On-line scheduling to maximize task completions*

Sanjoy Baruah [§] *Jayant Haritsa* [†] *Nitin Sharma* [‡]

[§] Department of Computer Science
The University of North Carolina at Chapel Hill
baruah@cs.unc.edu

[†] Supercomputer Education and Research Centre
Indian Institute of Science
haritsa@serc.iisc.ernet.in

[‡] Department of Computer Science and Engineering
The University of Washington
nitin@cs.washington.edu

Abstract

In overloaded task systems, it is by definition not possible to complete all tasks by their deadlines. However it may still be desirable to maximize *the number of in-time task completions. The performance of on-line schedulers with respect to this metric is investigated here. It is shown that in general, an on-line algorithm may perform arbitrarily poorly as compared to clairvoyant (off-line) schedulers. This result holds for general task workloads where there are no constraints on task characteristics. For a variety of constrained workloads that are representative of many practical applications, however, on-line schedulers that* do *provide a guaranteed level of performance are presented.*

Keywords: *Real-time systems, overload scheduling, completion count, competitive ratio.*

*An extended abstract of this paper was presented at the IEEE Real-Time Systems Symposium, San Juan, Puerto Rico, Dec 1994.

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 poorly upon overload. Consider, for example, the classic Earliest Deadline scheduling algorithm [6], which is used extensively in real-time systems. This algorithm, which preemptively processes tasks in deadline order, is optimal for uniprocessor systems under non-overloaded conditions in the sense that it meets all deadlines whenever it is feasible to do so [3]; however, under conditions of overload, it has been experimentally observed to perform worse than even *random* scheduling [4].

System Model: We address here the problem of scheduling a number of tasks on a single *processor*. The processor is assumed to be fully preemptable, in that no penalty is incurred if a task currently executing on the processor is interrupted, and resumed at a later time. Each task T is completely characterized by three attributes: a *request time* $T.a$, an *execution requirement* $T.e$, and a *(relative) deadline* $T.d$, with the interpretation that task T, for successful completion, needs to be allocated a total of $T.e$ units of processor time during the interval $[T.a, T.a + T.d)$. Only tasks that fully complete execution by their deadlines are of any value to the user application, and no credit is obtained for partial execution of tasks. Given a set of such tasks to be scheduled on a single processor, and some metric to be optimized, the *scheduling problem* is to determine a schedule for the tasks on the processor such that the specified metric is optimized. If all the parameters of all the tasks are known prior to schedule generation time, then we have an *off-line scheduling problem*; otherwise, it is an *on-line scheduling problem*. (Throughout this paper we will assume that, in the on-line case, the system knows nothing about a task T until the instant $T.a$, when task T makes its request; at that instant, all its parameters are completely known. Furthermore, we assume that there is no *a priori* bound on the number of tasks.)

Overload Performance Metrics: 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 system spends on executing tasks that complete by their deadlines, while CC measures the number of tasks executed to completion during the overloaded interval. That is, a scheduling algorithm obtains a *value* $v(T)$ for each task T, where $v(T)$ depends upon the metric:

Effective Processor Utilization (EPU):

$$v(T) \stackrel{\text{def}}{=} \begin{cases} T.e, & \text{if } T \text{ is successfully scheduled} \\ 0, & \text{otherwise} \end{cases}$$

Completion Count (CC):

$$v(T) \stackrel{\text{def}}{=} \begin{cases} 1, & \text{if } T \text{ is successfully scheduled} \\ 0, & \text{otherwise} \end{cases}$$

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.

The **competitive ratio** of an on-line algorithm compares the performance of the on-line algorithm with that of an optimal off-line (equivalently, clairvoyant on-line) algorithm for solving the same problem. An on-line scheduling algorithm has a *competitive ratio* r, or is *r-competitive*, $r \geq 1$, iff no off-line algorithm can achieve a cumulative value more than r times that obtainable by the algorithm, over any finite set of tasks. An on-line scheduling algorithm is *competitive* if it is r-competitive for some finite r. An on-line algorithm is *optimal* if it is 1-competitive.

