An Optimal Scheduling Algorithm with a Competitive Factor for Real-Time Systems *

Gilad Koren† Dennis Shasha†

July 29, 1991

Abstract

We consider real-time systems in which the value of a task is proportional to its computation time. The system obtains the value of a given task if the task completes by its deadline. Otherwise, the system obtains no value for the task.

When such a system is underloaded (i.e. there exists a schedule for which all tasks meet their deadlines), Dertouzos [6] showed that the earliest deadline first algorithm will achieve 100% of the possible value. We consider the case of a possibly overloaded system and present an algorithm which:

1. behaves like the earliest deadline first algorithm when the system is underloaded.

2. obtains at least 1/4 of the maximum value that an optimal clairvoyant algorithm could obtain even when the system is overloaded.

We implement our algorithm with an amortized cost of $O(\log n)$ time per task, where n bounds the number of tasks in the system at any instant.

1 Introduction

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

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.

We consider a less stringent class of systems denoted as *firm 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 no value from the task.

A system is *underloaded* if there exists a schedule that will meet the deadline of every task. Scheduling underloaded systems is a well-studied topic, and several on-line algorithms have been

*Supported by U.S. Office of Naval Research #N00014-91-J-1472, U.S. National Science Foundation grant #IRI-89-01699.

† Courant Institute, New York University, New York, NY 10012.

proposed for the optimal scheduling of these systems on a uniprocessor [6,8]. Examples of such algorithms include *earliest-deadline-first* (D) and *smallest-slack-time* (SL). However, none of the proposed algorithms make performance guarantees during times when the system is overloaded. In fact, it has been experimentally demonstrated that these algorithms perform quite poorly when the system is overloaded [7].

Practical systems are prone to intermittent overloading caused by a cascading of exceptional situations. A good on-line scheduling algorithm should not only be optimal under normal circumstances, but also should respond appropriately to such overloaded situations. Researchers and designers of real-time systems have devised intelligent on-line heuristics to handle overloaded situations [1,7,9]. The most successful of these heuristics was proposed by Locke as part of the CMU Archons project [7]. While Locke's heuristic is widely used and has been shown to perform well in practice, it offers no performance guarantee.

Our task model is based on the model defined by Locke [7] and used in [2,3,5,11]. Tasks may enter the system at any time. Nothing is known about a task until it arrives. When a task is released, its computation time, value, and deadline are known precisely.

Since we are trying to model firm real-time systems, the value of a task is obtained provided it completes by its deadline (i.e. it succeeds); once its deadline passes, there is no value to completing the task. At any time, a task may be preempted in favor of another task at no cost.

Koren, Mishra, Raghunathan and Shasha [2,5] suggested the first on-line scheduling algorithm with a performance guarantee for an overloaded system. This algorithm is called *D-star* (D*) since it behaves like *earliest-deadline-first* (D) in an underloaded system. They defined *overloaded* and *underloaded intervals* for a given system. This definition is based upon the behavior of the earliest-deadline-first algorithm with respect to these tasks.

D* gives the following performance guarantee: 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, where value is assumed to be proportional to computation time.

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. As in [2,4,10] 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. A *competitive algorithm* is an algorithm with non-zero competitive factor.

Unfortunately, D* is not competitive, since a clairvoyant scheduler can possibly obtain a value that is bigger than the length of the overloaded interval for tasks with deadlines in this interval. If, however, the entire execution is overloaded, then D* guarantees one-fifth of the maximal possible value.

Baruah et. al. [2] and independently Wang and Mao [11] demonstrated, using an adversary argument[1], that, in the same setting, there can be NO on-line scheduling algorithm with a

[1]The adversary creates the task set. At the outset it creates some tasks and observes the on-line scheduler responses to it. Then the adversary creates new tasks according to the scheduler decisions.

competitive factor greater than ONE-QUARTER [2].

Also, after seeing our D* result, Wang and Mao [11] presented an algorithm with the best possible competitive factor—one-fourth. Their algorithm is based on the *least latest start time* (LLST) scheduling algorithm. However, Wang and Mao's algorithm has a major draw-back: it is not optimal in an underloaded system. In other words, a system in which D (and for this matter, also D*) will get 100% of the possible value, Wang-Mao's algorithm might miss deadlines and obtain as little as 1/4 of the value achieved by D*.

It is therefore desirable to have a competitive algorithm that will also be optimal in an underloaded system. Such an algorithm is the main contribution of this paper. We devise an on-line scheduling algorithm that is optimal in an underloaded system and also has the BEST possible competitive factor — one-fourth — again assuming that the value of a task equals its computation time [3].

This algorithm is a melding of D* with a *doubling* algorithm inspired by the Wang and Mao result. We call it *DD-star* (DD*).

DD* 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.

2 The Algorithm

Before we describe the full algorithm, we need some notation. The tasks are denoted by $T_1, T_2 \cdots T_n$. For each task T_i, its release time is denoted by r_i, its computation time and hence its value by c_i and its deadline by d_i.
For each task T_i define Δ_i as follows:

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

In the algorithm described below, there are three kinds of *events* (each causing an associated interrupt) considered: Task Completion (successful termination of a task), Task Release (arrival of a task), and Latest-start-time Interrupt (the indication that a task must immediately be scheduled in order to complete by its deadline). Task Completion interrupts have higher priority than Task Release and Latest-start-time interrupts. Task Release and Latest-start-time interrupts share the same priority. Thus if several interrupts happen simultaneously, then the Task Completion interrupt is handled before the Task Release and Latest-start-time interrupts. It may happen that a Task Completion event removes the condition for a lower priority interrupt, e.g., by making the processor idle.

