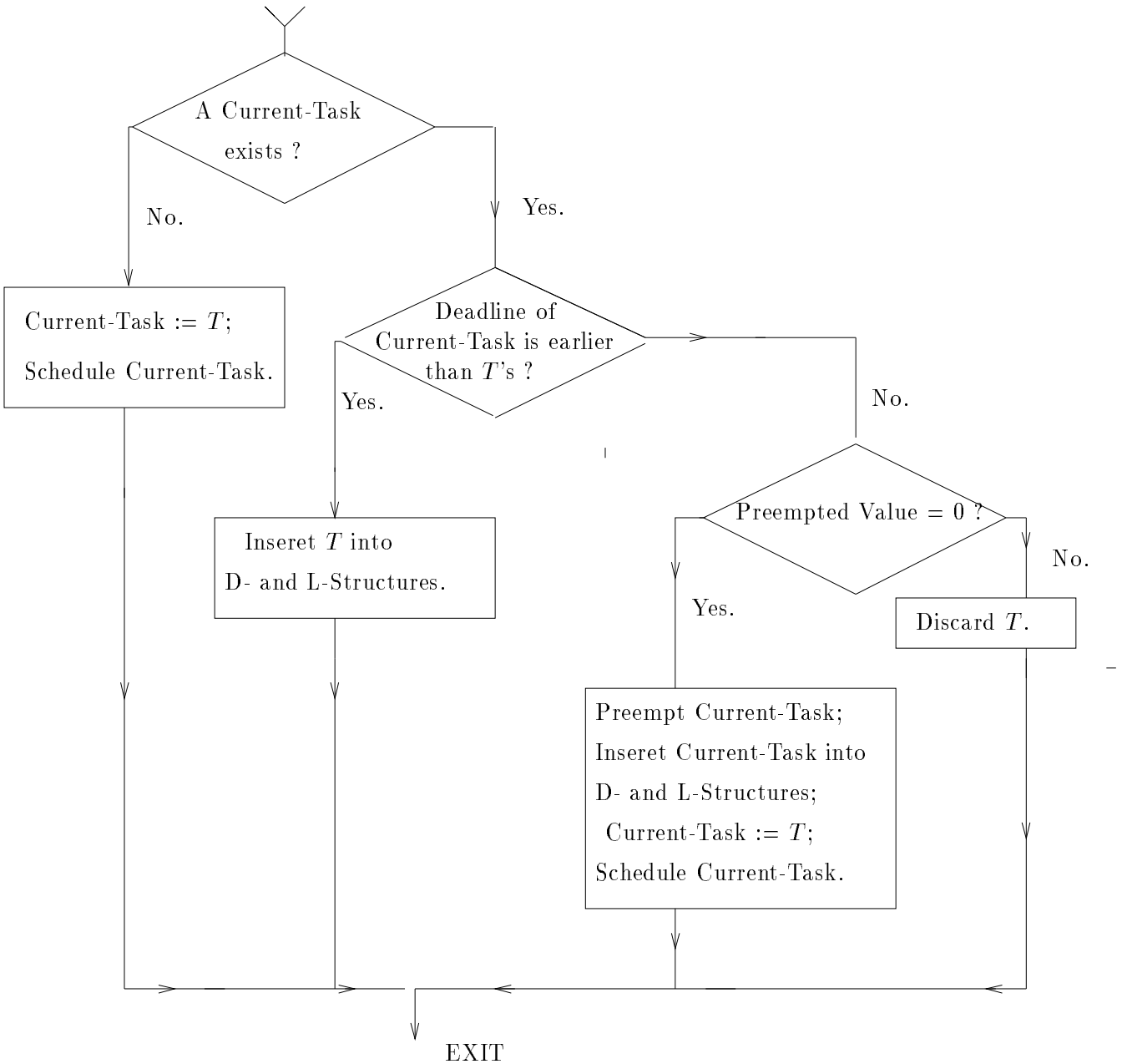



Figure 9: Flow Chart for the TaskRelease for task $T$.

