On-line Scheduling in the Presence of Overload [1]

S. Baruah [†] G. Koren [‡] B. Mishra [‡] A. Raghunathan [††] L. Rosier [†] D. Shasha [‡]

Dedicated to the memory of our friend and colleague, Dr. Louis Rosier

ABSTRACT

We consider a problem from the domain of real-time systems — the preemptive scheduling of sporadic tasks on a uniprocessor. A task may arrive at any time, and is characterized by a value *that reflects its importance, an* execution time *that is the amount of processor time needed to completely execute the task, and a* deadline *by which the task is to complete execution. The goal is to maximize the sum of the values of the completed tasks. The off-line version of this problem — where the timing constraints of all tasks are given as input — is NP-hard. With respect to the more realistic on-line setting (common to real-time systems) where nothing is known about a task until it arrives (at which instant all relevant parameters are known), we design an on-line scheduling algorithm that achieves optimal performance when the system is underloaded, and provides a non-trivial performance guarantee when the system is overloaded. To our knowledge, this is the first algorithm that achieves any such guarantee. We implement our algorithm using simple data structures to run at a cost of $O(\log n)$ time per task, where n bounds the number of tasks in the system at any instant. Furthermore, we derive upper bounds on the best performance guarantee obtainable by any on-line algorithm in a variety of settings. Our experimental data indicate that our algorithm is comparable to a widely used heuristic, Locke's* best effort scheduler, *in most cases, while guaranteeing performance even for the pathological cases that cripple Locke's heuristic.*

1 Introduction

Suppose you are building software to control processes within a nuclear power plant. Periodically, your control system will need to monitor the various parameters of the power plant, such as temperature and pressure, and take appropriate control actions, such as increasing or decreasing the amount of fuel, water, etc. These control actions or *tasks* need to be completed by *deadlines*, so that the system tracks the physical processes accurately. Typically, each of these tasks can be scheduled to completion by its deadline. Occasionally, however, the power plant may be overheating, requiring an overwhelming number of control actions to be performed. In such a situation, the system is likely to be overloaded and as such it will be impossible to schedule every such task to completion by its deadline. If your system responds to such an eventuality by simply stopping, or by executing processes of low importance, it has ceased to be useful in controlling the emergency; instead, it has become part of the emergency.

A software system such as the one above is called a *real-time system*, since tasks have deadlines by which they must complete. Real-time systems arise in applications such as aircraft and spacecraft control, robotics and factory automation. Real-time systems with weaker requirements also arise in program trading for financial markets and telephone networks.

A system is *underloaded* if there exists a schedule that will meet the deadline of every task. Scheduling underloaded systems is a well-studied topic, and several on-line algorithms have been proposed for the optimal scheduling of these systems on a uniprocessor. Examples of such algorithms include *earliest-deadline-first* (D) and *smallest-slack-time* (SL). However, none of the proposed algorithms make performance guarantees during times when the system is overloaded. In fact, it has been experimentally demonstrated that these algorithms perform quite poorly when the system is overloaded [6].

Practical systems are prone to intermittent overloading caused by a cascading of exceptional situations, often corresponding to emergencies. A good on-line scheduling algorithm should not only be optimal under normal circumstances, but also respond appropriately to emergency situations. Given the importance of scheduling overloaded systems, researchers and designers of real-time systems have devised intelligent on-line heuristics to handle overloaded situations [1,6,8].

[1]Supported by ONR grants #N00014-89-J-1913, #N00014-89-J-3042, #N00014-90-J-1110, and NSF grants #CMU-406349-55586, #CCR-9002819, #IRI-89-01699, #IRI-9003986.

[†] The University of Texas at Austin, Austin, TX 78712.

[‡] Courant Institute, New York University, New York, NY 10012.

[††] University of California at Davis, CA 95616.

The most successful of these heuristics was proposed by Locke as part of the CMU Archons project [6]. Roughly, Locke's ***best effort*** (BE) heuristic works as follows. Each task in Locke's model has a ***value***; the value of a task measures its importance. If the system is underloaded then BE operates like algorithm D; if an overloaded condition is detected then BE randomly abandons some of the less important tasks in an attempt to bring the system back to an underloaded state. The complete algorithm is much more sophisticated, since it provides a policy-mechanism-separation by letting the user influence how some tasks are abandoned. While Locke's heuristic is widely used and has been shown to perform well in practice, it offers no performance guarantee. In fact, no performance guarantee had been provided with respect to any such on-line algorithm when we achieved our results.

Our task model is based on the model defined by Locke [6]. Tasks may enter the system at any time. Nothing is known about a task until it arrives. When a task is released, its computation time, value, and deadline are known precisely. The value of a task is obtained provided it completes by its deadline; once its deadline passes, there is no value to completing the task. At any time, a task may be preempted in favor of another task at no cost.

To facilitate the presentation of our results, we define the ***value density*** of a task as the ratio of its value to its computation time. The ***importance ratio***, k, of a set of tasks is the ratio of the largest value density to the smallest value density. We will always take the smallest density to be one. When the importance ratio is 1, tasks are said to have ***uniform value density***.

The main contributions of this paper are listed below.