DD* requires three data structures, called Qdelayed, Qdeadline and Qlst. An entry of Qdeadline and Qlst consists of a single task, whereas an entry of Qdelayed is a 3-tuple (T, Previous-

[2] Baruah et. al. have extensions to this result for the cases of *non-uniform value density* and different *loading factors*. Wang and Mao have extensions for the multi-processor case)

[3] The *value density* of a task is the ratio of its value to its computation time. The *importance ratio*, k, of a set of tasks is the ratio of the largest value density to the smallest value density (see [2]). In a system with importance ratio k, DD* has a competitive factor of at least $1/(4k)$. This can be achieved by a version of DD* that ignores the value information of a task. That is, the value of each task is taking to be its computation length. Baruah et. al. showed that in this setting the best possible competitive factor is bounded by $1/[(1+\sqrt{k})^2]$

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.

Availtime will be the maximum computation time that can be taken by a new task without causing the current task or the delayed tasks to miss their deadlines.

These data structures support Insert, Delete, Min and Dequeue operations. The Min operation for Qdelayed and Qdeadline returns the entry corresponding to the task with the earliest deadline among all tasks in the queue. For Qlst the Min operation returns the entry corresponding to the task with the earliest latest-start-time (LST) among all tasks in the queue. The Dequeue operation on Qdelayed (or Qdeadline) deletes from the queue the element returned by Min, in addition this element is deleted from Qlst. Likewise a Dequeue operation on Qlst will delete an element from Qdeadline.

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, $laxity(T) = deadline(T) - (now() + remaining - computation - time(T))$. ϕ denotes the empty set.

```
1  delayedval  := 0    (* This will be the running value of delayed tasks  *)
                       (* Availtime will be the maximum computation time that can be taken
2  availtime   := ∞    by a new task without causing the current task or the delayed tasks
                       to miss their deadlines. *)
3  Qlst        := φ    (* All tasks are ordered according to their latest start time, LST *)
4  Qdelayed    := φ    (* All delayed tasks ordered by deadline order *)
5  Qdeadline   := φ    (* All non-delayed tasks ordered by deadline order *)
6  idle        := true (* In the beginning the processor is idle *)
```

```
7   loop
8   Task Completion :
9       if Qdelayed is not empty

            (* There are delayed tasks. Remove the first element from Qdelayed. The ele-
10          ment dequeued will be the one with the earliest deadline and, by construction,
            the one last inserted of those in Qdelayed.*)

11          (T_current, t_prev, avail_prev) := Dequeue(Qdelayed);

12          (* Adjust delayedval and availtime as appropriate. *)

13          delayedval := delayedval - value(T_current);

            (* The following value of availtime is the correct one because the element
14          dequeued from Qdelayed is the one last inserted of those in Qdelayed. The
            available computation time has decreased by the time elapsed since this ele-
            ment was inserted to the queue. *)

15          availtime := avail_prev − (now() − t_prev);
16          Schedule T_current;

            (* If Qdelayed contains a task with an earlier deadline than T_current, a Task
17          Release event with this task is generated. This is done in order to force
            earliest-deadline-first scheduling in an underloaded system. *)

18          if Qdeadline is not empty and d_current is later than
              the deadline of the first task in Qdeadline
19              T_arrival := Dequeue(Qdeadline) ;
20              (* Simulate a task release *)
21              Generate a task-release event with T_arrival;
22          end {if}
23      else

24          (* Qdelayed is empty. The current interval is closed here *)

25          if (Qdeadline is not empty)
26            T_current := Dequeue(Qdeadline);
27            Schedule T_current;
28            availtime := laxity(T_current);
              (* A new interval is created with t_b = now() and t_e = now()+c_current. Note
29            that t_e could be after the deadline of T_current, if T_current had been previously
              preempted. *)
30          else
31            idle := true;
32            availtime := ∞;
33          end {if}
34      end {if}
35  end (* task completion *)
```

```
36  Task Release :
    (* Note that this interrupt can be generated by the Task Completion handler
    as well as by the arrival of a new task. *)

37      if (idle )
38          T_current := T_arrival;
39          Schedule T_current;
40          availtime := laxity(T_current);
41          idle  := false;
42            (* A new interval is created with t_b = now() and t_e = now() + c_current. *)
43      else
44          if d_arrival < d_current and
            availtime ≥ remaining-computation-time(T_arrival) then

45            (* No overload is detected, so the running task is preempted and delayed. The
              current interval is extended by the value of T_arrival, t_e := t_e + c_arrival. All
              delayed tasks and the arriving task will complete no later than t_e, provided
              no new tasks enter.
              In statement 44 we refer to the remaining-computation-time and not just
              (total-)computation time because T_arrival might have executed earlier. This
              is possible when the current Task Release event was generated by the Task
              Completion handler. *)

46            Insert T_current into Qlst;
47            Insert (T_current, now(), availtime) into Qdelayed;

48            (* When we insert current task into Qlst and Qdelayed, this task will be, by
              construction, the task with the earliest deadline in Qdelayed*)

49            availtime := availtime - remaining-computation-time(T_arrival) ;
50            availtime := min(availtime , laxity(T_arrival))
51            delayedval := delayedval + value(T_current);
52            T_current := T_arrival;
53            Schedule T_current;
54          else  (* not enough laxity or later deadline  *)
55            Insert T_arrival into Qlst and Qdeadline;
56          end {if}
57      end {if}  (* idle  *)
58  end (* release  *)
```

```
59  Latest-start-time Interrupt :

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

61      T_next = Dequeue(Qlst);

62      (* The interval is extended to include the deadline of T_next :
        If d_next > t_e then t_e := d_next *)

63      if (c_next > 2(c_current + delayedval))
64          Insert T_current into Qlst and Qdeadline;
65          Remove all delayed tasks from Qdelayed and insert them into Qlst and Qdeadline;
66          (* Qdelayed = φ *)
67          delayedval := 0;
68          availtime := 0
69          T_current := T_next;
70          Schedule T_current;
71      else (* c_next is not big enough; it is abandoned. *)
72          Abandon T_next;
73      end {if}
74  end (* LST *)

75   end{loop }
```