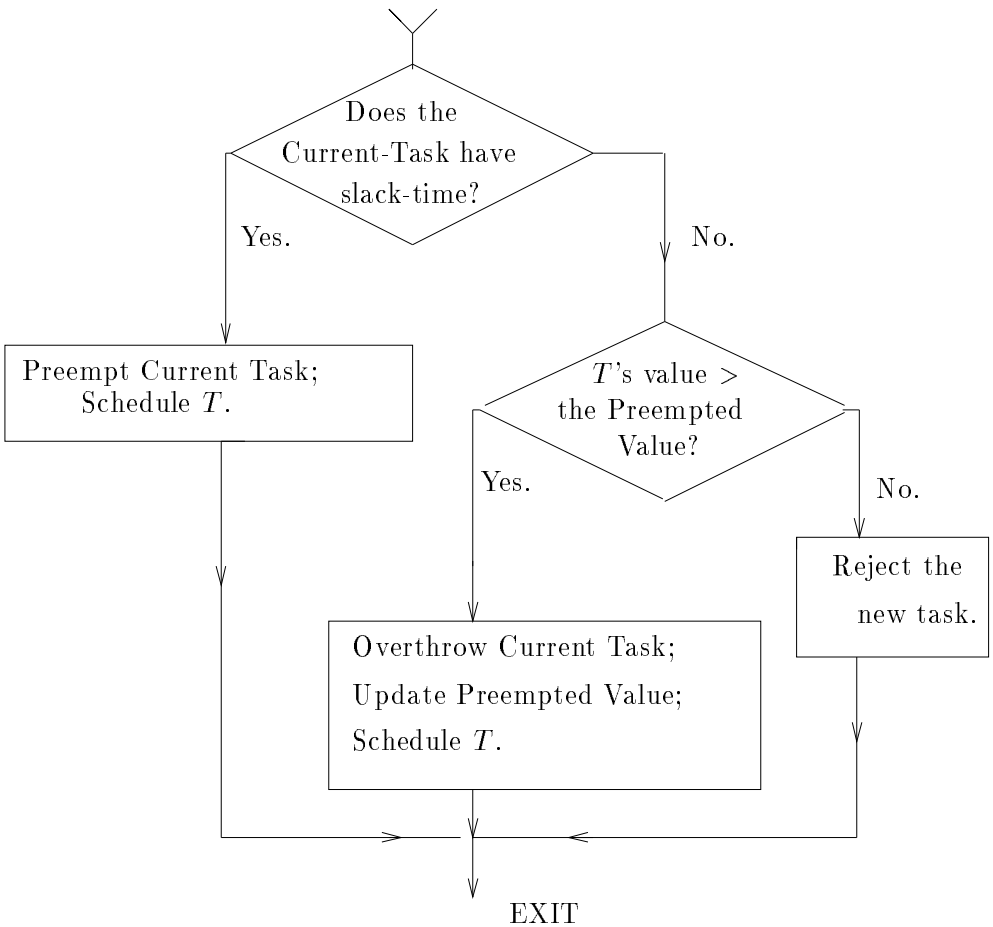



Figure 8: Flow Chart for TaskInterrupt for task $T$.

# B Coincident Interrupts.

Note that the D* algorithm allows *coincident interrupts*,i.e., several interrupts are allowed to occur at the same time.

Suppose at some point in time we have a *collection* of interrupt events $E$ waiting to be served. This collection can include at most one TaskCompletion event, but there is no bound on the number of TaskReleases and TimerInterrupts events. The events will be handled according to their priorities, i.e., TaskCompletion first and then all other interrupts. There is no specific order of service among the several TimerInterrupts and TaskReleases.

We will now describe what happens when a TaskCompletion event is present in $E$. Other cases are treated similarly. The TaskCompletion event is served first. It sets the preempted value to zero and sets a new Current-Task for execution (this task might be among the tasks that generated TimerInterrupt, in this case this specific TimerInterrupt will be canceled since it was already served) [11]. Next the TaskReleases and TimerInterrupts are served in some arbitrary order. We will view this as a sequence of consecutive events with 0 time between them. These events would be handled in the usual way. This mean that Current-Task and Preempted-Value might be updated several times, some tasks might be rejected and others might be added to the D- and L-Structures or execute *trivially.* [12]

Why should TaskCompletion events be handled first? The TaskCompletion routine implicitly removes the current task from the system and sets the Preempted-Value to zero. Suppose $E$ includes TaskCompletion and other events. Furthermore, suppose we serve the TaskCompletion event after a TaskRelease , which scheduled a new task. Then, when the TaskCompletion routine is executed it would implicitly remove the new task from the system and would schedule the next task from the D- and L-Structures. Both actions would be wrong. If we serve a TimerInterrupt event before the TaskCompletion event then obviously the value of Preempted-Value, when executing the TimerInterrupt routine, is not the correct one (it should be zero). Later on when the TaskCompletion is served, it will set the Preempted-Value to zero in which case the value of Preempted-Value would not correspond to the actual preempted value. [13]

All of our claims about the behavior of D* hold also when we consider coincident interrupts. The reason is that their proofs rely only on the temporal order of events and not on the specific temporal distance between two consecutive events. For example in Lemma 3.7, $T_i$ was abandoned due to $T_j$ which was abandoned due to $T_{J_1}$ and so on until $T_k$. The identity of these tasks is not important nor is the fact that some of the tasks executed *trivially.* All that matters is that each task was abandoned due to the next task in the sequence. Different orders of serving coincident events might result in an altogether different sequence and possibly a different $T_k$ but the claim holds for this sequence as well.

