Open access • Proceedings Article • DOI:10.1109/REAL.1994.342713

On-line scheduling to maximize task completions — Source link

Baruah, Haritsa, Sharma

Institutions: Indian Institute of Science

Published on: 07 Dec 1994 - Real-Time Systems Symposium

Topics: Scheduling (computing) and Performance metric

On-Line Scheduling to Maximize Task Completions

Sanjoy K. Baruah*
The University of Vermont
Jayant Haritsa and **Nitin Sharma**
Indian Institute of **Science**

Abstract

The problem of uniprocessor scheduling under conditions of overload is investigated. The system objective is to maximize the number of tasks that complete by their deadlines. For this performance metric it is shown that, in general, any *on-line algorithm may perform arbitrarily poorly as compared to a clairvoyant scheduler. Restricted instances of the general problem for which on-line schedulers can provide a guaranteed level of performance are identified, and on-line algorithms presented for these special cases.*

1 Introduction

A primary objective of safety-critical real-time systems is to meet all task deadlines. To achieve this goal, system architects typically attempt to anticipate every eventuality and design the system to handle all of these situations. Such a system would, under ideal circumstances, never miss deadlines and behave as expected by the system designers. In reality, however, unanticipated emergency conditions may occur wherein the processing required to handle the emergency exceeds the system capacity, thereby resulting in missed deadlines. The system is then said to be in *overload*. If this happens, it is important that the performance of the system degrade gracefully (if at all). A system that panics and suffers a drastic, fall in performance in an emergency is likely to contribute to the emergency, rather than help solve it.

Scheduling algorithms that work well under normal (non-overloaded) conditions often perform miserably upon overload. Consider, e.g., the Earliest Deadline scheduling algorithm [9], which is used extensively in uniprocessor real-time systems. This algorithm, which preemptively processes tasks in deadline order, is optimal under non-overloaded conditions in the sense that it meets all deadlines whenever it is feasible to do so [4]; however, under overload, it has been observed to perform worse than even *random* scheduling [7, 6].

System Model. We adopt the "firm-deadline" real-time model [5] in this study. In this model, only tasks that complete execution before their deadlines are considered to be successful; tasks that miss their deadlines are considered worthless and are immediately discarded without being executed to completion.

Each input task T is independent of all other tasks and is completely characterized by three parameters: $T.a$ (the *request time*), $T.e$ (the *execution requirement*), and $T.d$ (the *relative deadline*). The task's *absolute deadline* is represented by $T.D \stackrel{\text{def}}{=} (T.a + T.d)$. The interpretation of these pararneters is that task T, for successful completion, needs to be allocated the processor for $T.e$ units of time during the interval $[T.a, T.D)$. We assume that the system learns of a task's parameters only at the instant when it makes the service request. There is no *a priori* bound on the number of tasks that may make requests. Tasks may be preempted at any stage in their execution, and there is no cost associated with such preemptions.

We focus our attention here on the study of overload in *uniprocessor* systems. Our goal is to compare the performance of *on-line algorithms* — algorithms that make scheduling decisions at run time with no prior knowledge about the occurrence of future events — against that of an optimal off-line (or clairvoyant) algorithm. We will refer to the ratio of the worst-case performance of an on-line algorithm with respect to an optimal off-line algorithm as the *competitive factor* of the on-line algorithm.

Overload performance Metrics. For the firm-deadline uniprocessor model, two contending measures

*Supported in part by NSF grants OSR-9350540 and CCR-9410752, and UVM grant PSCI94-3. Mailing address: Department of Electrical Engineering and Computer Science/ 351 Votey Building/ The University of Vermont/ Burlington, VT 05405. Electronic mail: sanjoy@cs.uvm.edu

1052-8725/94 $04.00© 1994 IEEE

of the "goodness" of a scheduling algorithm under conditions of overload are **effective processor utilization (EPU)** and **completion count (CC).** Informally, EPU measures the amount of time during overload that the processor spends on executing tasks that complete by their deadlines, while CC measures the number of tasks executed to completion during the overloaded interval. Which measure is appropriate in a given situation depends, of course, upon the application. For example, EPU may be a reasonable measure in situations where tasks ("customers") pay at a uniform rate for the use of the processor, but are billed only if they manage to complete, and the aim is to maximize the value obtained. By contrast, CC may make more sense when a missed deadline corresponds to a disgruntled customer, and the aim is to keep as many customers satisfied as possible. Of course, many real-life applications are best modeled by modifications to these measures, or perhaps even some combination of them.

