release-Time* | *Max. Computation-Time* | *Value* | *Deadline* |
|---|---|---|---|---|
| $T_1$ | 0 | 1 | 1 | 1 |
| $T_2$ | 0 | 200 | 200 | 200 |

For an error margin $\epsilon < \frac{1}{201}$ this collection will always constitute an *overloaded* system. However, if $\epsilon \geq \frac{1}{201}$ then depending on the actual computation times, the system may be either underloaded or overloaded.

□

**Theorem 8.1** *An on-line scheduler that guarantees 100% of the value for an underloaded system has a zero competitive factor.*

PROOF.

Suppose an on-line scheduler $S$ guarantees 100% of the value for underloaded systems. Suppose the tasks of example 8.3 with error margin of $\epsilon = \frac{1}{200}$ are scheduled by $S$.

Consider the following possible cases:

1. The actual executing time of $T_1$ is the maximum possible — 1 while that of $T_2$ is the minimum possible — 199. In this case the system is underloaded and $S$ should be able to schedule both tasks to completion. That is schedule $T_1$ between 0 and 1 and $T_2$ from 1 to 200.

2. The actual executing time of both $T_1$ and $T_2$ are the maximum possible. In this case the system is overloaded and only one of the tasks can possibly complete. However, $S$ can not distinguish between case 1 and case 2 (not before time 200). Hence, $S$ will schedule $T_1$ between 0 and 1 and $T_2$ will reach its deadline without completing its execution.

In the second case, $S$ obtains a value of 1 out of the possible value of 200. Hence $S$ has a competitive factor of at most $\frac{1}{200}$. Of course the number 200 above is arbitrary and can be as large as wanted, which gives the desired result.

□

## 8.3 Overloaded Systems

Theorem 8.1 shows that we can not guarantee both a positive competitive factor and a 100% the value for an underloaded system.

The *earliest-deadline-first* algorithm is an optimal on-line scheduler for *underloaded* systems. We will show that a version of $D^{over}$ can achieve a competitive factor of $\frac{1}{(1+\sqrt{k})^2+(\epsilon\cdot k)(1+\sqrt{k})+1}$

.

We utilize the following *version* of $D^{over}$:

- $k$ is taken to be as in equation 9.
- $D^{over}$ assumes that the computation time of a task to be the maximum possible — $c_{i,max}$. This affects the values of availtime, *laxity*, *remaining_computation_time* and the *LST* point of a task (statements 16, 19, 20, 31, 50, 54, 59, 60 and 70).

**Theorem 8.2** *$D^{over}$ has a competitive factor of* $\frac{1}{(1+\sqrt{k})^2+(\epsilon\cdot k)(1+\sqrt{k})+1}$
PROOF.

The proof will be an adaptation of the analysis for the case of exact knowledge of computation time in section 5. The following is a list of modification that are needed in that analysis.

1. Lemma 5.5 should read :

$$c_{i,max} \geq d_i - t_{close}$$

   Hence,

$$c_i \geq c_{i,max} \cdot (1-\epsilon) \geq d_i - t_{close} - \epsilon \cdot c_{i,max}$$

2. In this set up lemma 5.8 should be replaced by:

   **Lemma 8.3** *The total net gain from scheduling the tasks abandoned during I is not greater than*

$$(1+\sqrt{k})(1+\epsilon\cdot k)\cdot \mathsf{achievedvalue}$$

   The proof is essentially the same but here the value of $L$ is taken to be [18]:

$$L = \max\{(1+\sqrt{k})\cdot(\frac{1}{k}+\epsilon)\cdot \mathsf{achievedvalue}(I), l_1\}$$

   - The total net gain from those tasks of $F, T_1, T_2, \cdots T_j$, whose total computation time after $t_{close}$ equals $L$, is not greater than

$$k\cdot L = (1+\sqrt{k})(1+\epsilon\cdot k)\cdot \mathsf{achievedvalue}(I)$$

[18]Instead of $L = \max\{\frac{(1+\sqrt{k})\cdot \mathsf{achievedvalue}(I)}{k}, l_1\}$ in section 5