[11] This is the only place where the presence of coincident interrupts make a difference in the execution of D*. In all other cases the events have no "dependencies."

[12] Note that a TaskRelease event might preempt the Current-Task. If the preempted task has no slack time, then an immediate TimerInterrupt event would be generated. This new event will be added to $E$.

[13] Also, in lemmas 3.6, 3.7 and 3.8 we refer to the first task $T$ that completed after some particular task $T_i$ was abandoned. If $T_i$ was abandoned coincidentally with a completion of some task $T_j$ we definitely do not want to consider $T_j$ as the task that completed after $Abandoned\ (T_i)$ but rather as a task that completed previously.

# Appendix

## A The earliest-deadline-first algorithm (D).

```
D-Structure := ∅;

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

      TaskCompletion:
            (* This interrupt occurs when either the Current-Task completes or it reaches
            its deadline. *)
            Remove from the D-Structure all tasks that passed their deadlines.
            if D-Structure is non-empty then
            (* T is the first task in D-Structure. *)
                  Remove T from the D-Structure;
                  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
                  (* If T has an earlier deadline than that of the current-task then T is sched-
                  uled for execution else T is added to the D-Structure. *)
                  Insert T into the D-Structure;
            else
                  Preempt Current-Task;
                  Insert Current-Task into the D-Structure by its deadline.
                  Current-Task := T;   Execute Current-Task;
            end{if }
      end{TaskRelease}

end{loop }
end{D}.
```

$D$ : On-line Scheduling Algorithm.

## References


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

[2] S. Baruah, G. Koren, B. Mishra, A. Raghunathan, L. Rosier and D. Shasha. On-line Scheduling in the Presence of Overload,. Submitted to *1991 IEEE Symposium on Foundations of Computer Science* , San Juan, Puerto Rico, 1991.

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

[4] 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

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

[6] E. L. Lawler, J. K. Lenstra, A. H. G. Rinnooy Kan and D. B. Shmoys. Sequencing and Scheduling: Algorithms and Complexity. *Handbooks in Operations Research and Management Science, Vol. 4: Logistics of Production and Inventory*, To appear, 1990.

[7] E. L. Lawler and C. U. Martel. Scheduling periodically occurring tasks on multiple processors. *Information Processing Letters*, 12:9-12, 1981.

[8] J. P. Lehoczky and L. Sha. The Average Case Behaviour of the Rate Monotonic Scheduling Algorithm. *Technical Report, Carnegie-Mellon University*, 1986.

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

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

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

[12] V. J. Rayward-Smith. UET Scheduling with unit interprocessor communication delays. *Discrete Applied Mathematics*, 18:55-71, 1987.

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

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

[15] A. S. Tanenbaum and R. van Renesse. Distributed Operating Systems. *ACM Computing Surveys*, 17:419-470, 1985.

# 6 Conclusion

We present a simple generalization of the earliest-deadline-first algorithm that gives a guarantee for overloaded real-time systems. Being able to make a guarantee is novel for overloaded systems with only sporadic jobs. Moreover, our simulation results show that the algorithm is a good heuristic. Finally, our algorithm can be implemented efficiently on basic data structures. We also show that no on-line algorithm can have a competitive factor greater than 1/3.224745.

There are four extensions, separately and in combination, that require study.

- How can we improve our bounds in systems that are "mostly" periodic?
- What can we say in the parallel case? Is there a generic transformation of our algorithm or any other algorithm that still gives acceptable guarantees?
- We have assumed the uniform value density model in this paper. Thus, for every task, the ratio of its value to its computation time is the same. What happens when multiple value densities are possible? (Bounds on the *competitive factors* in this setting are given in [2]).
- In spite of the good performance, exhibited both theoretically and empirically, D* is not a competitive algorithm, since it has a *competitive factor* equal to 0. To see the claim, consider the following system:

  The system has 1000 tasks. Task $T_i$ has release time $i - i\epsilon$, computation time 1 and deadline 1000, $i$ ranges from 0 to 998. Task $T_{999}$ has release time $999 - 999\epsilon$, deadline 1000 and no slack time (note that $T_{999}$ has the greatest value). At time 0 task $T_0$ starts to executes, $T_1$ preempts it at $1 - \epsilon$. $T_1$ is preempted by $T_2$ at $2 - 2\epsilon$. This process cascades until $T_{998}$ is preempted by $T_{999}$. At time $1000 - \epsilon$ all tasks reach their LST and all of them are rejected. $T_{999}$ is the only task that executes to completion. The value obtained by D* is approximately 1/1000 the value that a clairvoyant algorithm can obtain. By generalizing this argument, one can show that the competitive factor of D* is $< 1/N$, for arbitrarily large $N$

  An important and intriguing open question is either to devise an on-line scheduling algorithm with a nonzero *competitive factor* or to show that no such algorithm exists.

