On-Line Scheduling to Maximize Task Completions

Sanjoy Baruah
CSEE
University of Vermont
Burlington, VT 05405

Jayant Haritsa
SERC
Indian Institute of Science
Bangalore 560012, INDIA

Nitin Sharma
CSA
Indian Institute of Science
Bangalore 560012, INDIA

Abstract

The problem of uniprocessor scheduling under conditions of real-time overload is investigated here. The system objective is to maximize, in the overloaded interval, the number of tasks that complete by their deadlines. For this performance metric we prove that, in general, *any* on-line algorithm may perform arbitrarily poorly as compared to a clairvoyant scheduler. We identify restricted instances of the general problem for which on-line schedulers can provide a guaranteed level of performance, and present on-line algorithms 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.

A practical example of the above situation is the classic Earliest Deadline scheduling algorithm [15], which is used extensively in uniprocessor real-time systems. This algorithm, which preemptively processes tasks in deadline order, is optimal under normal (non-overload) conditions in the sense that it meets all deadlines whenever it is feasible to do so [3]; however, under overload, it has been observed to perform worse than even *random* scheduling [7, 8]. In fact, continuing to use Earliest Deadline scheduling in an emergency is probably the worst thing that a system can do!

Due to the above reasons, the design of "good" scheduling algorithms for overloaded systems has recently become an active area of real-time research (e.g. [1, 5, 12]). We address some aspects of the overload problem in this paper.

System Model

We adopt the "firm-deadline" real-time model [10] in this study. In this model, only tasks that *fully* complete before their deadlines are considered to be successful, whereas tasks that miss their deadlines are considered to be worthless and are immediately discarded without being executed to completion. (Preparation of papers for submission to conferences is an example of a firm-deadline system !)

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 parameters 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.

We focus our attention here on the study of overload in *uniprocessor* systems. The processor scheduling algorithms that we consider may be classified as *on-line* – these algorithms make scheduling decisions only at run-time, and typically do not possess any prior knowledge about the occurrence of future events. Our goal is to compare the performance of on-line algorithms 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. Obviously, the competitive factor is limited to the interval [0,1].

We assume that tasks can be preempted at *any* stage in their execution, and that there is no cost associated with such preemptions. Both preemptive and non-preemptive schedulers are included in the scope of our study. However, all the results presented here, except one, are with respect to premptive algorithms. Therefore, unless explicitly stated otherwise, statements made in the rest of the paper are in the context of preemptive schedulers.

Overload Performance Metrics

For the above firm-deadline uniprocessor model, two contending measures 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 uni-processor on-line scheduling algorithm can guarantee an EPU greater than 25 percent under overload [2, 1]. This bound has also been shown to be tight [1, 5]. The above results are applicable for the general case where 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 in [12].

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. However, the good news is that we have 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 (request time $T.a$, execution requirement $T.e$, relative deadline $T.d$, and absolute deadline $T.D$). These special cases and their 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.

Note that this is strictly *not* an "on-line" problem, since all necessary information becomes available at the common request time, before any scheduling decisions need to be made.

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 newly 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 FCFS 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.

While we have considered several varieties of constraints on task parameters in the above special cases, it is possible, of course, to come up with others. For example, where the ratio of the largest task to the smallest task with respect to processing times is bounded, etc. We have restricted our attention in this paper to the above set since they appear to be representative of many real-world systems. A summary of the above results is shown 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 is equal	1	1	EDD
EET	T.e is 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 is equal	2/3	1/2	SRPTF
EAD	T.D is equal	1	1	SRPTF

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. Then, in Section 4, we consider the special cases discussed 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, and outline future research directions.

For the sake of completeness, we include in Appendix A a proof that, in the general case, Earliest Deadline has an arbitrarily poor competitive factor with respect to both the EPU and CC metrics. This proof is reproduced from [9]. We have also proved that Earliest Deadline exhibits arbitrarily poor overload performance in all the special cases that are considered here – this is necessary to demonstrate the need for developing new scheduling algorithms for these special cases. Due to space limitations, we have omitted these proofs – they are available in [13].