- Every other task, $T_i$ where $j < i \leq m$, has an execution time of at least

$$d_i - t_{close} - \epsilon \cdot c_{i,max} \geq L + l_i - \epsilon \cdot c_{i,max}$$

$T_i$ was scheduled by the clairvoyant scheduler but used only $l_i$ time after $t_{close}$. Hence, $T_i$ executed at least $L - \epsilon \cdot c_{i,max}$ time before $t_{close}$ that is to say in $BUSY$.

$$\begin{array}{lcl} & L - \epsilon \cdot c_{i,max} & \\ \geq & L - \epsilon \cdot v_i & \text{, by assumption 3.2 } c_{i,max} \leq v_i \\ \geq & L - \epsilon \cdot (1 + \sqrt{k}) \cdot \mathsf{achievedvalue}(I) & \text{, by lemma 5.4} \\ \geq & \frac{(1+\sqrt{k}) \cdot \mathsf{achievedvalue}(I)}{k} & \end{array}$$

The "loss" from scheduling $T_i$ during $BUSY$ is at least $k \cdot \frac{(1+\sqrt{k}) \cdot \mathsf{achievedvalue}(I)}{k}$. The value obtained by scheduling $T_i$ is at most $(1 + \sqrt{k}) \cdot \mathsf{achievedvalue}(I)$ (lemma 5.4). Hence the net gain is less than or equal to zero.

3. Lemma 5.10 should state that

$$\begin{aligned} C(S^0 \cup F) &\leq (1 + \sqrt{k})(1 + \epsilon \cdot k) \cdot \mathsf{achievedvalue} + k \cdot BUSY \\ &\leq (1 + \sqrt{k})(1 + \epsilon \cdot k) \cdot \mathsf{achievedvalue} + k \cdot (\mathsf{achievedvalue} + \frac{1}{\sqrt{k}} \cdot \mathsf{lstvalue}) \\ &= (1 + \sqrt{k} + k + (\epsilon \cdot k)(1 + \sqrt{k})) \cdot \mathsf{achievedvalue} + \sqrt{k} \cdot \mathsf{lstvalue} \\ &\leq ((1 + \sqrt{k})^2 + (\epsilon \cdot k)(1 + \sqrt{k})) \cdot \mathsf{achievedvalue} \end{aligned}$$

The first inequality follows from the fact that lemma 5.3 holds without change. The last inequality is due to the fact that **lstvalue** is always less or equal to **achievedvalue**.

Finally we can prove the theorem:

$$\begin{aligned} C(\Gamma) &= C(F \cup S^0 \cup S^p) \leq C(F \cup S^0) + C(S^p) \\ &\leq C(F \cup S^0) + \mathsf{poslaxval} \\ &\leq ((1 + \sqrt{k})^2 + (\epsilon \cdot k)(1 + \sqrt{k})) \cdot \mathsf{achievedvalue} + \mathsf{poslaxval} \\ &\leq ((1 + \sqrt{k})^2 + (\epsilon \cdot k)(1 + \sqrt{k}) + 1) \cdot \mathsf{achievedvalue} \end{aligned}$$

The last inequality is due to the fact that **poslaxval** is always less or equal to **achievedvalue**.

□

## 9 Conclusions

This paper has presented an optimal on-line scheduling algorithm for overloaded systems. It is optimal in the sense that it gives the best competitive factor possible relative to an offline (i.e., clairvoyant) scheduler. It also gives 100% of the value of a clairvoyant scheduler when the system is underloaded. The model accounts for different value densities and generalizes to soft deadlines.

This work leaves many problems open. Here is a small sampling.