- We formalize the notion of when a system is overloaded and when it is underloaded (Section 2).
- We devise an on-line scheduling algorithm called D*, that provides the following performance guarantee (Section 3). Informally[2] speaking, D* achieves optimal performance while the system is underloaded and achieves a value that is at least 1/5 times the *length* of the overloaded period while the system is overloaded. To our knowledge, this is the first time that such a nontrivial performance guarantee has been obtained. D* is simple, efficient, and easy to implement. It can be implemented using balanced search trees, and thus runs at a cost of $O(\log n)$ time per task, where n bounds the number of tasks in the system at any instant. The analysis of the algorithm's performance guarantee is not straightforward.
- A performance guarantee of 20% may not seem extremely good, but in Section 4, we show that one cannot do much better. We prove that, during an overloaded interval, no on-line scheduling algorithm can guarantee a value greater than 1/4 times the length of the overloaded interval. The bounds of Section 4 apply in an even more general setting. Consider as a comparison vehicle clairvoyant on-line scheduling algorithms that know the arrival time, value, execution time, and deadline of all future task requests. One can then quantify the performance of on-line algorithms as compared to their clairvoyant counterparts. As in [4,9], we say that an on-line algorithm has ***competitive factor*** r, $0 \leq r \leq 1$, if and only if it is guaranteed to achieve a cumulative value at least r times the cumulative value achievable by the cleverest clairvoyant algorithm on ***any*** sequence of task requests. We prove that no on-line scheduling algorithm can have a competitive factor greater than $1/[(1+\sqrt{k})^2]$.[3] We generalize this result by considering environments where there is an upper bound on the "amount" of overloading allowed within an interval, i.e., a bound on the ***loading factor***[4] within an interval. In particular, given uniform value density, whenever the loading factor does not exceed one the competitive factor limitation is 1 — obviously. As the loading factor exceeds one, we show that the competitive factor limitation immediately falls to 0.385, and as the loading factor increases from one to two, we show that the competitive factor limitation falls from 0.385 to 0.250. For loading factors beyond two the competitive factor limitation remains at 0.250.
- Our experimental data indicates that D* is comparable to Locke's BE or a wide class of environments, while guaranteeing the bound even for the pathological cases that would cripple BE (Section 5 and Appendix C).
- Note added in proof: At the time of the submission, D* was the only algorithm that could make any performance guarantee. However, its competitive factor does not match the upper bound presented in this paper, suggesting room for further improvement. Two recent developments in this direction are worth mentioning. Wang and Mao at the University of Massachusetts at Amherst explored complexity questions and subsequently in May 1991, have proposed an algorithm that is 1/4-competitive, although it is non-optimal in the underloaded case. In June 1991, we obtained an algo-

[2] The results are more precisely stated in subsequent sections.

[3] When $k = 1$, the competitive factor is 1/4.

[4] This will be formally defined in Section 4.

rithm that is 1/4-competitive, and optimal in the underloaded case. Our result will be included in the full paper.

While most proofs are omitted, some of the more important proofs are presented in the appendix.

2 Preliminaries

Usually, real-time systems are divided into two classes: ***hard*** real-time systems and ***soft*** real-time systems (see, for instance, Mok's dissertation [7]). In a hard real-time system, deadlines must ***absolutely*** be met or the system will be considered to have failed; in a soft real-time system deadlines may occasionally be missed with only a degradation in performance. In this paper, we consider a special case of a soft real-time system, called a ***firm*** real-time system, in which there is no value gained for a task that completes after its deadline is missed, but there is no catastrophe either.

As mentioned in the previous section, tasks may enter the system at any time. Their computation times and deadlines are known exactly at their time of arrival. Nothing is known about a task before it appears. The goal is to make ***on-line*** scheduling decisions that will maximize the overall value of the resulting computation, even in the presence of overload[5]. We note that obtaining a scheduler that maximizes the overall value is a difficult problem, even for a clairvoyant scheduler. The problem of finding the maximum achievable value for such a scheduler can be shown to be reducible from the knapsack problem[3], and hence NP-hard.

2.1 Overload and Underload

We now need to define what we mean by an overloaded or an underloaded system. For instance, imagine a system that has so many tasks entering it at time t that it cannot possibly schedule all of them to completion. If this system throws out most of these tasks, it may be possible for the system to meet the deadline of every new task that enters the system after time t. Is this system really underloaded at t? Intuitively, it is not. When does the system return to the underloaded state?

We propose a definition that will not throw out old tasks whose deadlines are yet to pass. Suppose we run D (i.e., earliest-deadline-first) without throwing out tasks that cannot meet their deadlines until their deadlines pass. Intuitively, a transition to overload will occur when executing one task will cause another one to miss its deadline. A transition to underload occurs when this naive D scheduler would cause tasks to meet their deadlines again. This is formalized below.

During the execution of algorithm D, we take a snapshot at time t. At time t, let T_c be the currently running task, if there is one. Let the sequence of tasks T_{i_1}, T_{i_2}, ..., T_{i_n} be yet-to-be scheduled tasks (i.e., tasks with release times earlier than the current time, with deadlines after the current time and that have not yet completed) ordered by their respective deadlines.

We can now calculate the times at which D would schedule each of the above tasks, assuming that no new task enters the system. Each time a new task, T_e, enters the system, the currently running task, is preempted, and is referred to as T_{i_k}, $k = 1$ or $k = 2$, depending on whether the new task has deadline before or after the deadline of T_e. All start times are recalculated by the equations shown below, reflecting the newly expanded task set.

$$\begin{aligned} S(T_{i_1}) &= t, \\ S(T_{i_j}) &= F(T_{i_{j-1}}) \quad (j > 1), \end{aligned}$$

where, for $j \geq 1$, $F(T_{i_j})$ is

$$\min\{S(T_{i_j}) + \mathit{Computation\text{-}Time}(T_{i_j}), \mathit{Deadline}(T_{i_j})\}.$$

$\mathit{Computation\text{-}Time}(T_{i_j})$ refers to the remaining computation time of the task that is now referred to as T_{i_j}. We say that a task has *positive lateness* if at the time of its deadline, its *Computation-Time* is strictly positive. In other words, a task that did not complete by its deadline in the schedule of the algorithm D has positive lateness[6]. We emphasize that the start and finish times of a task, and hence its lateness, are dynamic quantities that change as new tasks enter the system. Note that if, for all j, $F(T_{i_j}) \leq \mathit{Deadline}(T_{i_j})$, then all the deadlines are met and the system is underloaded. However, if at any point in time, a task has a positive lateness, D will never allow this task to complete by its deadline, irrespective of whether or not new tasks enter the system.