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 $T.D \stackrel{\text{def}}{=} (T.a + T.d)$. 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, and (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 $T.e_r < (T.D - t)$; 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. Otherwise, it is **discarded** at the instant its deadline expires.

The *laxity* l of a task is the difference between its relative deadline and its execution time, that is, $T.l \stackrel{\text{def}}{=} (T.d - T.e)$. The *slack factor* f of a task is defined to be the ratio of its relative deadline to its execution time, that is, $T.f \stackrel{\text{def}}{=} (T.d/T.e)$. A task is said to be **tight** if its execution requirement is equal to its relative deadline, that is, its slack factor is 1. It is easy to see that for a tight task to complete, it has to be non-preemptively fully executed as soon as it makes its request.

Given a scheduling algorithm $\mathcal{A}$ and a set of tasks τ, let $\mathcal{A}.\tau$ denote the processing schedule of the tasks. Let $\mathcal{C}.(\mathcal{A}.\tau)$ denote the *set* of completions and $c.(\mathcal{A}.\tau)$ denote the *number* of completions that were effected by scheduling algorithm $\mathcal{A}$ when operated on the taskset τ. Let $\mathsf{OPT}.\tau$ denote the (possibly preemptive) schedule that *maximizes* the number of completions for the same taskset τ. With these definitions, an 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,

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

For all the on-line algorithms that are considered here, we assume that they do not artificially introduce idle periods in the schedule, that is, the processor is never made to be idle when there are active non-degenerate tasks available in the system.