The off-line scheduling problem with respect to EPU — given a set of tasks, determine a schedule that maximizes the EPU — is easily seen to be NP-hard (transformation from bin-packing). *On-line* scheduling to maximize EPU is also quite well understood (see, for example, [2, 1]). It is known that no on-line scheduling algorithm can be more than 4-competitive with respect to this metric; furthermore, this bound is tight in that 4-competitive scheduling algorithms for this problem have been designed.

Scheduling to maximize completion count has been rather less studied. From a result of Lawler [5], it follows that off-line scheduling to maximize CC is somewhat easier than with respect to the EPU metric, and can be done in polynomial time ($O(n^5)$, where n is the number of tasks). Moore [7] presented a more efficient algorithm for the special case when all the tasks have equal request times ($T.a = 0$ for all T).

In this paper, we study the on-line scheduling problem with respect to the completion-count metric. Despite the fact that off-line scheduling for CC is easier than for EPU, it turns out that there are, in general, no competitive on-line scheduling algorithms for this problem. While this conclusion is rather disappointing from the perspective of developing good on-line overload schedulers, it turns out that very *general* task workloads where there are no constraints on task characteristics and in which tasks of widely different attributes coexist, are necessary to prove such negative performance bounds. In practice, most real-time applications generate task workloads that are more "similar" to each other, and hence more restricted in their characteristics. For a variety of such workloads, we obtain efficient competitive on-line schedulers that have low competitive ratios.

1 General Task Systems

When no restrictions at all are placed upon the kinds of tasks that may need to be scheduled, it turns out that on-line algorithms for maximizing CC may perform arbitrarily poorly when compared to off-line ones:

Theorem 1 No on-line scheduling algorithm is competitive with respect to the CC metric.

Proof: The proof process is best described as the interaction betweeen any on-line scheduling algorithm and a malicious adversary. Initially, the adversary generates a set of tasks and "observes" the behavior of the on-line algorithm on these tasks. Depending upon the behavior of the on-line algorithm, the adversary adds to the set of tasks such that, after a certain period of time, the on-line algorithm is once again in a state very similar to the initial state. It will, however, have executed no task to completion thus far, while an optimal scheduler would have completed one task by having made a choice different from that made by the on-line algorithm, and would be in the same state as the on-line algorithm. By repeating this argument η times, the adversary can ensure that an optimal scheduler would complete n tasks to the OL alg's one, for arbitrary η. The theorem follows by having $\eta \to \infty$.