# 7 Acknowledgements

The authors are grateful for the enthusiastic support and encouragement they received from Marc Donner of IBM. The authors acknowledge with gratitude the considerable help and advice they have received from Doug Locke of IBM, C.U. Martel of University of California at Davis and Sanjoy Baruah and Louis Rosier of University of Texas at Austin. The authors also thank Robert Kagel for implementing the algorithms and running the experiments. The experiments helped us to improve the algorithm and, in one case, uncovered a subtle bug.

| Short Tasks' Duration | Algorithm | VI |
|---|---|---|
| *5 time units* | D* | 68.7 |
| | BE | 3.3 |
| *10 time units* | D* | 53.5 |
| | BE | 5.0 |
| *25 time units* | D* | 33.2 |
| | BE | 7.0 |
| *50 time units* | D* | 22.5 |
| | BE | 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 | Algorithm | # Short Tasks | VI |
|---|---|---|---|
| *0 time units* | D* | 10 | 68.7 |
| | BE | | 3.3 |
| *3 time units* | D* | 10 | 68.7 |
| | BE | | 3.3 |
| *5 time units* | D* | 10 | 68.7 |
| | BE | | 6.1 |
| *10 time units* | D* | 10 | 67.4 |
| | BE | | 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 figure 7. Assume that the long task arrives $\epsilon$ ($0 < \epsilon < 10$) time units after the 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.

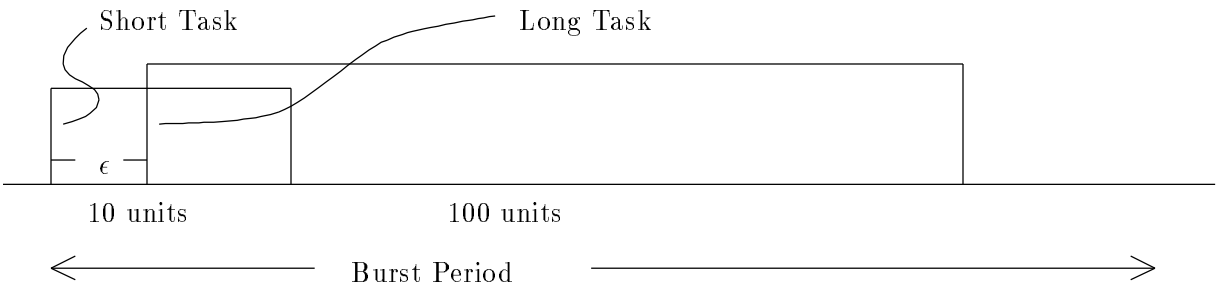



Figure 7: Situations where BE heuristics behave pathologically.

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

| Environment | Algorithm | VI |
|---|---|---|
| *Extreme* | D* | 17.5 |
| | BE | 17.6 |
| *Normal* | D* | 32.6 |
| | BE | 35.6 |
| *Mild* | D* | 91.5 |
| | BE | 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. (See figure 7.) 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.

Time is divided into *epochs*. In each epoch, the adversary starts off by first creating a task $T_1$ of length $\tau$. In general, after releasing a task $T_i$ of length $i\tau$, the adversary releases a task $T_{i+1}$ of length $(i+1)\tau$ at time $\epsilon$ before the deadline of $T_i$. 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 $T_6$. (See figure 6.)

We first note that the player never abandons a bait for one of its associated tasks, since the value so obtained is negligible. Thus, the player either scheduled only $T_i$, $i < 6$ to completion, or the player only scheduled $T_6$ to completion. In the first case, the player's value is $i\tau$, whereas the adversary obtains value

$$\left(\frac{(i+1)(i+2)}{2}\right)\tau$$

(by performing the associated tasks for $T_1 \ldots T_i$ and then performing $T_{i+1}$). In this case the player's value is $2i/((i+1)(i+2))$ times the adversary's value. It is easily checked that the player can maximize this by setting $i = 1$ (scheduling $T_1$ to completion), thus obtaining $1/3$ the value obtained by the adversary. (See figure 6.) In the second case, the player's value is $6\tau$, while the adversary's value is $21\tau$. In this case the player obtains $2/7$ times the adversary's value.