Performance Results. With respect to the EPU metric, it has been proved that no uniprocessor on-line scheduling algorithm can guarantee an EPU greater than 25 percent under conditions of overload [3, 2]. This bound has also been shown to be tight [2, 8]. These results hold in the general case, when the deadlines of the input tasks may be arbitrarily "tight" or stringent. Recently, the effect on EPU in environments where there is a ***limit*** on the stringency of task deadlines has been studied [1].

In this paper, we turn our attention to studying the impact of overload when CC (completion count) is the measure of scheduler performance. To the best of our knowledge, this is the first work to study this metric in the context of on-line overload guarantees. For this metric, we prove that in the general case, ***any*** uniprocessor on-line algorithm can be made to perform *arbitrarily* worse than a clairvoyant (or off-line) scheduler; that is, no on-line algorithm can have a non-zero competitive factor. This is certainly a disappointing result from the perspective of developing overload-resistant scheduling algorithms. We have, however, obtained more positive results for several restricted forms of the problem which may occur quite often in practice. Each of the special cases that we have considered require certain constraints to hold on the values of the task parameters. These special cases and the corresponding results are summarized below:

Equal Request Times (ERT): In this case, all tasks in the overloaded interval request service at the same time, that is, requests are made in ***bulk.*** A practical example is a switch in a communication network which cyclically polls its incoming links to forward packets that have arrived since its previous servicing of the link. On each link, different packets may have different processing times and deadlines as in, for example, a multimedia application. For this situation, we present an optimal on-line scheduling algorithm, that is, its competitive factor is 1.

Equal Execution Times (EET): Here, all tasks are identical with regard to their processing times. This can happen in an antiaircraft battle system, for example, where the same time is taken to process any target but targets may arrive at random and may have different firing deadlines based on the distance and speed of the aircraft.

For EET task systems, we show that any on-line non-preemptive scheduling algorithm will deliver a competitive factor of 1/2.

Monotonic Absolute Deadlines (MAD): Here, a task that requests service is guaranteed to have an absolute deadline no earlier than that of all tasks that have previously requested service. Monotonic deadlines correspond, in a sense, to first-come first-served fairness: a task that requests service later is not allowed to demand completion of service prior to that of an earlier request.

For MAD task systems, we present an efficient on-line scheduling algorithm that has a competitive factor of 1/2. We establish the tightness of the 1/2 bound by proving that no on-line scheduling algorithm may have a competitive factor greater than 1/2.

Equal Relative Deadlines (ERD): Here, all tasks have the same relative deadline. Such a situation may arise, for example, in environments where every customer is guaranteed service within a fixed duration from arrival – for example, some pizzerias offer a lunch special "within 15 minutes, or it's free" irrespective of the time it takes to prepare the specific pizza which is ordered. Note that ERD is a refinement of the monotonic absolute deadlines case.

For ERD task systems, we show that no on-line algorithm can have a competitive factor better than 2/3. We are currently investigating whether this bound is tight.

Equal Absolute Deadlines (EAD):

Here, all tasks have the same absolute deadline. A practical example is a computerized stock exchange wherein transactions arrive in random fashion and require different processing times but all transactions that arrive on a given day have to be cleared by closing time. Note that EAD is a refinement of the monotonic absolute deadlines case.

For EAD task systems, we present an optimal scheduling algorithm, that is, its competitive factor is 1.

Organization. The rest of this paper is organized as follows: We present, in Section 2, the definitions and notation that are used in the remainder of the paper. In Section 3, we prove that no on-line algorithm can have a non-zero competitive factor for the CC performance metric. In Section 4, we consider the special cases described above and show that more positive results may be obtained for some of them. We conclude in Section 5, with a summary of the results presented here.

2 Definitions and Notation

A system is said to be in **overload** if no scheduling algorithm can satisfy all task requests that are made on the system. As mentioned in the Introduction, the Earliest Deadline algorithm is optimal in the sense that it will successfully schedule any set of task requests which are in fact schedulable. Given this optimality of the Earliest Deadline algorithm, it follows that a system is in overload if the Earliest Deadline algorithm fails to meet the deadline of some task in the system.

Each input task T is independent of all other tasks and is completely characterized by three parameters: a *request time* $T.a$, an *execution requirement* $T.e$, and a *relative deadline* $T.d$. The task's absolute deadline is represented by $T.D \stackrel{\text{def}}{=} (T.a + T.d)$. With respect to a particular schedule, a task T is said to be **active** at time-instant t if (i) $t \geq T.a$; i.e., the task requests service by time t, (ii) $T.e_r > 0$, where $T.e_r$ is the remaining amount of processor time that needs to be allocated to task T before its deadline, and (iii) $t < T.D$; i.e. the deadline has not yet been reached. An active task T is **non-degenerate** at time t if $T.e_r \leq (T.D - t)$; i.e. its remaining execution requirement is not greater than the remaining time to its deadline. An active task **completes** if it receives its full execution requirement before its deadline expires.