More formally, let $\mathcal{A}$ be any on-line scheduling algorithm. The exact procedure for task generation employed by the adversary is detailed in

```
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 zero-slack task over [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 task-generation strategy of the adversary

Figure 1 (a "zero-slack" task T is one in which the execution requirement $T.e$ is equal to the relative deadline $T.d$). 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 $e \gg 2$ (Figure 2 (a)). 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 generated 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 (Figure 2 b). 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

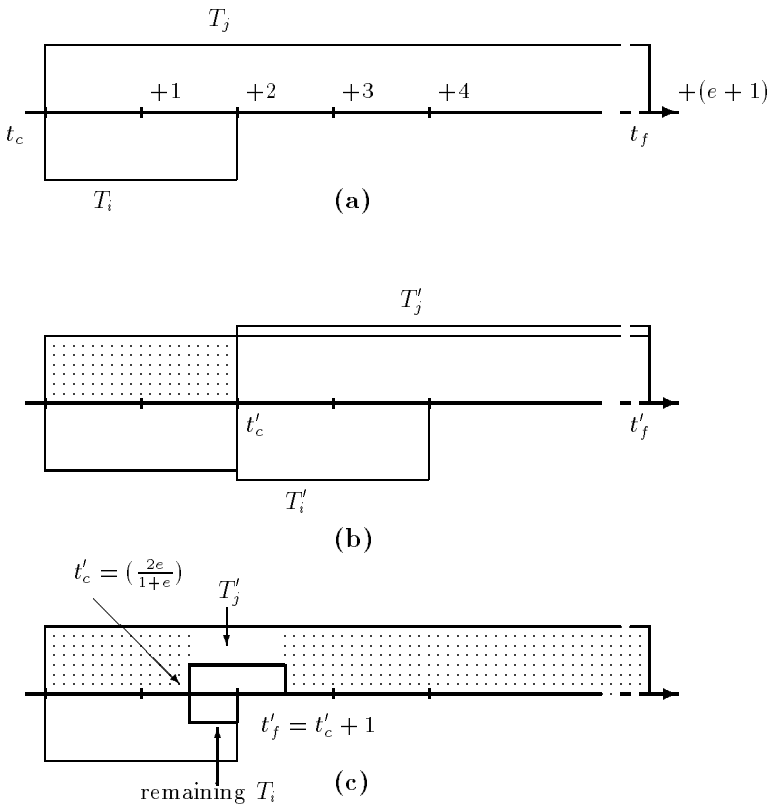

Figure 2: The adversary's strategy. (a) Tasks initially generated. (b) If the on-line algorithm chooses T_j. (c) If the on-line algorithm chooses T_i.

have completed one task – T_i – and 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. 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 $t'_c + 2/(1+e)$ (Figure 2 (c)). 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 $[t'_c + 1, t_c + (1 + e))$ and 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, for a competitive ratio of η. ■

While the conclusion that may be drawn from Theorem 1 — that an on-line algorithm may, in general, perform arbitrarily poorly when compared to an off-line one — is certainly severe, observe that the task set generated by the adversary is rather atypical of those found in most actual application systems. Specifically, to force a competitive ratio of η, it is necessary that the adversary choose $e \geq 2\eta$. For an on-line algorithm that causes this adversary to execute the "then" branch of the "if-then-else" statement on every iteration of the "for" loop (Figure 2), the task with the smallest relative deadline has relative deadline $2/(1 + e)^{(\eta-1)}$. The *deadline spread ratio (dsr)* of the set of tasks — the ratio of the longest relative deadline to the shortest — is thus $\geq (1 + 2\eta)^{\eta}/2$. (This means that, for example, the adversary must generate a task set with a dsr at least 13 to force a competitive ratio of 2, and a dsr of at least 172 to force a competitive ratio of 3.) Differently stated, if restricted to generating task sets with a dsr of at most k, the adversary of Figure 2 proves a lower bound of $\Omega\left(\frac{\log k}{\log \log k}\right)$ on the competitive ratio.

2 Special Cases

Theorem 1 means that on-line algorithms may in general perform arbitrarily poorly vis-a-vis off-line algorithms, when the completion count metric is the measure of performance. This is certainly disappointing from the perspective of developing good on-line overload schedulers; however, as we observed above, the proof of Theorem 1 required a particular kind of workload — one with an extremely large deadline spread ratio. In practice, most real-time applications generate task workloads that are rather more constrained in their attributes. For a variety of such restricted workloads,

we have been able to design on-line schedulers that *do* provide a guaranteed level of performance. We describe our results below.

2.1 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 MAD property corresponds, in a certain sense, to our intuitive notion of first-come first-served fairness, in that a task is not allowed to demand service by a deadline earlier than any of the tasks that preceded it.

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 (Theorem 2) 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 (Theorem 3) that we cannot hope to do better, that is, 2 is a lower bound on the competitive ratio of any on-line scheduling algorithm for MAD task systems.

Theorem 2 The SRPTF algorithm is 2-competitive on monotonic-absolute-deadline task systems.

Proof: Let τ be a set of tasks, and suppose that an optimal (off-line) scheduling algorithm can schedule n tasks in τ, $n \leq |\tau|$. Let $T_1, T_2, \ldots, T_n$ be the deadline-ordered sequence of these tasks. The set $\{T_1, \ldots, T_n\}$ constitutes a complete schedulable set of tasks; from the optimality of the Earliest Deadline First scheduling algorithm (EDF) for non-overload systems [3], it follows that there is a preemption-free schedule which executes each task $T_1, T_2, \ldots, T_n$ in deadline order. We can view this schedule as consisting of n disjoint intervals, $I_1, I_2, \ldots, I_n$, where I_i is the interval over which T_i was scheduled; $I_i = [T_i.t_s, T_i.t_f)$.