Let us introduce some convenient notation. When a task T_e enters the system, the quantity d_e denotes $\mathit{Deadline}(T_e)$ and the quantity c_e denotes $\mathit{Computation\text{-}Time}(T_e)$. Later, when new tasks enter the system, $\mathit{Computation\text{-}Time}(T_e)$ is the remaining

[5]Note that maximizing the value may not minimize the number of missed deadlines, since many low-valued tasks may be abandoned in favor of a single high-valued task.

[6]A related concept is *slack time*, which roughly means the amount of time before its deadline that a task will complete, if the current schedule were to continue. Thus, a task with no slack time will complete at its deadline.

computation time of T_e; however, c_e continues to denote the original length of computation of T_e. Let us also call the interval starting at $d_e - c_e$ and ending at d_e the *Latest start interval* of T_e and denote it by δ_e. Also, for reasons that will become clear later, let us denote the interval starting at $d_e - 3c_e$ and ending at $d_e + 2c_e$ by Δ_e.

A *history* of the system, H, is a set of tasks, and for each task, its *release time, computation time* and *deadline*. Given such a history H, many algorithms may attempt to schedule it. In particular, we focus attention on the schedule of the algorithm D, as described above.

The *overloaded time period OL* is defined by:

$$OL = \bigcup \left\{ \delta_i \right\}_{T_i \text{ has positive lateness}}.$$

OL is not necessarily connected. We will call a maximal connected component of OL an *overloaded interval*. The *underloaded time period* is the complement of the overloaded time period. Furthermore, we say that *a task is in the overloaded time period* if its deadline is in an overloaded interval. Otherwise, we say that *the task is in the underloaded time period*.

3 D* and its Analysis

The algorithm D* behaves like the algorithm D during the underloaded period. However, when the system is overloaded, the algorithm D* will abandon the currently running task in favor of another task *if and only if* the value to be obtained by running that task is higher than the cumulative value of all the abandoned tasks[7] since the last time a task successfully completed its execution. If the new task has "too little" value, D* will simply *abandon* the new task, without including its value in the cumulative total.

The algorithm D* requires two data structures, called D-Structure (Deadline Structure) and L-Structure (Latest-start-time Structure). D-Structure maintains the tasks in their deadline order. L-Structure maintains the tasks in the order of their *latest start times*, denoted LST, where

$$LST(T) = \mathit{Deadline}(T) - \mathit{Computation\text{-}Time}(T).$$

Recall that *Computation-Time*(T) refers to the remaining computation time of T. Intuitively, $LST(T)$ is the latest time when T can be scheduled and still complete by its deadline.

These data structures are implemented as balanced search trees, e.g. 2-3 trees. The D- and L-Structures support Insert, Delete and Min operations, each taking $O(\log n)$ time for a structure with n tasks. The structures share their leaf nodes which represent tasks. We will show later that our algorithm ensures that each task causes the structure to be accessed a constant number of times. Thus if n bounds the number of unscheduled tasks in the system at any instant then each task incurs only an $O(\log n)$ cost.

In the algorithm described below, there are three kinds of *events* (each causing an associated interrupt) considered: TaskCompletion, TaskRelease, and TimerInterrupt, in order of their priorities[8].

- The system event TaskCompletion occurs when a task successfully completes (and obtains its full value). In that case, the task with the earliest deadline (i.e., the minimal element in the D-Structure) is removed from both the structures and is then executed.
- The external event TaskRelease occurs when a task is released into the system.
- The system event TimerInterrupt occurs when the wall-clock time equals the minimal value in the L-Structure. This interrupt indicates an overloaded situation, and will not occur in the underloaded case.

Also, in the algorithm, the variable Current-Task denotes the task that is currently executing. The accumulator Preempted-Value represents the sum of the values of all tasks preempted thus far since the termination of the last task that completed execution.

For purposes of later comparison, the *earliest-deadline-first* algorithm (D) consists of the events TaskCompletion and TaskRelease with the small (but significant) change that the last "else if" condition in TaskRelease is replaced by a simple else. The other difference is that TimerInterrupt is never needed in D.

[7] We refine this notion further below.

[8] Thus if several interrupts happen simultaneously, then the TaskCompletion interrupt is handled before the TaskRelease interrupt, which in turn is handled before TimerInterrupt. It may happen that a TaskCompletion event will remove the condition for a lower priority interrupt, e.g., by removing a task from a data structure.

```
On-Line Scheduling Algorithm, D*:
D-Structure := ∅;   L-Structure := ∅;   Preempted-Value := 0;

loop
    (* In the algorithm, the statement "Execute Current-Task" schedules the task to run; however,
    this task may be interrupted before it finishes. *)

    TaskCompletion:
        Preempted-Value := 0;
        if D-Structure is non-empty  then
        (* Task T has minimum deadline in D-Structure. *)
            Remove T from the D- and L-Structures;
            Current-Task := T;  Execute Current-Task;
        end{if }
    end{TaskCompletion}

    TaskRelease:
        (* Task T is released into the system. *)
        if there is no Current-Task  then
            Current-Task := T;  Execute Current-Task;
        else if Deadline(Current-Task) < Deadline (T)  then
            Insert T into the D- and L-Structures;
        else if Preempted-Value = 0  then
            (* Note that in the other case, if T has an earlier deadline than that of the current-
            task and the preempted-value is positive, i.e., the current-task was scheduled by
            TimerInterrupt, the new task is discarded(note that T has less value than current-
            task. *)
            Preempt Current-Task;
            Insert Current-Task into the D- and L-Structures by its Deadline and
                LST (= Deadline − remaining Computation-Time), respectively;
            Current-Task := T;   Execute Current-Task;
        end{if }
    end{TaskRelease}

    TimerInterrupt:
        (* Task T has minimum LST in L-Structure. There must be a Current-Task, when this
        interrupt arrives, since, otherwise, task T must have been already scheduled. *)
        Remove T from the D- and L-Structures;
        if Current-Task has slack time  then
            Preempt Current-Task;
            Insert Current-Task into the D- and L-Structures by its Deadline and
                LST (= Deadline − remaining Computation-Time), respectively;
            Current-Task := T;   Execute Current-Task;
        else if value(T) > Preempted-Value + value(Current-Task)  then
            (* The Current-Task is overthrown *)
            Preempted-Value := Preempted-Value + value(Current-Task);
            Preempt Current-Task;
            Current-Task := T;   Execute Current-Task;
        end{if }
        (* If task T has too little value then T is rejected. In this case, Preempted-Value is not
        changed. *)
    end{TimerInterrupt}
end{loop }
end{D*}.
```