DD* : A Competitive Optimal On-line Scheduling Algorithm.

3 Analysis of DD*

In all that follows we assume that DD* schedules the history consisting of tasks $T_1, T_2, \cdots T_n$.

The proof uses intervals in the spirit of Wang-Mao's algorithm. When an interval is created it is considered *opened*, meaning that it may be extended. $I(t)$ denotes the interval I at time t. We will drop the parameter t when it is clear from the context. An interval becomes *closed* when it cannot be extended any further (see statement 24 of DD*). The length of an interval I will also be denoted by I. Let delayedval(t) denote the value of delayedval at time t and achievedvalue(t) denote the value achieved during the current interval before t (i.e. the value of all order-scheduled tasks that completed before t).

Note that the DD* algorithm allows *coincident interrupts*, i.e., several interrupts are allowed to occur at the same time.[4] All of our claims about the behavior of DD* hold even when we

[4]Suppose at some point in time we have a *collection* of interrupt events E waiting to be served. This collection can include at most one Task Completion event, but there is no bound on the number of Task Release and Latest-start-time Interrupt events. The events will be handled according to their priorities, i.e., Task Completion first and then all other interrupts. There is no specific order of service among the several Latest-start-time and Task Release interrupts. This means that the value of $I(t)$ and $delayedval(t)$ might be changed several times (each Latest-start-time Interrupt can change $I(t)$ and each Task Release can change $I(t)$ and $delayedval(t)$) during one time instance t. In this case we will regard these events as if they occurred one after the other but with zero time distance between them.

consider coincident interrupts. The reason is that their proofs rely only on the order of events and not on the specific temporal distance between two consecutive events.

The first task in each interval is scheduled according to deadline order. The subsequent tasks (if any) in this interval may be scheduled according to deadline order or 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, see statement 59 of DD*). Call the first *order-scheduled* and the second *lst-scheduled*. Let us assert and prove that once a task is lst-scheduled all subsequent tasks (if any) of this interval must be lst-scheduled.

Proposition 3.1 *According to the schedule of DD* 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, because when the current task is lst-scheduled availtime equals zero (see statement 68 of DD*) hence no task can be scheduled by the Task Release handler (see statement 44 of DD*)
□

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 in the interval).

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 DD*, we will partition the set of tasks according to the question of whether the task had zero or positive slack time at the last time it was scheduled.

- Let F (for fail) denote the set of tasks that were abandoned.
- Let S^0 (for successful with *0* slack time) denote the set of tasks that were completed successfully and were last scheduled with zero slack time.
- Let S^p (for successful with *p*ositive slack time) denote the set of tasks that were completed successfully and were scheduled (their last time) with positive slack time.

Note: These sets will also denote the corresponding sets of indexes (of tasks).
Before the detailed analysis, let us first study an example of DD*'s scheduling.

Example 3.1 Consider the following overloaded system with six tasks (see table 1). For notational convenience we will denote the tasks by their deadlines, hence for example T_{20} is a task with deadline at time 20.

Task	*Release-Time*	*Computation-Time*	*Deadline*	Δ_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 3.1.

DD* schedules the above history as follows: In the beginning availtime is ∞ and Qdelayed is empty.
First, DD* 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 Qdeadline (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 Qdeadline 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 Qdelayed and Qlst with LST equals 16. Availtime is decremented by the computation time of T_{18}. Its new value is 9.
T_{18} is executing 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.
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 the value of the current task plus delayedval $(2 + 11)$. 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.
At time 5, T_5 completes and since T_{17} is the task with the earliest deadline it is scheduled. Availtime is now the value of availtime when T_{17} was executing (7) minus the time elapsed since it was inserted to Qdelayed $((5 - 4) = 1)$.
The remaining computation time of T_{17} is one unit, hence at time 6 it completes. The next task in Qdelayed 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 Qdelayed $((6 - 3) = 3)$. 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 Qdelayed are moved to Qdeadlineand availtime is reset to zero.
The LST of T_{18} is 16 and of T_{20} (the only other task in Qlst) is 15. These task will generate Latest-start-time Interrupt in these respective times, both will be abandoned.
At time 34, T_{34} completes its execution and DD* finished scheduling this history. Table 2 summerizes the scheduling decisions of DD*.

time	released	preempted (LST)	completed	scheduled	availtime	Qdelayed	delayedval	Qdeadline	*comment*
0					∞	[]	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}	$min(9-2, 15)$	$[T_{18}, T_{20}]$	$5+6$	$[T_{24}, T_{34}]$	
4					$min(9-2, 15)$	$[T_{18}, T_{20}]$	$5+6$	$[T_{34}]$	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: DD* 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 DD* 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 DD* is at time 4.