Given a scheduling algorithm $\mathcal{A}$ and a set of tasks τ, let $\mathcal{A}.\tau$ denote the schedule generated by $\mathcal{A}$ on τ. Let $\mathcal{C}.(\mathcal{A}.\tau)$ denote the set of tasks executed to completion, and $c.(\mathcal{A}.\tau)$ denote the *number* of tasks executed to completion, by scheduling algorithm $\mathcal{A}$ when operated on the taskset τ — $c.(\mathcal{A}.\tau) \stackrel{\text{def}}{=} |\mathcal{C}.(\mathcal{A}.\tau)|$. Let $\mathsf{OPT}.\tau$ denote the schedule that *maximizes* the number of completions for the same taskset τ. On-line algorithm $\mathcal{A}$ is said to have a **competitive factor** r, $0 \leq r \leq 1$, iff $\mathcal{A}.\tau$ is guaranteed to complete at least r times as many tasks as $\mathsf{OPT}.\tau$, for any τ. That is,

$$\forall \tau :: c.(\mathcal{A}.\tau) \geq r \;\; c.(\mathsf{OPT}.\tau)$$

3 An upper bound on performance

Theorem 1 *The competitive factor of any on-line schedulzng algorithm is arbitrarily close to zero.*

Proof: For any on-line scheduling algorithm $\mathcal{A}$, we describe below a set of tasks τ such that $\mathcal{A}$ completes exactly one task in τ, while an off-line schedule for τ completes at least η tasks. It follows that as $\eta \rightarrow \infty$, the competitive factor of $\mathcal{A}$ (equal to 1/17) becomes arbitrarily poor. The exact procedure for task generation is detailed in Figure 1 (a zero-slack task T is one in which the execution requirement $T.e$ is equal to the relative deadline $T.d$). We provide an informal rationale below for the generation process.

Initially, at some time t_c, $\mathcal{A}$ is offered a choice of 2 tasks: (i) task T_i, which requires 2 units of processor time by a deadline of $t_c + 2$, and (ii) Task T_j, which requires e units of processor time by a deadline of $t_f = t_c + (e + 1)$, where $e \gg 2$. Any scheduling of T_i or T_j must be done within the interval $[t_c, t_f)$; we refer to this as the interval of interest.

- If $\mathcal{A}$ executes T_j at all over $[t_c, t_c + 2)$, then it cannot hope to complete T_i on time. Two new tasks T_i' and T_j' are then added to τ at time $t_c' = t_c + 2$, with T_i' requiring 2 units of processor time by a deadline of $t_c' + 2$, and T_j' requiring $(e - 2)$ units of processor tinie by a deadline of t_f. Clearly, d can hope to complete at most one of the two tasks T_j or T_j' on time; without loss of generality, assume $\mathcal{A}$ gives task T_j' priority over T_j. The situation at time t_c' is then virtually identical to the situation at time t_c, with the tasks T_i' and T_j' playing the roles of tasks T_i and T_j, and $[t_c', t_f)$ the new interval of interest. Furthermore, (i) $\mathcal{A}$ has as yet

```
Adversary(η)
/* A call to function gen(e, d) generates a task
T with T.a = t_v, T.e = e · t_v, and T.d = d · t_v.
Variables t_c and t_v represent the "current time"
and the "time scale factor" respectively. */
  t_c := 0.0; t_s := 1.0;
  e := 2η;
  gen(2, 2); gen(e, e + 1);
  for i := 1 to (η − 1) do
    if A executes a zero-slack task over
       the interval [t_c, t_c + 2e · t_s/(1 + e))
    then
      t_v := t_c + (2e · t_s)/(1 + e);
      t_v := t_s/(1 + e);
      gen(e, e + 1)
    else
      t_c := t_c + 2t_s;
      e := e − 2;
      gen(2, 2); gen(e, e + 1)
    fi
  od
end
```

Figure 1: The Adversary Algorithm

executed no tasks to completion, and (ii) an off-line schedule can execute task T_i over $[t_c, t_v + 2)$ and thus have completed one task – T_i – and be in the same situation as **A** at time t'_c.