Algorithm 1: D*: ON-LINE SCHEDULING ALGORITHM.

3.1 The Correctness and Bounds

Let T_i be a task that did not complete its execution by its deadline. Hence a decision to abandon T_i was made at some point in time, henceforth called *Abandoned* (T_i). The algorithm abandons a task only in the TimerInterrupt and the TaskRelease routines. If T_i was abandoned in the TimerInterrupt routine, then either T_i was the Current-Task with no slack time or a task whose LST had just occurred but whose value was not big enough to preempt the Current-Task. In the first case, we say that T_i was **overthrown**, and in the second case, we say that it was **rejected**. If T_i was abandoned in TaskRelease we say that T_i was **discarded**. Exactly one of the above must apply to T_i. If T_i was discarded or rejected and the current task at *Abandoned* (T_i) was T_j or if T_i was overthrown by a TimerInterrupt generated by T_j we will say that T_i was *abandoned due to* T_j.

In what follows assume that a history, H, is given, and that at least one task has positive lateness. Let OL be the overloaded period and let $[s,t]$ be the first overloaded interval. We know that

$$[s,t] = \bigcup_{i \in V} \delta_i,$$

where V corresponds to the set of tasks, each of which has positive lateness and deadline in $[s,t]$.

Lemma 3.1 *There is no idle time between s and t when algorithm D schedules H.*

Lemma 3.2 *All tasks scheduled by D between s and t have deadlines at or before t.*

Now, let D* schedule the same history H. D* operates exactly like D till the first TimerInterrupt . (Recall that the TaskRelease of D* is different from that of D only if the preempted value is not zero, indicating that a TimerInterrupt event had previously occurred). We now assert that the first TimerInterrupt must occur in $[s,t]$. If it occurs before s, then the task that caused the TimerInterrupt event must have positive lateness. By the definition of the first overloaded interval, the *Latest start interval* of this task would be in the overloaded interval, meaning that the first overloaded interval started before s, a contradiction. It cannot occur after t, for otherwise all tasks in V will be executed to completion in D. This indicated that none of them has positive lateness, also a contradiction.

Lemma 3.3 *All tasks with deadlines before s complete their execution (in D*).*

Let us look at the schedule of D* in $[s,t]$. D* might have abandoned some tasks but we assert that all such tasks have deadlines prior to t. We will need the following definition.

Definition 3.1 Suppose that both D and D* schedule the same history H and let T be a task in H. We will say that at time t, T's position in D* is ***no worse*** than its position in *D* ***if and only if*** the following holds. The remaining computation time at time t, under D*, is no larger than the remaining computation time of T at time t under D.[9] under D

Let T be a task with deadline after t. Since D and D* behave identically up to s, we know that T has the same share of processor time in D and in D* up to s. By Lemma 3.2, we also know that in D's schedule, T does not have any execution time during [s,t]. It is, of course, possible that T might have some execution time in the schedule of D*. We conclude that if T is not abandoned by D* in [s,t] then its position in D* is no worse. Is it possible that T is abandoned? The following lemma shows that the answer is no.

Lemma 3.4 *In the schedule of D*, if at some point x in [s,t] a task T has no slack time then its deadline is in [s,t].*

The previous lemma implies that a task T with deadline after t cannot be rejected or overthrown in [s,t] since both imply that T has no slack time when abandoned. Is it possible that T is discarded? When a task is discarded , the preempted value is greater than 0, hence the Current-Task reached its LST in [s,t] and T had an earlier deadline than Current-Task. Hence (again by lemma 3.4) T's deadline is before t. We conclude that no task with deadline after t is abandoned by D* in [s,t].

Imagine that D* schedules the system up to time t and at time t the control is handed over to algorithm D. Call this history H'. The history H' for D is different from history H; t is the start time with every task currently in the system having computation time equal to the "remaining" computation time in the schedule of D*.

Let OL' be the overloaded period of the schedule of D for H'. We assert that $OL' \subseteq OL$ (i.e. the overloaded period resulting from using D* until time t is contained in the overloaded period resulting from using D until time t). This holds because at time t the task set for

[9] **In particular, the above holds, if T already completed at time t under D*.**

D* is a subset [10] of the task set for D and for each task the remaining computation time in D* is less than or equal to the remaining computation time in D. Let $[s',t']$ be the first overloaded interval in D's schedule of H'. All previous claims will hold with $[s',t']$ playing the role of [s,t]. The process can now go on inductively, thus yielding the following lemma.

Lemma 3.5 *All tasks in the underloaded period are executed to completion in D^*.*

In all that follows let C be an overloaded interval, and let $T_1, T_2 \ldots T_q$ be the tasks with deadlines in C. The definition of the overloaded period implies that $C \subseteq \cup_{i=1}^{q} \delta_i$. Also, let $S \subseteq \{1,2,\ldots q\}$ be the set of indices of those tasks that successfully complete execution.

We now state the following two lemmas without proofs. In Lemma 3.6, a task executes *nontrivially* if it executes for some strictly positive amount of time[11].