4 DD* is an optimal scheduling algorithm

The following lemma asserts that availtime is indeed the maximum computation time that can be taken by a new task without causing the current task or the delayed tasks to miss their deadlines [5].

Lemma 4.1 *Suppose $T_{current}$ is the current task and suppose that a task, $T_{arrival}$, with an earlier deadline is released, then $T_{arrival}$ in addition to $T_{current}$ and all the tasks in* Qdelayed *can be scheduled by D, the earlist-deadline-first scheduling algorithm,*

if and only if

$$\text{availtime} \geq \textit{remaining computation time}(T_{arrival})$$

PROOF.

[5] If there is no current task the processor is idle. In this case Qdelayed is empty and availtime equals ∞. The program maintains this condition as follows. Initially idle is true, Qdelayed is empty, and availtime is ∞. Later, idle is set to true only if Qdelayed is empty. In that case, availtime is set to ∞ (see statement 31 of DD*).

The proof is by induction on time. The induction is done separately on each interval.

If $T_{current}$ is the first task in an interval then Qdelayed is empty and availtime is set to $laxity(T_{current})$, see statements 28 and 40 of DD*. In this case, the claim obviously holds.

Assume that the induction is true up to the current time with the currently executing task.

If the current task is order scheduled then it and all the tasks in Qdelayed are ordered according to their release times which is also the ascending order of their deadlines. Moreover, DD* schedules these tasks in *earliest-deadline-first* order (as long as no Latest-start-time Interrupt succeeds in scheduling a task). $T_{arrival}$ will be scheduled if and only if (see statement 44 of DD*)

$$\text{availtime} \geq \textit{remaining computation time}(T_{arrival})$$

By the induction hypothesis this condition indeed guaranties that $T_{arrival}$ can be scheduled without causing any deadline to be missed. The available computation time is now reduced by the amount to be consumed by $T_{arrival}$ (see statement 49 of DD*). However, this is not the only constraint, the available computation time is also bounded by the laxity of $T_{arrival}$. This constraint is due to the fact that if another task T with an even earlier deadline were to be released, then T and $T_{arrival}$ would have to complete before $T_{arrival}$'s deadline. Hence, the minimum between these two time bounds is the new available computation time (see statement 50 of DD*).

If the current task completes, then the task with the earliest deadline in Qdelayed is scheduled, call it T_{next} (see statement 11 of DD*). What is the available computation time now? When T_{next} was inserted into Qdelayed it was accompanied by the available computation time at that point (see statement 47 of DD*). The tasks of Qdelayed at the dequeuing point are exactly the same tasks as in the insertion point because Qdelayed is a *first-in-first-out* queue. None of these tasks executed during the interval between the insertion and dequeuing of T_{next}. Hence, the current available computation time is the value at the time of insertion less the elapsed time. A new task can be scheduled if and only if its computation time is less than or equal to this value, which is the new value of availtime (see statement 15 of DD*).

The induction proceeds until the interval is closed or until the first lst-scheduled task (in this interval) is scheduled.

Suppose no task was lst-scheduled the interval is closed when the processor is idle and Qdelayed is empty (see statement 24 of DD*). In this case, the induction process is complete.

Suppose the current interval had an lst-scheduled task. An lst-scheduled task has no laxity. Any preemption of such a task will cause it to miss its deadline. Hence, the available computation time equals 0. Indeed, whenever the current task is lst-scheduled availtime is zero (see statement 68 of DD*). According to proposition 3.1 once a task is lst-scheduled all subsequent (in this interval) are lst-scheduled hence the induction is complete.

Thus, the induction hypothesis holds during the entire (open) interval and the claim is proved. □

Theorem 4.2 *DD* is an optimal scheduling algorithm. That is, if any scheduler can schedule the entire history, then DD* can do it.*

PROOF.

If any scheduling algorithm can schedule the entire history, so can the *earliest-deadline-first* scheduling algorithm (D) which is known to be optimal for an underloaded system [6]. We assert and prove that DD* behaves like D in an underloaded system.

We observe that D can be characterized by the following two rules:

1. The current task is preempted in favor of a newly released task if the latter has an earlier deadline.
2. The task to be scheduled after task-completion is the one with the earliest deadline among all ready tasks.

Suppose D successfully scheduled a history H. Let DD* schedule the same history. We will prove by induction that both algorithms make exactly the same scheduling decisions.

In the beginning the system is idle. Suppose that until time t, D and DD* made the same scheduling decision.

What events can happen at time t?