- If **A** executes T_i exclusively over $[t_c, t_c + 2e/(1 + e))$, then **A** cannot hope to complete T_j on time. A new task T'_j is generated at time $t'_c = t_v + 2e/(1 + e)$, requiring $e/(1 + e)$ units of processor time by a deadline of $t'_c + 1$. Task T_i, meanwhile, requires $2/(1+e)$ (i.e., $2-(2e/(1+e))$) more units of processor time by a deadline of $t'_c + 2/(1 + e)$. The situation at time t'_c is then again virtually identical to the situation at time t_v, except that all execution-requirements and relative deadlines as well as the size of the interval of interest are scaled by a factor of $1/(1 + e)$, with the tasks T_i and T'_j playing the roles of tasks T_i and T_j, and $[t'_c, t'_c + 1)$ the new interval of interest. Furthermore, (i) $\mathcal{A}$ has as yet executed no tasks to completion, and (ii) an off-line schedule can execute task T_j over $[t_c, t'_v)$ and $[t'_c + 1, t_c + (1 + e))$ and be in the same situation as **A** at time t'_c. Since $t'_c - t_c + (t_c + (1+e)) - (t'_c + 1) = e$, task T_j would have completed in the off-line schedule.

Notice that, in both the above cases, neither **A** nor the off-line algorithm has allocated the processor at all over the new interval of interest. The above argument can therefore be repeated over this new interval of interest. By doing so $(\eta - 1)$ times, with one of the above cases being made to occur each time, we see that an off-line schedule executes $(\eta - 1)$ tasks to completion, and **A**, none. After $\eta - 1$ iterations, $\mathcal{A}$ can then be allowed to execute either of the two currently active tasks to completion; the off-line schedule does the same, thus ensuring that **A** has completed 1 task to the off-line schedule's η tasks. It follows that as $\eta \to \infty$, the competitive factor of **A** (equal to $1/\eta$) becomes arbitrarily poor. □

4 Special Cases

When the measure of performance of an on-line algorithm is the number of tasks executed to completion, we have seen that, in general, on-line algorithms perform poorly vis-a-vis off-line ones. In this section, we identify restricted kinds of task sets for which on-line algorithms may be expected to perform better with respect to their competitive factor. The special cases impose certain constraints on the values of the task parameters, but are representative of workloads encountered in many real systems.

4.1 Equal Request Times (ERT)

We first consider task sets τ in which all tasks in the overloaded interval have the same request times, that is, requests are made in bulk. Since all the necessary information — the request times, execution requirements, and deadlines of all tasks in τ – is known a *priori*, scheduling such a task set is not really an "on-line" problem. For this case, we have the following result:

Theorem 2 *The* **EDD** *(Earliest Deadline with Discard) Algorithm, shown in Figure 2, has a Competitive factor of 1 on equal-request-lime task systems.*

Proof Sketch: The basic idea of the **EDD** algorithm is to create a deadline-ordered sequence of the entire task set and then to iteratively keep removing the largest execution time task from the sequence until the remaining set of tasks becomes feasible with an Earliest Deadline (ED) schedule.

When all tasks have the same request times, the problems of preemptive scheduling and non-preemptive scheduling on a uniprocessor are identi-

```
EDD(τ) /* Earliest Deadline with Discard */
  Sort τ by increasing deadline, such that
  ⟨T_1, T_2, ..., T_n⟩ is a permutation of the
  tasks in τ with T_i.D ≤ T_{i+1}.D for all
  i, 1 ≤ i < n;
  demand := 0
  for i := 1 to n do Discarded[i] := 0 od
  for i := 1 to n do
    demand := demand + T_i.e
    if demand > T_i.D then
      Let
      T_k be the task from among T_1, T_2, ..., T_i
      with the largest execution requirement
      that has not yet been discarded; i.e., T_k ∈
      {T_1, T_2, ..., T_i} ∧ Discarded[k] ≠ 1 ∧ (∀j :
      j ≤ i : T_k.e ≥ T_j.e);
      Discarded[k] := 1;
      demand := demand − T_k.e
    fi
  od
  Let τ' = {T_i | Discarded[i] = 0}
  Schedule τ' using the ED algorithm
end EDD
```

Figure 2: The EDD Algorithm

cal. in the sense that every set of tasks that can be scheduled preemptively can also be scheduled without preemption. In [10], an algorithm is presented for non-preemptive scheduling to maximize task completions for bulk request systems. This algorithm turns out to be essentially equivalent to **EDD**; we therefore refer the interested reader to [10] for a proof of the correctness of **EDD**. □

4.2 Equal Execution Times (EET)

We now move on to considering the case where all tasks have equal execution times.