Lemma 3.6 *Suppose T_i was abandoned. Let T be the the first task to complete after Abandoned (T_i). All tasks that execute nontrivially from Abandoned (T_i) to the completion of T have no slack time at the time that they are scheduled for execution.*

Lemma 3.7 *Suppose T_i was abandoned. Let T be the the first task that completed execution after Abandoned (T_i). Then T has its deadline in C hence there is an element $k \in S$ for which $T = T_k$.*

We now come to the following crucial technical lemma, whose proof is in Appendix A.

Lemma 3.8 *Suppose T_i was abandoned in C. Let T_k be the the first task that completed execution after Abandoned (T_i) then [12] $\delta_i \subseteq \Delta_k$.*

PROOF.
Please see Appendix A. □

We are now ready to state our performance guarantee for D*.

[10]We just proved that no task with deadline after t is abandoned in [s,t]. It is possible, however, that some tasks with deadline after t already executed to completion under D*.

[11]A task can execute *trivially* for 0 time units if a TimerInterrupt event preempts the task as soon as it is scheduled.

[12]Recall that for a task T_i, d_i denotes *Deadline*(T_i) and c_i denotes the initial *Computation-Time*(T_i). Also, δ_i denotes the interval starting at $d_i - c_i$ and ending at d_i, Δ_i is the interval between $d_i - 3c_i$ and $d_i + 2c_i$.

Theorem 3.9 *The algorithm D^* schedules to completion every task whose deadline is in the underloaded time period. In addition, D^* obtains a value that is at least one-fifth the length of the overloaded period from all tasks in the overloaded period.*

PROOF.
The first part of the theorem deals with the underloaded case, and is covered by Lemma 3.5.

Let C be an overloaded interval, and let $T_1, T_2 \ldots T_q$ be the tasks with deadlines in C. Let $S \subseteq \{1,2,\ldots q\}$ be the set of indices of those tasks that successfully complete execution.

We know that $C \subseteq \cup_{i=1}^{i=q} \delta_i$. Lemma 3.7 assert that for every $j \in \{1,2,\ldots q\}$ there is a $k \in S$ such that $\delta_j \subseteq \Delta_k$.

The value V obtained by D* then satisfies[13] the following:

$$V = \sum_{j \in S} |\delta_j| = \frac{1}{5} \times \sum_{j \in S} |\Delta_j| \geq \frac{1}{5} \times |\cup_{j \in S} \Delta_j|$$

$$\geq \frac{1}{5} \times |\cup_{i=1}^{q} \delta_i| \geq \frac{1}{5} \times |C|.$$

This is the desired result. □

3.2 The Complexity

Theorem 3.10 *If n bounds the number of unscheduled tasks in the system at any instant then each task incurs only an $O(\log n)$ cost.*

4 Upper Bounds on the Competitive Factor

In this section, we examine the limitations to the power of any on-line algorithm under a variety of settings. The proofs of some lemmas in this section are contained in Appendix B.

Lemma 4.1 *There does not exist an on-line scheduling algorithm that obtains, for tasks in the overloaded period, a value greater than 1/4 times the length of the overloaded period. Furthermore, given uniform value density, there does not exist an on-line scheduling algorithm with a competitive factor greater than 1/4.*

In deriving the bound above, an adversary argument was used wherein the malevolent adversary was allowed to introduce new tasks at will; thus the amount of overloading is unbounded. A natural question to ask at this

[13]$|\cdot|$ means the length of the interval.

stage would be: is the *amount* of overloading permitted by the environment related to the best competitive factor that may be guaranteed by an on-line algorithm? To answer this question, let us quantify the notion of overloading.

We say a sporadic real-time environment has a *loading factor b* iff it is guaranteed that there will be no interval of time $[t_x, t_y)$ such that the sums of the execution-times of all task-requests making requests and having deadlines within this interval is greater than $b \cdot (t_y - t_x)$. It is easily observable that a system can never become overloaded if it has a loading factor no greater than 1.

The on-line algorithm knows *a priori* what the loading factor for the environment is, and may use this information in making on-line scheduling decisions. Consider, as an example, on-line scheduling in an environment which is known to have a loading factor no larger than 1 (i.e., a non-overloaded environment). Dertouzos [2] has shown that the algorithm D is optimal in such an environment. D is, therefore, an on-line scheduler with a competitive factor of 1 in sporadic real-time environments with a loading factor no larger than 1. At the other extreme, Lemma 4.1 proves that no on-line scheduler can offer a competitive factor larger than 0.250 in environments where the loading factor may be arbitrarily large[14].

The following lemma quantifies the relationship between the loading factor and the upper bound on the competitive factor of an on-line algorithm in environments where the loading factor is between 1 and 2.

Lemma 4.2 *Given uniform value density, no on-line scheduling algorithm operating in an environment with a loading factor b, $1 < b \leq 2$, can have a competitive factor greater than p, where p satisfies*

$$4(1-(b-1)p)^3 = 27p^2$$

The next question to address is: how does the importance ratio k affect the best possible guarantee? Unfortunately, the guarantees that can be made by an on-line algorithm in such an environment is even less than 0.250, as the following lemma states:

Lemma 4.3 *Let $i_{\max}$ ($i_{\min}$) be the largest (smallest) value density a task may have in an environment. No on-line scheduling algorithm operating in this environment can have a competitive factor greater than*

$$\frac{1}{(1+\sqrt{k})^2}$$

where $k = (i_{\max}/i_{min})$ is the importance ratio *of the environment.*

[14] Actually, the proof of Lemma 4.1 requires a loading factor of $2+\epsilon$, where ϵ is an arbitrary small positive number.