In the following sections, we derive our overload performance results, using the above terminology and notation.

```
Adversary(η)
/* A call to function gen(e, d) generates a task
T with T.a = t_c, T.e = e · t_s, and T.d = d · t_s.
Variables t_c and t_s 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 tight task over
       the interval [t_c, t_c + 2e · t_s/(1 + e))
    then
       t_c := t_c + (2e · t_s)/(1 + e);
       t_s := 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

3 An upper bound on performance

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

Proof Sketch: 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 $\eta \to \infty$, the competitive factor of $\mathcal{A}$ (equal to $1/\eta$) becomes arbitrarily poor as compared to the off-line scheduler. The exact procedure for task generation is detailed in Figure 1. 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) A tight 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. In this situation, one of the following cases is made to occur:

- If $\mathcal{A}$ at all executes T_j 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 time by a deadline of t_f. Clearly, $\mathcal{A}$ 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 executed no tasks to completion, and (ii) an off-line schedule can execute task T_i over $[t_c, t_c+2)$ and thus have completed one task – T_i – <u>and</u> be in the same situation as $\mathcal{A}$ at time t_c'.

- If $\mathcal{A}$ executes T_i exclusively over $[t_c, t_c + 2e/(1+e))$, then $\mathcal{A}$ cannot hope to complete T_j on time, since $2e/(1+e))$ is greater than 1, which was the laxity for T_j. A new task T_j' is generated at time $t_c' = t_c + 2e/(1+e)$, 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_c, 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 $[t_c, t_c')$ and $[t_c'+1, t_c+(1+e))$ <u>and</u> be in the same situation as $\mathcal{A}$ 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 $\mathcal{A}$ 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 $\mathcal{A}$, 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 $\mathcal{A}$ has completed 1 task to the off-line schedule's η tasks. It follows that as $\eta \to \infty$, the competitive factor of $\mathcal{A}$ (equal to $1/\eta$) becomes arbitrarily poor with respect to the off-line scheduler. □

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 many workloads encountered in 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-time task systems.*

Proof Sketch: When all tasks have the same request times, the problems of preemptive scheduling and non-preemptive scheduling on a uniprocessor are identical, in the sense that every set of tasks that can be scheduled preemptively can also be scheduled without preemption. In [6], an algorithm is presented for the 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 [6] for a proof of the correctness of EDD.

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. □

4.2 Equal Execution Times (EET)

We now move on to considering the case where all tasks have equal execution times. For this case, we do not as yet have provable results for preemptive

EDD(τ) /* Earliest Deadline with Discard */
Sort τ by increasing deadline, such that $\langle T_1, T_2, \ldots, T_n \rangle$ is a permutation of the tasks in τ with $T_i.D \le T_{i+1}.D$ for all i, $1 \le i < n$;
$\mathsf{demand} := 0$
for $i := 1$ **to** n **do** $\mathsf{DISCARDED}[i] := 0$ **od**
for $i := 1$ **to** n **do**
  $\mathsf{demand} := \mathsf{demand} + T_i.e$
  if $\mathsf{demand} > T_i.d$ **then**
    Let T_k be the task from among $T_1, T_2, \ldots, T_i$ with the largest execution requirement that has not yet been discarded; i.e., $T_k \in \{T_1, T_2, \ldots, T_i\} \wedge \mathsf{DISCARDED}[k] \neq 1 \wedge (\forall j : j \le i : T_k.e \ge T_j.e)$;
    $\mathsf{DISCARDED}[k] := 1$;
    $\mathsf{demand} = \mathsf{demand} - T_k.e$
  fi
od
Let $\tau' = \{T_i | \mathsf{DISCARDED}[i] = 0\}$
Schedule τ' using the ED algorithm
end EDD

Figure 2: The EDD Algorithm

scheduling algorithms. However, an encouraging result is available for *non-preemptive* scheduling algorithms. This is perhaps surprising since results about non-preemptive scheduling algorithms typically tend to be negative. The result is as follows:

Theorem 3 *Any on-line non-preemptive scheduling algorithm is* $1/2$ *competitive (with respect to a clairvoyant preemptive scheduler) 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, we use NPT to denote a generic non-preemptive scheduling algorithm.

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. The proof then proceeds as follows:

Basis: The theorem is true for $n = 1$ and $n = 2$ tasks.

Step: Suppose the theorem is true for $n \le (k-1)$ tasks. We now show that it is true for $n = k$ tasks. Consider the optimal schedule $\mathcal{S}$ which completes k tasks in the interval $[0, x)$. Let T^* denote the <u>first</u> 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 $\mathcal{R}$ wherein T_i is non-preemptively scheduled in the interval [0,1), with the remainder of $\mathcal{R}$ 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 $1/2$. It is important to note here that the competitive factor of $1/2$ is with respect to a *preemptive* clairvoyant scheduler. We are currently investigating the competitive factor of NPT with respect to a non-preemptive clairvoyant scheduler. □

The above upper bound on competitive factor is <u>tight</u> 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 $\mathcal{A}$, 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) $\mathcal{A}$ 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 $\mathcal{A}$ is $(1/2 + 1/2k)$; it follows that as $k \to \infty$, the competitive factor of $\mathcal{A}$ becomes arbitrarily close to $1/2$.

The task generation process is such that initially, at some time t_c, $\mathcal{A}$ is offered a choice of 2 tasks: (i) A tight 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, whereas 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 $t'_c \stackrel{\text{def}}{=} t_c+2$, $T'_i \stackrel{\text{def}}{=}$ 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 $2k$. □