A scheduling algorithm is said to use no inserted idle time if the processor is never idle while there are active non-degenerate tasks that need to be scheduled.

Theorem 3 *Any on-line non-preemptive scheduling algorithm that uses no inserted idle time is* 1/2 *competitive on equal-execution-time task systems.*

Proof Sketch: We prove this result by using induction on n, the number of tasks completed by the optimal off-line scheduler in the overloaded set τ. In the following proof, let **NPT** to denote a generic non-preemptive scheduling algorithm that uses no inserted idle time.

Lemma 4.1 *Given a feasible schedule that completes n tasks, any non-preemptive on-line algorithm will complete at least $\lceil \frac{n}{2} \rceil$ tasks.*

Without loss of generality, assume that all tasks need unit service, and that $\min_{T \in \tau}\{T.a\} = 0$.

Basis: The lemma is observed to be true for $n = 1$ and $n = 2$.

Step: Suppose the lemma is true for all $n \leq (k-1)$. We now show that it is true for $n = k$. Consider the optimal schedule $\mathcal{S}$ which completes k tasks in the interval $[0, z)$. Let T' denote the first task that completes in $\mathcal{S}$, and let t^* denote its completion time. Now replace all the time intervals in $\mathcal{S}$ where T^* is scheduled with idle periods – obviously, the total time of these "holes" adds up to unit time. The next step is to *compact* the modified schedule $\mathcal{S}$ until t^* by "sliding" all task executions in the interval $[0, t^*)$ to the right until all the holes left by the removal of T' have been covered up. The compaction, upon completion, will result in a free slot of unit size in the interval $[0,1)$. Note also that the sliding does not affect the completion status of any of the remaining tasks, since all these tasks have deadlines greater than t^*.

Now identify the task T_i which was executed in the interval $[0, 1)$ in **NPT.** Create a new schedule R wherein T_i is non-preemptively scheduled in the interval $[0,1)$, with the remainder of R being identical to that of the compacted $\mathcal{S}$ schedule, except that the execution intervals of task T_i are replaced by idle periods, if T_i was executed in $\mathcal{S}$.

By inspection, it is clear that the new schedule $\mathcal{R}$ is feasible for the interval $[1, x)$ and in this interval completes the task set $\tau - \{T^*, T_i\}$, which is of size $(k-2)$.

Applying the induction hypothesis to schedule $\mathcal{R}$, we note that **NPT** will complete $\lceil \frac{k-2}{2} \rceil$ tasks in the interval $[1, x)$. Therefore, over the entire interval $[0, x)$, **NPT** will complete at least $1 + \lceil \frac{k-2}{2} \rceil$ tasks, that is, at least $\lceil \frac{k}{2} \rceil$ tasks.

We therefore conclude that any non-preemptive algorithm completes at least half of the number of tasks completed by the optimal off-line scheduler, resulting in a competitive factor of $1/2$. □

The above upper bound on competitive factor is tight for non-preemptive on-line scheduling algorithms, since it is easily proven that no on-line non-preemptive algorithm can deliver a competitive factor greater than $1/2$ for equal-execution-time task sets. This is shown in the following example. Consider the set of tasks T_1, T_2, and T_3 described by the following temporal characteristics: $T_1.a = 0, T_1.e = 1, T_1.d = 2$, $T_2.a = 0.5, T_2.e = 1, T_2.d = 1.5$, and $T_3.a = 0.9, T_3.e = 1, T_3.d = 1.9$. For this set of tasks, NPT will obviously complete only T_1, whereas an optimal scheduler would complete T_1 and T_2 over the interval $[0,2)$ by using preemption.

4.3 Monotonic Absolute Deadlines (MAD)

Task system τ is said to be **monotonic absolute deadline** iff it is guaranteed that a newly-arrived task will not have a absolute deadline before that of any task that has previously arrived, i.e.,

$$\forall T_i, T_j : T_i, T_j \in \tau : T_i.a < T_j.a \Rightarrow T_i.D \leq T_j.D$$

The **Smallest Remaining Processing Time First (SRPTF)** on-line scheduling algorithm allocates the processor at every instant to the non-degenerate task with smallest remaining execution requirement. We prove below in Theorem 4 that SRPTF is a reasonably good on-line scheduling algorithm for monotonic-deadline systems, in that it always performs at least half as well as an optimal algorithm. Furthermore, we show in Theorem 5 that we cannot hope to do better, that is, $1/2$ is an upper bound on the competitive factor of any on-line scheduling algorithm for MAD task systems.