A couple of points worth noting:

1. The quantity $1/(1+\sqrt{k})^2$ decreases rather rapidly as the importance ratio k increases. For $k=1$, this value is .250; for $k=2$, it falls to .172; for $k=3$, it is .134; for $k=5$, it is .095; and for $k=10$, it is as low as .028.
2. From Lemma 4.3 it follows that, in an environment where the importance ratio is not *a priori* bounded from above (i.e., $i_{\min}$ and $i_{\max}$ are not *a priori* bounded from below and above respectively) no on-line algorithm can have a competitive factor greater than 0.

The proof of Lemma 4.3 above made no assumptions regarding the loading factor of the environment. An analysis similar to the one made for Lemma 4.1 would reveal that the loading factor in this case would need to be greater than 2. The following lemma relates the loading factor and the importance ratio of an environment to an upper bound on the competitive factor of any on-line scheduler operating in this environment.

Lemma 4.4 *Consider an environment with a loading factor b, $1 < b \leq 2$, and an importance ratio k. Let $q = k \cdot (b-1)$. If $q \geq 1$, then no on-line scheduling algorithm can have a competitive factor greater than $1/(1+\sqrt{q})^2$, whereas if $q < 1$, no on-line scheduling algorithm can have a competitive factor greater than p, where p satisfies $4(1-qp)^3 = 27p^2$.*

For environments with $1 < b \leq 2$, Lemma 4.4 provides an upper bound for the competitive factor of on-line algorithms, while Lemma 4.3 provides an upper bound for the competitive factor of on-line algorithms in environments where $b \geq 2$. These results are summarized in the following theorem:

Theorem 4.5 *Let k be the importance ratio of an environment, and b its loading factor. Let $q = k \cdot (b-1)$. For $b \leq 1$, there exist on-line schedulers which have a competitive factor of* 1.0. *For $1 < b \leq 2$, Lemma 4.4 defines an upper bound on the competitive factor of any on-line scheduler, and for $b > 2$, no on-line scheduler can have a competitive factor greater than p, where p satisfies $4(1-kp)^3 = 27p^2$.*

5 Performance Results

We have implemented our algorithm and Locke's Best Effort algorithm and have run simulations to accumulate empirical data. These results are further described in Appendix C. We would like to thank Mr. Robert Kagel for implementing the two algorithms and running the simulation studies that produced these experimental results.

References

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

[2] M. Dertouzos. Control Robotics: the procedural control of physical processes. In *Proc. IFIP congress*, pages 807–813, 1974.

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

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

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

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

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

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

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

Appendix

A. Proof of Lemma 3.8

First let us show that $d_i - c_i \geq d_k - 3c_k$.

Let τ be the Preempted-Value when T_k completes its execution (this must be at d_k since T_k starts execution at its LST, by Lemma 3.6). Let τ_0 be the preempted value immediately before *Abandoned* (T_i).

We begin with the case where T_i is overthrown. In this case, c_i is added to τ_0. Furthermore, all the tasks that execute until T_k must also be overthrown and hence their values will be added to Preempted-Value. The sum of these values is exactly $(\tau - (\tau_0 + c_i))$. The total amount of processor time these tasks used cannot be greater than this value. Hence the length of the time interval between *Abandoned* (T_i) and the point where T_k starts its execution is at most $(\tau - (\tau_0 + c_i))$. We also know that *Abandoned* $(T_i) - (d_i - c_i) \leq c_i$ (a task can be overthrown only in its *Latest start interval*) and that $c_k \geq \tau$ (otherwise T_k could not be scheduled by TimerInterrupt). Hence,

$$\begin{aligned} d_i - c_i &\geq \textit{Abandoned}\,(T_i) - c_i \\ &\geq (d_k - c_k - (\tau - (\tau_0 + c_i))) - c_i \\ &\geq d_k - c_k - \tau \geq d_k - 2c_k. \end{aligned}$$

The second case is when T_i is rejected by T_j. In this case, T_i's value is not added to Preempted-Value. However we know that $c_i \leq c_j + \tau_0$. We also know that $c_j \leq c_k$ since T_k was scheduled by TimerInterrupt after T_j (or $T_i = T_j$). Hence we have that $c_i \leq c_j + \tau_0 \leq c_k + \tau_0$. As before the length of the time interval between *Abandoned* (T_i) and the point where T_k starts its execution is at most $(\tau - \tau_0)$ and *Abandoned* $(T_i) - (d_i - c_i)) \leq c_i$ and that $c_k \geq \tau$. Hence,

$$\begin{aligned} d_i - c_i &\geq \textit{Abandoned}\,(T_i) - c_i \\ &\geq (d_k - c_k - (\tau - \tau_0)) - c_i \\ &\geq (d_k - c_k - \tau + \tau_0) - (c_k + \tau_0) \\ &\geq d_k - 2c_k - \tau \geq d_k - 3c_k. \end{aligned}$$

Lastly, we have to consider the case in which T_i is discarded by T_j. T_j can have no slack time since Preempted-Value is greater than 0, hence $\delta_i \subseteq \delta_j$. If $j = k$, the proof is complete. If $j \neq k$, then T_j is overthrown. Replacing i by j in the above arguments, we conclude that $d_i - c_i \geq d_j - c_j \geq d_k - 3c_k$.

Next, we now show that $d_i \leq d_k + 2c_k$. It always holds that $d_k \geq$ *Abandoned* $(T_i) \geq d_i - c_i$. Assume that $d_i - d_k > 2c_k$ then $c_i \geq d_i - d_k > 2c_k \geq c_k + \tau$. Under these conditions the value of T_i is too big for it to be abandoned in the first place [15].

In conclusion $\delta_i \subseteq \Delta_k$ holds in every case.

B. Proofs for Section 4 Lemmas

In proving bounds such as those in Section 4, one usually refers to the on-line algorithm under consideration as the player. The bounds are best described as a game between a player and an adversary who makes up part of the task set, observes the player's response to it, and then extends the history by creating a new task. This process is repeated until the entire task set is complete. At the end of this process, the adversary indicates its schedule, the optimal off-line schedule.