- A new task, $T_{arrival}$, is released:
  If $d_{arrival} < d_{current}$ then D will preempt $T_{current}$ and will schedule the arriving task. We know that D succeeded in scheduling all the tasks hence there is (at time t) enough laxity to schedule all the ready tasks. Hence, DD* finds (statement 44 of DD* and lemma 4.1) that there is enough laxity and preempts $T_{current}$ in favor of $T_{arrival}$.
  If $d_{arrival} \geq d_{current}$ both algorithms continue to execute the current task.
- A task completion:
  Since both algorithms made the same scheduling decisions until t, both have a task-completion event at t.
  D schedules the ready-task with the earliest deadline. DD* does the same (see statements 11 and 18).
- Latest-start-time Interrupt:
  Suppose there is a Latest-start-time Interrupt event because T_{lst} reached its LST. We know that $d_{lst} \geq d_{current}$ otherwise T_{lst} would have been the current executing task. This means that only one of $T_{current}$ and T_{lst} can possibly complete. This is true for both schedulers, contradicting the fact that D successfully scheduled the entire history. Hence, a Latest-start-time Interrupt is impossible.
  Note that it is possible that one task LST coincides with another task completion point. However, since task completion events are given priority over Latest-start-time Interrupt events, the handling of the task completion event would clear the condition for the Latest-start-time Interrupt event. The task that reached its LST will be scheduled without raising the Latest-start-time Interrupt.

□

Corollary 4.3 *According to DD* scheduling, in an underloaded system there will be no* Latest-start-time Interrupt.

4.1 DD* is $\frac{1}{4}$-competitive

The main problem with D is that when the system becomes overloaded, D can miss arbitrarily many deadlines. For example, suppose there is a set of tasks with computation time 2 and deadlines $2, 4, 6, 8, \ldots$. If a new task enters at time 0 with computation time 1 and deadline 1, then that task will be the only one to complete under D. DD* does as well as D for the underloaded case and is competitive with a clairvoyant algorithm when the system is overloaded.

We will prove that DD* has a competitive factor by showing that it gets "enough" value from each interval. We will bound the value that a clairvoyant algorithm can get from all the intervals and will show that this is at most 4 times the value DD* can get.

In the following *covered* means pointwise inclusion. That is, a time interval I is covered by a collection of intervals W if every point in I is in some interval of W

In DD*, when the processor is not idle there is one and only one currently open interval. The reason is that an interval can be closed only in a Task Completion event (see comment 24 of DD*). Such an event can immediately open the next open interval (when Qdeadline is not empty, see statements 25 to 29 of DD*) or can set the variable idle to true (statement 31). In the later case, since idle is true, a new interval can be opened by a Task Release, but not before setting idle back to false (see statements 37 to 42).

In the next lemma we show that the current (open) interval always covers the current time. Recall that achievedvalue(t) denotes the value achieved during the current interval before t.

Lemma 4.4 *According to the schedule of DD* if at time t the current open interval, $I(t)$, is $[t_b, t_e]$ the currently executing task is T_i and T_i is order-scheduled, then*

$$t - t_b \leq \mathsf{delayedval}(t) + \mathsf{achievedvalue}(t) + c_i \leq I(t)$$

PROOF.

When the processor becomes idle, the previously open interval is closed (see statements 24 and 31 of DD*). Since the same interval is open between t_b and t, the processor could not have been idle during that time. The length of the interval between t_b to t cannot therefore be greater than the sum of the computation times (and therefore values) of all tasks that were scheduled between t_b and t.

Since T_i is order-scheduled, all previously scheduled tasks in I are order-scheduled. These tasks either completed or were interrupted by a task release. If a task completes, then its value is added to achievedvalue. If the task is preempted then its value is added to delayedval (see statement 51 of DD*). Hence, the sum of the values of all the tasks that were scheduled in $I(t)$ before t equals delayedval(t) + achievedvalue$(t) + c_i$, and the first inequality is proved.

As for the second inequality, since all previously scheduled tasks in $I(t)$ are order-scheduled, the length of $I(t)$ must be at least the sum of their computation times and therefore their values (see comments and 29, 42 and 45 of DD*). □

Whereas the previous lemma was mainly concerned with the prefix of the current interval between t_b and t, the following lemmas deal with the entire interval at time t which (see comments 29, 42, 45 and 62 of DD*) may be longer than $t - t_b$. The lemmas are proved using induction.

The induction is on the scheduling decisions of DD*. A scheduling decision is made each time an *event* occurs. The possible events are a Task Release, a Task Completion or a Latest-start-time Interrupt. Each scheduling decision may change the current task and cause a change in the values of the length of the current interval, delayedval or achievedvalue. Let I_{new}, $\mathsf{delayedval}_{new}$ and $\mathsf{achievedvalue}_{new}$ denote those values immediately after the event.

Lemma 4.5 *According to DD* scheduling, if at time t the current open interval, I, is $[t_b, t_e]$ and the executing task, T_i, is order-scheduled, then*

$$|I(t)| \leq 3(\mathsf{delayedval}(t) + \mathsf{achievedvalue}(t) + c_i)$$

PROOF.

By induction on the order of the scheduling decisions (events) of DD*.

A new interval is created (*opened*) when the processor is idle and a new task is released (see comment 42) or when Qdelayed is empty and Qdeadline is not empty (see comment 29 of DD*).