Theorem 4 *The SRPTF algorithm is $1/2$-competitive on monotonic-absolute-deadline task systems.*

Proof Sketch: Suppose OPT executes a set of n tasks in some order. Let $T_1, T_2, \ldots, T_n$ be the deadline-ordered sequence of the same set of tasks. Then there exists a schedule which successfully completes all these tasks consecutively in the order $T_1, T_2, \ldots, T_n$. This follows from the optimality of Earliest Deadline for non-overload conditions.

We view OPT's schedule as consisting of n disjoint intervals, $I_1, I_2, \ldots, I_n$, where I_i is the interval over which OPT scheduled task T_i. We assess SRPTF's performance for each such interval to prove SRPTF's competitiveness. Since, at any instant, SRPTF schedules the shortest remaining execution time task, one of the following must hold for every $I_i = [T_i.t_s, T_i.t_f)$,

Statement (1) : SRPTF completes some task T_k at or before $T_i.t_f$, or

Statement (2) : SRPTF has already completed T_i before $T_i.t_s$.

Note that SRPTF scheduling tasks other than T_i over I_i may mean that it might not be able to complete T_i later. However, it will certainly not affect the feasibility of tasks other than T_i.

The interval I_i is defined to be "good" for SRPTF if :

Case 1 : SRPTF completes T_i before $T_i.t_s$, or completes it at some later time after completing T_k, or

Case 2 : T_k (as defined by Statement 1) is not completed by OPT at all. In this case, SRPTF might lose T_i, but it completes T_k in return (which OPT does not). So, its performance is the same as that of OPT for this interval.

Conversely, the interval I_i is "bad" for SRPTF if OPT completes both T_i and T_k, while SRPTF misses out on T_i. But, notice that (i) SRPTF can lose at most one task (i.e., T_i) for a bad interval, and (ii) Each bad interval results in at least one good interval, that is, interval I_k corresponding to task T_k, since Case 1 would be true for I_k.

From the above we conclude that at most $n/2$ intervals can be bad for SRPTF and it therefore misses at most half the tasks that OPT completes. It then immediately follows that SRPTF is 1/2 competitive. □

Theorem 5 *The Competitive factor of any on-line scheduling algorithm operating on monotonic-absolute-deadline task sets is arbitrarily close to $1/2$.*

Proof Sketch: For any on-line scheduling algorithm A, we describe below a set of tasks τ such that either (i) $\mathcal{A}$ completes m tasks in τ while an off-line schedule for τ completes at least $2m$ tasks, or (ii) A completes $k+1$ tasks in τ, while an off-line schedule for τ completes at least $2k$ tasks. In the former case, the competitive factor of $\mathcal{A}$ is clearly $1/2$. In the latter case, the competitive factor of A is $(1/2 + 1/2k)$; it follows that as $k \rightarrow \infty$, the competitive factor of A becomes arbitrarily close to $1/2$.

The task generation process is such that initially, at some time t_c, A is offered a choice of 2 tasks: (i) task T_i, which requires 2 units of processor time by a deadline of $t_c + 2$, and (ii) Task T_j, which requires 1 unit of processor time by a deadline of $t_c + 3$.

Case (1) If $\mathcal{A}$ executes T_j at all over $[t_c, t_c+1)$, then it cannot hope to complete T_i on time. Task set τ in this case consists of T_i and T_j. An off-line schedule would schedule T_i over $[t_c, t_c+2)$, and T_j over $[t_c+2, t_c+3)$, thus completing two tasks.

Case (2) If $\mathcal{A}$ executes T_i exclusively over $[t_c, t_c+1)$, then two tasks T_1 and T_2 are added to τ at time t_c+1, each requiring 1 unit of procesor time by a deadline of t_c+3. Task T_i now needs to be scheduled for the next unit of time by $\mathcal{A}$ in order to complete, and T_j, T_1, and T_2 each need to be scheduled for one of the next 2 units of processor time by $\mathcal{A}$ in order to complete. We consider 2 cases:

Case (2.1) If $\mathcal{A}$ schedules T_i exclusively over $[t_c+1, t_c+2)$, or if it schedules at most 2 of the three tasks T_j, T_1, or T_2 over $[t_c+1, t_c+2)$, then a new task T_3 is added to τ at time t_c+2, requiring 0.5 units of processor time by a deadline of $t_c+3.5$.