The tasks created by the adversary are of two kinds:

- **major tasks**, which have no slack time. In other words, the time between a major task's release time and its deadline corresponds exactly to its computation time. It is easily observed that if the release time of a major task is between the release time and deadline of another, at most one of the two can complete.

[15] Suppose T_i was *Abandoned* by T_j. $c_i \geq c_k + \tau \geq c_j + \tau_0$ implies that T_i could not be rejected or overthrown . If T_i was discarded then T_j can have no slack time since Preempted-Value is greater than 0, hence $c_i < c_j$, a contradiction.

- **associated tasks**, which may or may not have slack time.

The adversary uses the following device to force the hand of the player. For a major task S_i of length L_i, with release time r_i and deadline d_i, the adversary may also create a sequence of associated tasks of length ϵ, each one being released at the deadline of the previous one of the sequence. Clearly any algorithm that schedules any one task of this associated task sequence cannot schedule S_i. This sequence of associated tasks stops the moment the player chooses to abandon S_i in favor of a task of the sequence. Otherwise, the sequence continues until d_i is reached. If the player chose to abandon S_i in favor of a task of the sequence, the value obtained by the player is ϵ rather than L_i. The adversary chooses ϵ to be arbitrarily small compared to L_i. A major task S_i that has associated tasks as above is called a *bait*. Otherwise it is simply called *normal*.

Time is divided into *epochs*. In each epoch, the adversary starts off by first creating a major task T_0 of length $t_0 = 1$. In general, after releasing major task T_i of length t_i, the adversary releases a major task T_{i+1} of length t_{i+1}, at time ϵ before the deadline of T_i. If the player ever chooses to schedule an associated task the epoch ends with the release of T_{i+1}. If the player chooses to abandon T_i in favor of T_{i+1}, this process continues, otherwise the epoch ends with the release of T_{i+1}. In the above description, all tasks except T_{i+1} are baits; T_{i+1} is normal. At any rate, no epoch continues beyond the release of T_m, where m is a finite positive integer.

We note that the player never abandons a bait for one of its associated tasks, since in doing so the value obtained by the player during the epoch is negligible — i.e., ϵ. Thus, during an epoch the player either schedules only T_i, $i < m$ to completion, or the player only schedules T_m to completion.

Lemma 4.1 PROOF.
For this proof, the associated tasks have no slack time, and the length of task T_{i+1} is computed according to

$$t_{i+1} = (c \cdot t_i - \sum_{j=0}^{i} t_j)$$

(where c is a constant whose exact value will be specified later in this proof). If the player scheduled only T_i, $i < m$ to completion, the player's value is t_i, whereas the adversary obtains value $\sum_{j=0}^{i+1} t_j$ (by performing the associated tasks for $T_0, \ldots, T_i$ and then performing T_{i+1}). In this case the player's value is $1/c$ times the adversary's value. If the player scheduled only T_m to completion, the player's value is t_m, while the adversary's value is $\sum_{j=0}^{m} t_j$. If c and m can be chosen such that the ratio $t_m / \sum_{j=0}^{m} t_j$ is no larger than $1/c$, then in either case the player obtains no more than $1/c$ times the adversary's value. In attempting to provide the tightest bound on the competitive factor of an on-line algorithm, therefore, our attempt is to find the smallest $1/c$ (equivalently, the largest c) such that the series defined by the recurrence relation

$$t_0 = 1 \text{ and } t_{i+1} = c \cdot t_i - \sum_{j=0}^{i} t_j$$

satisfies the property

$$\exists m : m \geq 0 : \frac{t_m}{\sum_{j=0}^{m} t_j} \leq \frac{1}{c}.$$

standard techniques from the theory of difference equations can be used to show that the property is satisfied when $c < 4$, and that the property is not satisfied when $c \geq 4$. it therefore follows that $1/4 = 0.250$ is an upper bound on the competitive factor that can be made by any on-line scheduling algorithm in an overloaded environment. □

lemma 4.2 PROOF.
in this proof, it follows from the restriction on the loading factor that the associated tasks have to have some slack time — specifically, for loading factor b, each associated task has deadline ϵ and computation time $(b-1)\epsilon$. executing all the associated tasks corresponding to task t_i (of length t_i) therefore yields a value $(b-1)t_i$. the length of task t_{i+1} is computed according to the following rule

$$t_{i+1} = c \cdot t_i - \sum_{j=1}^{\lceil i/2 \rceil} t_{i+1-2j} - (b-1) \sum_{j=0}^{\lfloor i/2 \rfloor} t_{i-2j}$$

(where c is a constant whose exact value will be specified later in this proof). as before, note that the player either scheduled only t_i, $i < m$ to completion, or the player only scheduled t_m to completion. in the first case, the player's value is t_i, whereas the adversary obtains value

$$t_{i+1} + \sum_{j=1}^{\lceil i/2 \rceil} t_{i+1-2j} + (b-1) \sum_{j=0}^{\lfloor i/2 \rfloor} t_{i-2j}$$

(by performing tasks $t_{i+1}, t_{i-1}, t_{i-3}, \ldots$, and the associated tasks for $t_i, t_{i-2}, t_{i-4}, \ldots$). in this case the player's value is $1/c$ times the adversary's value. in the second case, the player's value is t_m, while the adversary's value is

$$\sum_{j=0}^{\lfloor m/2 \rfloor} t_{m-2j} + (b-1) \cdot \sum_{j=1}^{\lceil m/2 \rceil} t_{m+1-2j}$$