- In practice, real-time systems have some periodic critical tasks and other less critical tasks which may be aperiodic. A typical solution (as taken in the Spring Kernel for example [14]) is to devote certain intervals to the critical tasks and to allow the less critical tasks to run during the rest of the time. $D^{over}$ gives its usual guarantee with respect to the less critical tasks (the accounting is a little more difficult since useful time has "holes" in it corresponding to subintervals allocated to critical tasks). A much more subtle question is what is a good competitive algorithm that can take advantage of the cases when a given critical task executes in less time than is allocated for it. We suspect the competitive factor may be worse, since the clairvoyant algorithm might then execute a task that $D^{over}$ has unnecessarily abandoned.

- For the case of uncertain computation time, can the gap between the complexity bound of $\frac{1}{(1+\sqrt{k})^2}$ and the algorithm guarantee of $\frac{1}{(1+\sqrt{k})^2+(\epsilon\cdot k)(1+\sqrt{k})+1}$. be closed?

- For the gradual descent model, is $D^{over}$ an optimal scheduler? What is the inherent bound in this case?

- What guarantees can be given for parallel scheduling algorithms?

- In general, the question of proof tools for such systems is open. We believe that the technique in subsection 5.2 will prove to be very useful.

- What performance guarantees can be given in more general value descending schemes?

# References


[1] T. P. Baker and Alan Shaw. The Cyclic Executive Model and Ada. *The Journal of Real-Time Systems*, 1, 7-25, 1989.

[2] S. Baruah, G. Koren, B. Mishra, A. Raghunathan, L. Rosier and D. Shasha. On-line Scheduling in the Presence of Overload,. *IEEE Foundations of Computer Science Conference*, San Juan, Puerto Rico, pp. 101-110 October 1991.

[3] S. Baruah, G. Koren, D. Mao ,B. Mishra, A. Raghunathan, L. Rosier, D. Shasha and F. Wang. On The competitiveness of On-line Task Real-Time Task Scheduling,. *1991 IEEE Real-Time Systems Symposium* , San Antonio, Texas, pp. 106-115 December 1991. *Invited to the Real-Time Systems journal.*

[4] M. Dertouzos. Control Robotics: the procedural control of physical processes. In *Proc. IFIP congress, 1974*, pp. 807-813, 1974.

[5] M. R. Garey and D. S. Johnson. *Computers and Intractability: a guide to the theory of NP-Completeness.* W. H. Freeman and Company, New York, 1979

[6] A. Karlin, M. Manasse, L. Rudolph, and D. Sleator, Competitive Snoopy Caching, *Algorithmica* **3**, (1988), pages 79–119.

[7] G. Koren, B. Mishra, A. Raghunathan and D. Shasha. On-Line Schedulers for Overload Real-time Systems,. *Technical report no. 558, Courant Institute, NYU* , May 1991.

[8] G. Koren, D. Shasha. An Optimal Scheduling Algorithm with a Competitive Factor for Real-Time Systems,. *Technical report no. 572, Courant Institute, NYU* , July 1991.

[9] C. L. Liu and J. Layland. Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment. *Journal of the ACM*, 20:46-61, 1973.

[10] C. Douglass Locke. Best-Effort Decision Making for Real-Time Scheduling. *Doctoral Dissertation, Computer Science Department, Carnegie-Mellon University*, 1986.

[11] A. Mok. Fundamental Design Problems of Distributed Systems for the Hard Real-time environment. *Doctoral Dissertation, M.I.T.*, 1983.

[12] L. Sha, J. P. Lehoczky and R. Rajkumar. Solutions for some Practical Problems in prioritized preemptive scheduling. *Proceedings, Real-Time Systems Symposium*, 1986.

[13] D. Sleator and R. Tarjan, Amortized Efficiency of List Update and Paging Rules, *CACM* 28, February 1985, pages 202–208.

[14] John A. Stankovic and Krithi Ramamrithm, The Spring Kernel: a new paradigm for real-time systems, IEEE Software, May 1991, pp. 62-72.

[15] F. Wang and D. Mao, *Worst Case Analysis for On-Line Scheduling in Real-Time Systems.* Technical Report 91-54, Department of Computer and Information Science, University of Massachusetts at Amherst, 1991.



Imprimé en France
par
l'Institut National de Recherche en Informatique et en Automatique.