If $\mathcal{A}$ schedules T_i, or exactly one of the tasks T_j, T_1, or T_2, over $[t_c+1, t_c+2)$, then that task completes in $\mathcal{A}$ at time t_c+2. If $\mathcal{A}$ schedules 2 of the 3 tasks T_j, T_1, or T_2 over $[t_c+1, t_c+2)$, then neither of the two tasks will have completed by t_c+2; however, both may complete by t_c+3. Without loss of generality, therefore, we may assume that one task has completed at time t_c+2, and (at least) another one needs to be scheduled for the time unit $[t_c+2, t_c+3)$ in order to complete. Let $t'_c \stackrel{\text{def}}{=} t_c+2$, T'_i ? one of the tasks that need to be scheduled for the next time unit (i.e., one of T_j, T_1, T_2 that has not been scheduled over $[t_c, t_c+2)$), and $T'_j \stackrel{\text{def}}{=} T_3$.

Case (2.2) Otherwise, at most one of the tasks T_j, T_1, or T_2 will be completed by $\mathcal{A}$, while an off-line algorithm could schedule T_j over $[t_c, t_c+1)$, T_1 over $[t_c+1, t_c+2)$, and T_2 over $[t_c+2, t_c+3)$.

In Cases (1) and (2.2) above, $\mathcal{A}$ has executed exactly one task to completion, while an off-line algorithm will have executed at least two tasks to completion.

In Case (2.1) above, $\mathcal{A}$ executes one task over $[t_c, t_c+2)$ (and may complete another by t_c+3). However, an off-line schedule completes two tasks over $[t_c, t_c+2)$ by executing T_j over $[t_c, t_c+1)$, and T_1 over $[t_c+1, t_c+2)$. Furthermore, the situation at time $t'_c = t_c+2$ is virtually identical to the situation at time t_c, with all task parameters — execution requirements and relative deadlines — halved, with tasks T'_i and T'_j playing the roles of tasks T_i and T_j respectively. The above argument may therefore be recursively applied whenever Case (2.1) occurs. Doing this $k-1$ times would result in $\mathcal{A}$ having completed $k-1$ tasks, while an off-line algorithm completes $2(k-1)$ tasks. The kth time, tasks T_1, T_2, T_3 are not generated: both $\mathcal{A}$ and an off-line algorithm therefore complete 2 tasks. On this sequence of task requests, therefore, the number of tasks completed by $\mathcal{A}$ is $k+1$, and the number completed by an optimal algorithm is $2k$. □

4.4 Equal Relative Deadlines (ERD)

We now move on to the case where all tasks in the overloaded interval have the same relative deadline. For this case, we have the following result:

Theorem 6 *No* on-line algorithm can have a *competitive factor* greater than $2/3$ *for* equal-relative-deadline *task* systems.

Proof Sketch: Without loss of generality, assume that the relative deadline of all tasks is 1. Let **ONL** denote the on-line algorithm. Consider the following task sequence: At $t = 0$, a task T_1, with $T_1.e = 1$ arrives. Later, at $t = 0.25$, another task T_2 arrives ($T_2.D$ is therefore 1.25), with $T_2.e = 0.25$.

Case 1 : If ONL executes T_2 at all over $[0.25, 0.5)$, then it cannot hope to complete T_1. Hence, it completes only T_2, whereas an off-line scheduler would be able to execute T_1 until t=1 and then execute T_2.

Case 2 : If ONL schedules only T_1 over $[0.25, 0.5)$, the adversary generates two more tasks T_3 and T_4 at t=0.5, with $T_3.e$ = T4.e = 0.5 (note that T3.D = T4.D = 1.5). ONL can now complete at most two of the four active tasks before t=1.5, since T_1, T_3, T_4 all have requirement of 0.5 units. In contrast, the off-line scheduler would execute T_2 over $[0.25, 0.5)$, and then complete T_3 and T4.

At best, therefore, ONL completes 2 tasks whereas the off-line algorithm can complete 3. This implies that no on-line algorithm has a competitive factor greater than 2/3 for the equal relative deadlines case. □

The above theorem presents an upper bound on the competitive factor. We are currently working on establishing whether the above bound is tight. As discussed in the Introduction, equal-relative-deadlines is a special case of monotonic-absolute-deadlines. Therefore, from Theorem 4, it is clear that the **SRPTF** algo-

rithm will deliver a competitive factor of at least 1/2 in the ERD case.