(by performing tasks $t_m, t_{m-2}, t_{m-4}, \ldots$, and the associated tasks for $t_{m-1}, t_{m-3}, t_{m-5}, \ldots$). if c and m can be chosen such that the ratio $t_m / (\sum_{j=0}^{\lfloor m/2 \rfloor} t_{m-2j} + (b-1) \cdot \sum_{j=1}^{\lceil m/2 \rceil} t_{m+1-2j})$ is no larger than $1/c$, then in either case the player obtains no more than $1/c$ times the adversary's value. in attempting to provide the tightest bound on the competitive factor of an on-line algorithm, therefore, our attempt is to find the smallest $1/c$ (equivalently, the largest c) such that the series defined by the recurrence relation

$$\begin{aligned} t_1 &= 1 \\ t_{i+1} &= ct_i - \sum_{j=1}^{\lceil i/2 \rceil} t_{i+1-2j} - (b-1) \sum_{j=0}^{\lfloor i/2 \rfloor} t_{i-2j} \end{aligned}$$

satisfies the property that there is an $m \geq 0$ with

$$\frac{t_m}{\left(\sum_{j=0}^{\lfloor m/2 \rfloor} t_{m-2j} + (b-1) \cdot \sum_{j=1}^{\lceil m/2 \rceil} t_{m+1-2j}\right)} \leq \frac{1}{c}.$$

standard techniques from the theory of difference equations can be used to show that this condition is satisfied by any c that satisfies $(4(c - b + 1)^3 < 27c)$. it therefore follows that p is an upper bound on the competitive factor that can be made by any on-line scheduling algorithm in an environment with a loading factor b, $1 < b \leq 2$, where p satisfies $(4(1-(b-1)p)^3 = 27p^2)$. □

C. Performance Results

We have implemented our algorithm and Locke's Best Effort algorithm and have run simulations to accumulate empirical data. Thus far, we have run many separate experiments, while varying several parameters: e.g., the computation times, deadlines and task separation times. In each run, we typically assumed uniform distributions for each of the quantities, and varied their ranges. The parameters are treated as mutually independent random variables. Each run of the experiment had anywhere from 300 to 1000 tasks.

Since computing the maximum value obtainable by a clairvoyant algorithm is a hard problem, we have instead used a rather simplistic upper bound on this maximum value, which is obtained by summing up the value of all tasks; this quantity will be referred to as UB (Upper Bound). (Note that even a clairvoyant algorithm may not achieve such a high value.) We compare the value obtained by our algorithm, D*, with the value obtained by Locke's best effort scheduler, BE, using the following parameter:

$$\text{VI} = \text{Percentage of UB value obtained}$$

We classify our environment into three categories based on how much overloading is expected: *extreme* (small inter-arrival time, large execution time, and small slack time), *normal* (average inter-arrival time, large execution time, and small slack time) and *mild* (average inter-arrival time, average execution time, and average slack time).

The following table summarizes our empirical data assuming a uniform arrival rate:

Environment	VI of D*	VI of BE
Extreme	17.5	17.6
Normal	32.6	35.6
Mild	91.5	89.7

Table 1: PERFORMANCE COMPARISON—UNIFORM MODE. (# Tasks = 1000, # Runs = 300–1000.)

This shows that the algorithms are comparable. Since the Best Effort algorithm is a heuristic that cannot guarantee any performance bound, it is not hard to come up with cases where Locke's heuristic exhibits pathological behavior. While it is not surprising that our algorithm would behave better in these situations, the magnitude of the difference is instructive.

Consider a situation in which tasks enter in bursts. Each burst consists of several short tasks arriving simultaneously with a relatively long task that completes before the next burst arrives. Long tasks have zero slack time and short tasks, a maximum slack time of 3 units. The simulation executes for approximately 50 bursts. All simulations executed for 5000 time units, whereas long tasks last 100 units which is less than the period between bursts. The large performance differences are due to the fact that Locke's algorithm will choose a task to delete at random. It is as likely to be a long job as a short one, whereas our our algorithm will correctly discard the shorter ones.

Short Tasks' Duration	VI of D*	VI of BE
5 time units	68.7	3.3
10 time units	53.5	5.0
25 time units	33.2	7.0
50 time units	22.5	8.3

Table 2: PERFORMANCE COMPARISON—BURSTY MODE. (# Short Tasks = 10, # Long Tasks = 1, Short Tasks' Duration = 5–50 units, Long Tasks' Duration = 100 units, Short Tasks' Slack Time = 3 units, Long Tasks' Slack Time = 0 unit, # Bursts = 50, Simulation Time = 5000 units.)

The effect of varying slack-times is significant. In the following table we summarize the cases where the maximum slack time for short tasks (each of duration 5 units) varies in a range between 0 and 10 units. As in the earlier simulation, the long tasks last for 100 units; the simulation executes for approximately 50 bursts and executed for 5000 time units.

Short Tasks' Slack Time	# Short Tasks	VI of D*	VI of BE
0 time units	10	68.7	3.3
3 time units	10	68.7	3.3
5 time units	10	68.7	6.1
10 time units	10	67.4	8.5

Table 3: PERFORMANCE COMPARISON—BURSTY MODE WITH SLACK.

(# Short Tasks = 10, # Long Tasks = 1, Short Tasks' Duration = 5–50 units, Long Tasks' Duration = 100 units, Short Tasks' Slack Time = 3 units, Long Tasks' Slack Time = 0 unit, # Bursts = 50, Simulation Time = 5000 units.)

An explanation of the phenomena observed in the last two experiments can be given in terms of the following simple example. Assume that a long task arrives ϵ $(0 < \epsilon < 10)$ time units after a short task and that the long task has no slack time. The BE heuristic then observes that the short task has a value density of $\frac{10}{10-\epsilon} > 1$, whereas the long task has a value density of 1 and decides to continue with the short task. However, D* abandons the short task in favor of the long task, since the long task has a value much higher than the cumulative preempted value so far (which is < 10). In the process, the algorithm D* obtains a much higher value during each burst period, in comparison to the BE heuristics.