By definition, SRPTF schedules at every time instant the task with the shortest remaining execution time. For it to not execute T_i over $[T_i.t_s, T_i.t_f)$, therefore, it must be the case that either (1) SRPTF has already completed T_i by time $T_i.t_s$, or (2) at time $T_i.t_s$, SRPTF finds some task with a smaller remaining execution requirement than T_i; in this case either this smaller task, or some new task arriving during the interval with an even smaller execution requirement, will execute to completion by time $T_i.t_f$.

The interval I_i is defined to be "good" if SRPTF either completes some other task T_k not in $\{T_1, \ldots, T_n\}$ during this interval, or it completes T_i (at some time, although not necessarily during I_i). Conversely, the interval I_i is "bad" if it completes some task $T_k \neq T_i$, $T_k \in \{T_1, \ldots, T_n\}$, during I_i, and fails to complete T_i at any other time. (Observe that any interval I_i during which SRPTF completes no task is – paradoxically – good, since this means that I_i has already been completed by SRPTF.)

For every good interval, both SRPTF and the optimal schedule add to their completion counts (by completing either the same or different tasks); for every bad interval I_i, SRPTF loses exactly one task that the optimal schedule completes (the task T_i). The crucial observation is that each bad interval I_i "matches" with at least one good interval (interval I_k, corresponding to the task T_k that was completed during the bad interval I_i). Thus, at most $\lfloor n/2 \rfloor$ of the intervals may be bad. The theorem follows. ■

Theorem 3 2 is a lower bound on the competitive ratio of any on-line scheduling algorithm for scheduling monotonic-absolute-deadline task sets.

Proof Sketch: The proof is very similar to the proof of Theorem 1; hence we only provide a sketch here. The interested reader may fill in the details by essentially mimicking the proof of Theorem 1.

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 ratio of $\mathcal{A}$ is clearly 2. In the latter case, the competitive ratio of $\mathcal{A}$ is $2k/(k+1)$; it follows that as $k \to \infty$, the competitive ratio of $\mathcal{A}$ becomes arbitrarily close to 2.

The task generation process is such that 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 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 $t'_c \stackrel{\text{def}}{=} t_c + 2$, $T'_i \stackrel{\text{def}}{=}$ 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 $2k$. ■

2.2 Equal Request Times (ERT)

We next 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.)

Theorem 4 There are optimal (1-competitive) algorithms for scheduling equal-request-time task systems.

Proof: Moore [7], presented an optimal algorithm for *non*-preemptive scheduling to maximize task completions in equal-request-time task systems. However, when all tasks have the same request times, the problems of preemptive scheduling and non-preemptive scheduling on a uniprocessor are equivalent, in the sense that every set of tasks that can be scheduled preemptively can also be scheduled without preemption. (To see why, let τ be any set of tasks, and suppose that an optimal scheduling algorithm schedules a subset τ' of the tasks. The schedule generated by the Earliest Deadline First scheduling algorithm (EDF) [3] on τ' would meet all deadlines in τ'; further, since EDF schedules tasks in deadline order, this schedule would involve no preemption.) The algorithm of Moore is therefore an optimal algorithm for scheduling such systems. ■

2.3 Equal Execution Times (EET)

We now consider 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.

Lemma 1 Given a set τ of EET tasks such that an optimal off-line schedule can complete n of the tasks, any non-preemptive on-line algorithm that uses no inserted idle time will complete at least $\left\lceil \frac{n}{2} \right\rceil$ tasks.

Proof: Let NPT denote a generic non-preemptive scheduling algorithm that uses no inserted idle time. Our proof is by induction on n, the number of tasks completed by the optimal off-line scheduler on the overloaded set τ.

Without loss of generality, assume that all tasks $T \in \tau$ have $T.e = 1$, and that $\min_{T \in \tau}\{T.a\} = 0$.

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