4.5 Equal Absolute Deadlines (EAD)

We now consider the case where all tasks in the overloaded interval have the same absolute deadline, that is, we have "bulk deadlines". For this case, we have the following result:

Theorem 7 *The* **SRPTF** *Algorithm has a competitive factor of 1 on equal absolute deadline task systems.*

Proof Sketch: It has been shown [11] that a SRPTF schedule will always have completed at least as many tasks as any other schedule at any observation time. Given this result, it is straightforward to see that if a common deadline was drawn for all tasks, **SRPTF** would have completed at least the same number of tasks as any other schedule by the time of the deadline. □

5 Conclusions

Earlier studies that have investigated the overload performance characteristics of real-time scheduling algorithms have done so with respect to the effective processor utilization (EPU) metric. In this paper, we have studied the impact of overload when completion count (CC) is the measure of algorithm performance. To the best of our knowledge, this is the first work to study this metric in the context of on-line overload guarantees.

Our study showed that, in the general case, no on-line algorithm can have a non-zero competitive factor. We then considered several special cases, representative of practical systems, for which more positive results were obtained. In particular, we described optimal on-line scheduling algorithms for the following cases: Equal Request Times, Monotonic Absolute Deadlines, and Equal Absolute Deadlines. For Equal Request Times and Equal Absolute Deadlines, the performance of the optimal on-line scheduler was equal to that of the optimal off-line scheduler, while for the Monotonic Absolute Deadlines case, the on-line scheduler could deliver one half the performance of the clairvoyant scheduler. We also presented an upper bound on the achievable competitive factor for the Equal Relative Deadlines case. For the Equal Execution Times situation, we showed that any non-preemptive on-line scheduling algorithm has a competitive factor of 1/2 and that this bound is tight for non-preemptive algorithms. Our results are summarized in the following table (the last three columns describe the upper bound on achievable competitive factor (CFB), the competitive factor that we have managed to achieve (CFA), and the algorithm which delivers this competitive factor).

Case	Definition	CFB	CFA	Algorithm
ERT	$T.a$'s are equal	1	1	EDD
EET	$T.e$'s are equal	1	1/2	NPT
MAD	$T_i.a < T_j.a \Rightarrow T_i.D \leq T_j.D$	1/2	1/2	SRPTF
ERD	$T.d$'s are equal	2/3	1/2	SRPTF
EAD	$T.D$'s are equal	1	1	SRPTF

References

[1] S. Baruah and J. Haritsa. ROBUST: A Hardware Solution to Real-Time Overload. In *Proceedings of the 13th ACM SIGMETRICS Conference,* Santa Clara, California, pages 207–216, May 1993.

[2] S. Baruah, G. Koren, D. Mao, B. Mishra, A. Raghunathan, L. Rosier, D. Shasha, and F. Wang. On the competitiveness of on-line real-time task scheduling. In *Real-Time Systems,* **4:125–144**, 1992. Also in *Proceedings of the 12th Real-Time Systems Symposium, San Antonio, Texas, December 1991.*

[3] S. Baruah, G. Koren, B. Mishra, A. Raghunathan, L. Rosier, and D. Shasha. On-line scheduling in the presence of overload. In *Proceedings of the 32nd Annual IEEE Symposium on Foundations of Computer Science,* San Juan, Puerto Rico, October 1991. IEEE Computer Society Press.

[4] M. Dertouzos. Control robotics : the procedural control of physical processors. In *Proceedings of the IFIP Congress,* pages 807–813, 1974.

[5] J. Haritsa, M. Carey, and M. Livny. On being optimistic about real-time constraints. In *Proc. of the 1990 ACM Principles of Database Systems Symposium,* April 1990.

[6] J. Haritsa, M. Carey, and M. Livny. Earliest-deadline scheduling for real-time database systems. In *Proceedings of the 12th IEEE Real-Tame Systems Symposium,* San Antonio, Texas, December 1991.

[7] E. Jensen, D. Locke, and H. Tokuda. A time-driven scheduling model for real-time operating systems. In *Proceedings of the 6th IEEE Real-Time Systems Symposium*, December 1985.

[8] G. Koren and D. Shasha. D^{over}: An optimal on-line scheduling algorithm for overloaded real-time systems. Technical Report TR 594, Computer Science Department, New York University, 1992.

[9] C. Liu and J. Layland. Scheduling algorithms for multiprogramming in a hard real-time environment. In *Journal of the ACM,* January 1973.

[10] J. Moore. An n job, one machine sequencing algorithm for minimizing the number of late jobs. In *Management Science,* 15(1), 1968.

[11] L. Schrage. A proof of the optimality of the shortest remaining processing time discipline. In *Operations Research,* 16, pages 687–690, 1968.