Consider the following two cases at time t_b: the processor is idle and T_k is released; or Qdelayed is empty and T_k is the first task of Qdeadline. In either case, the interval $[t_b, t_b + c_k]$ is created. In this case the entire length of I is c_k, hence the induction hypothesis holds for the base case, for $i = k$ (note that T_k is order-scheduled).

Assume that the induction is true for the current executing task T_k with the open interval $[t_b, t_e]$ and that T_k is **order-scheduled**.

Which events can occur (at time t)?

1. T_k completes and Qdelayed is empty:
   The interval I is closed at t (see statement 24 of DD*). So, the induction process completes.
2. T_k completes and T_l is the first task in Qdelayed :
   The length of the interval is not changed. At time t, c_k is added to achievedvalue and c_l is reduced from delayedval (see statements 10 to 15 of DD*).
   Hence,

$$\begin{aligned} |I| &\leq 3(\mathsf{delayedval}(t) + \mathsf{achievedvalue}(t) + c_k) \\ &= 3((\mathsf{delayedval}_{new} + c_l) + (\mathsf{achievedvalue}_{new} - c_k) + c_k) \\ &= 3(\mathsf{delayedval}_{new} + \mathsf{achievedvalue}_{new} + c_l) \end{aligned}$$

   and the induction holds for $i = l$
3. T_l is released and preempts T_k:
   In this case the interval is extended by c_l, i.e the new right end-point of I is set to $t_e + c_l$ (see statements 45 to 53 of DD*). At time t, c_k is added to delayedval and achievedvalue does not change.
   Hence,

$$
\begin{aligned}
|I_{new}| &= |I + c_l| \\
&\leq 3(\mathsf{delayedval}(t) + \mathsf{achievedvalue}(t) + c_k) + c_l \\
&= 3(\mathsf{delayedval}(t) + c_k + \mathsf{achievedvalue}(t)) + c_l \\
&< 3(\mathsf{delayedval}_{new} + \mathsf{achievedvalue}_{new} + c_l)
\end{aligned}
$$

and the induction holds for $i = l$

4. T_l is released but is not scheduled (either T_l has a later deadline or there is not enough laxity):
   In this case (see statement 55 of DD*) none of the relevant parameters[6] is changed, hence the induction still holds.

5. T_l reaches its LST at time t and causes a Latest-start-time Interrupt, but T_l is **not** scheduled:
   Since T_l is not scheduled we know that $c_l \leq 2(c_k + \mathsf{delayedval}(t))$ (see statement 63 of DD*). The only parameter[7] that might be changed is the length of I. I is changed only if $d_l > t_e$. Assume $d_l > t_e$ then,

$$
\begin{aligned}
|I_{new}| &= (t - t_b) + (d_l - t) \\
&\leq \mathsf{achievedvalue}(t) + (\mathsf{delayedval}(t) + c_k) + c_l \\
&\leq \mathsf{achievedvalue}(t) + 3(\mathsf{delayedval}(t) + c_k) \\
&\leq 3(\mathsf{delayedval}_{new} + \mathsf{achievedvalue}_{new} + c_k)
\end{aligned}
$$

   The first inequality is due to following two facts. First,

$$t - t_b \leq delayed(t) + achievedvalue(t) + c_k$$

   by lemma 4.4. Second, $d_l - t =$ the *remaining computation time*; that time must be less than the original computation time c_l.
   Hence, the induction still holds.

6. T_l reaches its LST at time t and causes a Latest-start-time Interrupt. T_k is abandoned and T_l is scheduled:
   Note that this case introduces an lst-scheduled task (see statements 64 to 70 of DD*). All subsequent tasks (if any) will also be lst-scheduled (proposition 3.1). Hence, the induction process is complete.

The induction hypothesis holds in all cases, hence the claim is proved.

□

[6] The relevant parameters are the length of the current interval, the current task, delayedval and achievedvalue.

[7] Note that T_l can not be one of the delayed tasks, since there is enough laxity for all of Qdelayed tasks (lemma 4.1).

Lemma 4.6 *According to DD* scheduling, if at time t the current open interval, I, is $[t_b, t_e]$, and the executing task, T_i, is lst-scheduled, then*

$$t - t_b < 3 \text{ achievedvalue}(t) + 2c_i$$

and,

$$|I(t)| < 3 \text{ achievedvalue}(t) + 4c_i$$

PROOF.

The proof is by induction on the scheduling decisions of DD*.

The base case here is the first time that a task is scheduled due to a Latest-start-time Interrupt[8].

First, let us prove the induction hypothesis for the base case. Suppose that T_k is **order-scheduled** and at time t, T_l reaches its LST. T_l causes a Latest-start-time Interrupt, T_k is abandoned and T_l is scheduled.

Since T_l was scheduled we know that $c_l > 2(c_k + \text{delayedval}(t))$. The length of I might be changed and the value of delayedval is set to zero (see statements 62 and 70 of DD*).

- $d_l > t_e$
  In this case I is extended to include d_l.

$$\begin{aligned} |I_{new}| &= (t - t_b) + (d_l - t) \\ &\leq ((\text{delayedval}(t) + \text{achievedvalue}(t) + c_k) + c_l) \\ &\leq \text{achievedvalue}(t) + (\text{delayedval}(t) + c_k) + c_l) \\ &< \text{achievedvalue}(t) + \frac{3}{2}c_l \end{aligned}$$

  As in item 5 in lemma 4.5 above, the first inequality is due to following two facts. First,

$$t - t_b \leq delayed(t) + achievedvalue(t) + c_k$$

  by lemma 4.4. Second, $d_l - t =$ the *remaining computation time*; that time must be less than the original computation time c_l.