¿From the above discussion, it is clear that the competitive factor of $\mathcal{A}$ is not better than $(1/2 + 1/2k)$; it follows that as $k \to \infty$, the competitive factor of $\mathcal{A}$ becomes arbitrarily close to 1/2.

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 of 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. Also, denote the on-line algorithm by ONL. Then, consider the following task sequence: At $t=0$, a tight task T_1 which requires unit service arrives. Later, at $t=0.25$, another task T_2, which requires service of 0.25, arrives. Then, one of the following two cases occurs:

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, both requiring execution time of 0.5. Then, ONL can complete at most two of the four active tasks before t=1.5, as 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 T_4.

¿From the above cases, we see that, at best, ONL could complete 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. □

Note that the above theorem merely states an upper bound. We are currently working on establishing whether the above bound is tight.

As discussed before, equal-relative-deadlines is a special case of monotonic-absolute-deadlines. Therefore, from Theorem 4, it is clear that the SRPTF algorithm 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: In operating systems theory, it is well-known that a SRPTF schedule will always have completed the same or more number of tasks than any other schedule at any observation time [4]. This feature has also been utilized in the construction of the Best Effort Scheduler, developed in the CMU Archons project [7]. 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 moved on to studying the impact of overload when CC (completion count) 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 online algorithm can have a non-zero competitive factor. We then considered several special cases, representative of practical systems, wherein more positive results were obtained for the competitive factor. 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 online scheduler was equal to that of the optimal off-line scheduler, while for the Monotonic Absolute Deadlines case, the on-line scheduler could deliver only 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.

In our ongoing research work, we are working on developing efficient algorithms for handling the Equal Relative Deadlines situation. We are also investigating the performance of preemptive schedulers for Equal Execution Times task systems. Our long-term research goals include identifying the effect of task laxity on system performance and developing integrated schedulers that deliver good performance for both underload and overload conditions.

References

[1] 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. *Real-Time Systems*, 4:125–144, 1992. Also in *Proceedings of the 12th Real-Time Systems Symposium, San Antonio, Texas, December 1991.*

[2] 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.

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

[4] L. Schrage. A Proof of the Optimality of the Shortest Remaining Processing Time Discipline. In *Operations Research*, 16, pages 687–690, 1968.

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

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

[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] J. Haritsa, M. Carey, and M. Livny. Earliest-Deadline Scheduling for Real-Time Database Systems. In *Proceedings of the 12th IEEE Real-Time Systems Symposium*, San Antonio, Texas, December 1991.

[9] S. Baruah and J. Haritsa. On Improved Performance Guarantees Through the Use of Slack Times. Submitted for publication.

[10] 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.

[11] J. Dey, J. Kurose, D. Towsley, C. Krishna, and M. Girkar. Efficient On-Line Processor Scheduling for a Class of IRIS Real-Time Tasks. In *Proceedings of the 13th ACM SIGMETRICS Conference*, Santa Clara, California, pages 217–228, May 1993.

[12] 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.

[13] S. Baruah, N. Sharma, and J. Haritsa. On-Line Scheduling to Maximize Task Completions. *Technical Report No. XXX*, Dept. of Computer Science and Electrical Engineering, University of Vermont, September 1994.

[14] M. Garey and D. Johnson. *Computers and intractability: A guide to the theory of NP-completeness*, W. H. Freeman and Company, New York, 1979.

[15] C. Liu and J. Layland. Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment. In *Journal of the ACM*, Jan. 1973.

Appendix

A Earliest Deadline scheduling upon overload

Irrespective of the metric of performance used (EPU or CC), the Earliest Deadline algorithm is a very poor scheduling algorithm to use under conditions of overload. Consider, for example, a situation with two tasks T_1 and T_2, with $T_1.a = T_2.a = 0$, $T_1.e = 1$, $T_1.d = 1$, $T_2.e = n$, $T_2.d = n$, for some $n > 1$. Using Earliest Deadline, the *EPU* over the interval $[0, n)$ is $1/n$, which can be made arbitrarily small through an appropriate choice of n.

With respect to CC also, there exist situations wherein the overload performance of Earliest Deadline can be arbitrarily poor. Consider a workload of $n+1$ tasks, $T_0, T_1, \ldots, T_n$ with the following temporal characteristics: $T_i.a = 0$ for all i, $0 \leq i \leq n$; $T_0.e = T_0.d = 2n$; and, for $i = 1, .., n$, $T_i.e = 2, T_i.d = 2n+1$. If Earliest Deadline is used to schedule this task set, it is clear from inspection that only task T_0 will be completed. On the other hand, it is easily shown that a schedule which executes tasks T_1 through T_n in any order will complete all these tasks before their deadlines (but not be able to complete T_0). It follows that as $n \to \infty$, the CC performance of Earliest Deadline may be arbitrarily poor as compared to that of a clairvoyant scheduler.

$\square$