Induction 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 $[0, x)$. 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 $\mathcal{R}$ 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 $\left\lceil \frac{k-2}{2} \right\rceil$ tasks in the interval $[1, x)$. Therefore, over the entire interval $[0, x)$, NPT will complete at least $1 + \left\lceil \frac{k-2}{2} \right\rceil$ tasks, that is, at least $\left\lceil \frac{k}{2} \right\rceil$ tasks.

The lemma follows. ■

We therefore conclude that any non-preemptive algorithm that uses no inserted idle time completes at least half of the number of tasks completed by an optimal off-line scheduler:

Theorem 5 There are 2-competitive on-line scheduling algorithms for scheduling equal-execution-time task systems.

This bound on competitive ratio is tight for the subset of on-line scheduling algorithms that are restricted to being non-preemptive. (Consider for example the set of tasks T_1 and T_2, with $T_1.a = 0, T_1.e = 1, T_1.d = 2$, $T_2.a = 0.5, T_2.e = 1, T_2.d = 1.0$, and $T_3.a = 0.9, T_3.e = 1, T_3.d = 1.9$. On this set of tasks, any non-preemptive algorithm can only complete one task, while an optimal preemptive scheduler would complete both.)

2.4 Equal Relative Deadlines (ERD)

For systems where all tasks are *a priori* known to have the same relative deadline, the following result holds:

Theorem 6 No on-line algorithm can be better than 3/2-competitive for equal-relative-deadline task systems.

Proof: Without loss of generality, assume that the relative deadline of all tasks is 1 (i.e., $T.d = 1$ for all T). Let ONL denote any 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, 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 $T_3.e = T_4.e = 0.5$. 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. The off-line scheduler, on the other hand, would execute T_2 over $[0.25, 0.5)$, and then complete T_3 and T_4.

At best, therefore, ONL completes 2 tasks whereas the off-line algorithm can complete 3. The theorem follows. ■

The above theorem establishes a lower bound on the competitive ratio. We are currently working on determining whether this bound is tight. Since equal-relative-deadline task systems also satisfy the monotonic-absolute-deadline property (Section 2.1), it follows by Theorem 2 that the SRPTF algorithm is 2-competitive for ERD task systems.

2.5 Equal Absolute Deadlines (EAD)

A task system τ has equal absolute deadlines if it is guaranteed that

$$\forall\ T_i, T_j \in \tau :: \quad T_i.a + T_i.d = T_j.a + T_j.d\ .$$

Theorem 7 The SRPTF Algorithm is optimal for equal absolute deadline task systems.

Proof: It has been shown [8] 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 this deadline. ■

3 Conclusions

Effective processor utilization (EPU) and *completion count (CC)* are two very different metrics for measuring the performance of scheduling algorithms under conditions of overload. The EPU metric has been well studied: it is known that the off-line problem is NP-hard, and 4-competitive

algorithms have been designed for the on-line problem. Not quite as much attention has been paid to the CC metric: a relatively recent result of Lawler [5] yields a polynomial-time off-line scheduling algorithm as a corollary. In this paper, we have studied the on-line scheduling problem with respect to this metric. Our major result is negative — in contrast to EPU, for which 4-competitive on-line schedulers exist, it turns out that there can be no competitive on-line algorithms that maximize CC. However, when task systems are restricted in some manner, competitive scheduling often becomes possible: we have been able to obtain efficient on-line scheduling algorithms with low competitive ratios for a wide variety of restricted task systems.

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*, pages 100–110, 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] J. Haritsa, M. Carey, and M. Livny. Earliest-deadline scheduling for real-time database systems. In *Proceedings of the Real-Time Systems Symposium*, 1991.

[5] E. L. Lawler. A dynamic programming algorithm for preemptive scheduling on a single machine to minimize the number of late jobs. *Annals of Operations Research*, 26:125–133, 1990.

[6] C. Liu and J. Layland. Scheduling algorithms for multiprogramming in a hard real-time environment. *Journal of the ACM*, 20(1):46–61, 1973.

[7] J. M. Moore. An n job, one machine sequencing algorithm for minimizing the number of late jobs. *Management Science*, 15:102–109, 1968.

[8] L Schrage. A proof of the optimality of the shortest remaining processing time discipline. *Operations Research*, 66:687–690, 1968.