- $d_l \leq t_e$

$$\begin{aligned} |I_{new}| &= |I(t)| \\ &\leq 3(\text{delayedval}(t) + \text{achievedvalue}(t) + c_k) \\ &< 3 \text{ achievedvalue}(t) + \frac{3}{2}c_l \end{aligned}$$

  The first inequality is due to lemma 4.5 above.

[8] Note that the first tasks in an interval are always order-scheduled (see comments 29 and 42 of DD*).

We proved the claim for the base case. Now, let us prove the induction step. Assume that the induction is true for the current executing task T_k with the open interval $[t_b, t_e]$ and that T_k is **lst-scheduled**.

What can the next event (at time t) be?

1. T_k completes:
   Interval I is closed at t (statement 24). No other task will be scheduled during I so the induction process is complete.
2. T_l is released:
   When the current task is lst-scheduled a newly released task will not be scheduled (see proposition 3.1)
   So, in this case none of the relevant parameters is changed, hence the induction still holds.
3. T_l reaches its LST at time t and causes a Latest-start-time Interrupt. T_l is not scheduled:
   Since T_l was not scheduled we know that $c_l \leq 2c_k$. (Note that delayedval becomes zero once an lst-scheduled task is scheduled see statement 67 of DD*, delayedval stays zero until the interval is closed). The only parameter that might be changed is the length of I. This will happen only if $d_l > t_e$.
   Assume $d_l > t_e$,

$$\begin{aligned} |I_{new}| &= (t - t_b) + (d_l - t) \\ &< 3\ \mathsf{achievedvalue}(t) + 2c_k + c_l \\ &\leq 3\ \mathsf{achievedvalue}(t) + 4c_k \end{aligned}$$

   The first inequality follows from the induction hypothesis and the fact that $c_l \geq T_l$'s *remaining computation time* $= d_l - t$. Hence, the induction still holds.
4. T_l reaches its LST at time t and causes a Latest-start-time Interrupt. T_k is abandoned and T_l is scheduled:
   Since T_l is scheduled we know that $c_l > 2c_k$ (see statement 63 of DD*, as before, delayedval equals zero). The length of I might be changed.
   For any t_0 such that $t_0 \geq t$, if T_l is still executing at t_0 then $t_0 - t \leq c_l$ and

$$\begin{aligned} t_0 - t_b &= (t_0 - t) + (t - t_b) \\ &< c_l + 3\ \mathsf{achievedvalue}(t) + 2c_k \\ &< 3\ \mathsf{achievedvalue}(t) + 2c_l \end{aligned}$$

   Hence, the first part of the induction hypothesis is proved.
   As for the second part:

   - $d_l > t_e$
     In this case I is extended to include d_l (see comment 62 of DD*).

$$\begin{aligned} |I_{new}| &= (t - t_b) + (d_l - t) \\ &< 3\ \mathsf{achievedvalue}(t) + 2c_k + c_l \\ &< 3\ \mathsf{achievedvalue}(t) + 2c_l \end{aligned}$$

- $d_l < t_e$

$$\begin{aligned} |I_{new}| &= |I| \\ &< 3\ \mathsf{achievedvalue}(t) + 4c_k \\ &< 3\ \mathsf{achievedvalue}(t) + 4c_l \end{aligned}$$

The induction hypothesis holds in all cases, hence the claim is proved.
□

Corollary 4.7 *Consider any closed interval I. If the total value achieved by order-scheduled tasks, during the period in which I was open, is* ordervalue *and the value achieved by an lst-scheduled task, during the same period, is* lstvalue*, then*

$$|I| \leq 3\ \mathsf{ordervalue} + 4\ \mathsf{lstvalue}$$

Note that ordervalue *or* lstvalue *may equal zero.*

PROOF.

Let T_i be the **last** task to complete in I at t_c. (There must be one such task because each time a task is preempted another feasible task, i.e a task that can still meet its deadline, is scheduled). An interval is closed only as a result of task completion event (see statement 24). T_i can be order or lst scheduled, in both cases it is the last task to be **scheduled** in I while I is still open.

Let t be a point in I just prior to t_c but after all the events that preceded T_i's completion. Since t is after all the events in I except the end of the interval, the values of $I(t)$, $\mathsf{ordervalue}(t)$ and $\mathsf{delayedval}(t)$ are the final value of the corresponding variables for the interval I.

- T_i is order-scheduled
  From lemma 4.5 we know that

  $$|I| = |I(t)| \leq 3(\mathsf{delayedval}(t) + \mathsf{ordervalue}(t) + c_i)$$

  But $\mathsf{delayedval}(t)$ must be zero (otherwise the next delayed task would have been scheduled at t_c). Also, $\mathsf{ordervalue}(t_c) = \mathsf{ordervalue}(t) + c_i$. Hence,

  $$|I| \leq 3\ \mathsf{ordervalue}$$

- T_i is lst-scheduled
  From lemma 4.6 we get that

$$|I| = |I(t)| < 3 \text{ ordervalue}(t) + 4c_i$$

  But lstvalue $= c_i$ and ordervalue $=$ ordervalue(t). Hence,

$$|I| < 3 \text{ ordervalue} + 4 \text{ lstvalue}$$

□

Lemma 4.8 *According to the DD* algorithm, if at time t the processor is not idle then $t \in \bigcup I$ (i.e. the union of all intervals cover all the time in which some task was scheduled).*

PROOF.

Suppose T_i is running at time t and was scheduled at time t_s.

If T_i is order-scheduled then, by lemma 4.4, t is in the interval that is open at time t.

If T_i is lst-scheduled then at time t_s (when T_i began) there was already (an open) interval that started at or before t_s. If this interval did not include $d_i \geq t$ then it is must have been extended to d_i (see statement 62 of DD*).

In any case t is included in an interval.

□

Lemma 4.9

$$\bigcup_{i \in S^0 \cup F} \Delta_i \subseteq \bigcup I.$$

PROOF.

For $T_i \in S^0$, T_i completes its execution at exactly d_i. There can be no idle time between r_i and d_i (i.e. Δ_i) because T_i is ready to execute during all this period. From lemma 4.8 we can conclude that Δ_i is covered by the union of the intervals created by DD*.

For $T_i \in F$, suppose T_i was abandoned at some point, call it t_a ($t_a \in \Delta_i$). As in the previous paragraph, there is no idle time between r_i and t_a and hence $[r_i, t_a]$ is covered. T_i is abandoned at t_a only if it reached its LST at t_a but it was not scheduled. By inspection of the algorithm (see comment 62 of DD*) the currently open interval is extended to include d_i. So, $[t_a, d_i]$ is covered by the current interval which means that the entire Δ_i is covered by the union of the intervals created by DD*.

□

Lemma 4.10 *A clairvoyant scheduler can not achieve a value greater than*

$$|\cup_{i \in S^0 \cup F} \Delta_i| + \sum_{i \in S^p} c_i$$

PROOF.

The tasks of $F \cup S^0$ can execute only during $\bigcup_{i \in S^0 \cup F} \Delta_i$, hence the total possible value from this task is $|\bigcup_{i \in S^0 \cup F} \Delta_i|$. Of course the total possible value from the rest of the tasks (i.e. exactly S^p) is $\sum_{i \in S^p} c_i$ and the proof is complete.
□

Theorem 4.11 *The DD* algorithm achieves a competitive factor of* $\frac{1}{4}$
PROOF.

The total value achieved by DD* is:

$$\sum_{i \in S^0} c_i + \sum_{i \in S^p} c_i$$

Note that any two intervals are not necessarily disjoint. However, the periods in which these intervals are open are disjoint. Intuitively, corollary 4.7 shows that the value achieved during the open period is "big enough" for the full length interval. Formally,

$$4 \sum_{i \in S^0} c_i + 3 \sum_{i \in S^p} c_i \quad \geq \quad \sum I$$

Hence,

$$\begin{aligned} \sum_{i \in S^0} c_i + \sum_{i \in S^p} c_i \quad &\geq \quad \frac{\sum I + \sum_{i \in S^p} c_i}{4} \\ &\geq \quad \frac{|\bigcup_{i \in S^0 \cup F} \Delta_i| + \sum_{i \in S^p} c_i}{4}. \end{aligned}$$

The last inequality is due to lemma 4.9. By lemma 4.10, this value is at least a quarter of the maximal achievable value.
□

4.2 The Complexity

Theorem 4.12 *If n bounds the number of unscheduled tasks in the system at any instant then each task incurs only an $O(\log n)$* **amortized** *cost.*
PROOF.

DD* requires three data structures, called Qdelayed, Qdeadline 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.

DD* 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 of this clauses.

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 at most one Task Completion event and at most one Latest-start-time Interrupt event. Hence, while scheduling m tasks there are at most m Task Completion and Latest-start-time Interrupt events. However, a task can cause more than one Task Release events, one is at its actual release time and the others may be simulated Task Release events as a result of Task Completion (see statements 19 and 20 of DD*). One Task Completion event can cause at most one simulated Task Release event. Hence the total number of Task Release events is bounded by $2m$.

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

- In the handler for the Task Release event (statement 36), there is a constant number of queue operations. Hence, this contributes a total of $O(2m)$ queue operations during the entire history.
- In the handler for the Task Completion event (statement 8) there is a constant number of queue operations (aside from a possible simulated Task Release event). 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 59), the number of queue operations is proportional to the number of tasks in Qdelayed plus a constant. How many tasks can be in Qdelayed throughout the history? A task can enter Qdelayed only as a result of Task Release event (actual or simulated), there are at most $2m$ such events. Hence, the total number of tasks in Qdelayed is at most $2m$, which means that the total number of queue operations is $O(2m)$ during the entire history.

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

□

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,. *1991 IEEE Symposium on Foundations of Computer Science* , San Juan, Puerto Rico, 1991.

F. Wang and D. Mao.

[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, December 1991.

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

[5] 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.

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

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

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

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

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

[11] F. Wang and D. Mao. Worst Case Analysis for On-Line Scheduling in Real-Time Systems. Private communication, Submitted to RTSS, 1992.