The factor can be easily improved to $1/(2+\frac{\sqrt{6}}{2})$ by the following simple modification. The adversary first starts off with task $T_0$ of length $\delta\tau$ followed by the task $T_1$ of length $\tau$ released at time $\epsilon$ before its deadline, and so on as above. $T_0$ is also a bait. The factor of $1/(2+\frac{\sqrt{6}}{2})$ is now easily obtained by an argument similar to the preceding one by setting $\delta = \sqrt{6} - 2$. □

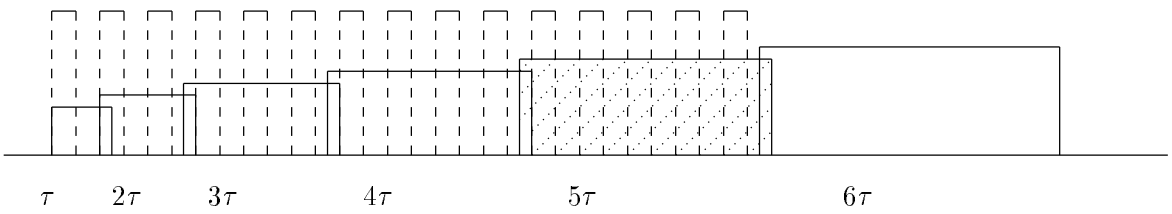
# 5 Performance Results

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.

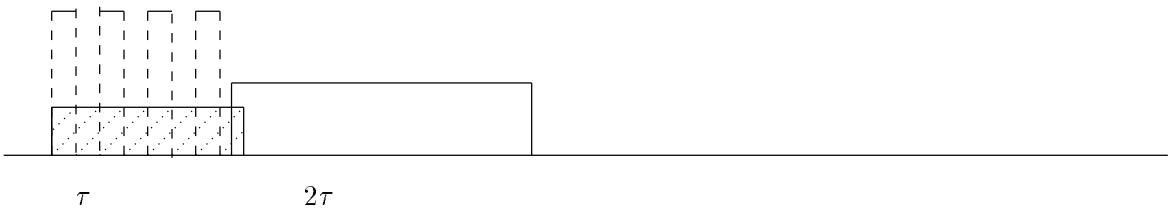
Since computing the maximum value obtainable by a clairvoyant algorithm is a hard problem, we have instead used a rather simplistic upper bound on this maximum value, which is obtained by summing up the value of all tasks; this quantity will be referred to as UB (Upper Bound). (Note that even a clairvoyant 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:

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

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).




*Case 1*: The player chooses a task of length $5\tau$.




*Case 2*: The player makes the optimal choice.

---

Figure 6: Lower Bound Proof.

For the sake of this lower bound argument, we assume that tasks have no slack time. In other words, the time between a task's release time and its deadline corresponds exactly to its computation time. It is easily observed that if the release time of a task is between the release time and deadline of another, at most one of the two can complete.

The adversary uses the following device to force the hand of the player. For a 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 $\epsilon$, 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 $\epsilon$ rather than $L_i$. The adversary chooses $\epsilon$ to be arbitrarily small compared to $L_i$. A task $S_i$ that has associated tasks as above is called a *bait*. Otherwise it is simply called *normal*.

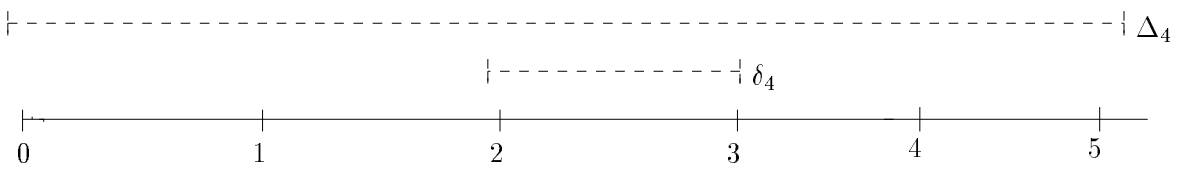



Figure 5: $\Delta_i$ AND $\delta_i$ OF THE TASK $T_4$.

A clairvoyant scheduler can do much better. Abandoning $T_1$ and $T_2$ and scheduling $T_4$ before the release of $T_3$ would enable a successful completion of $T_3$, $T_4$ and $T_5$. The value obtained in this way is $4 + 7\epsilon$.

### 3.4 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.*

PROOF.

As described before, the algorithm D* requires two data structures, called D-Structure (Deadline Structure) and L-Structure (Latest-start-time Structure).

These data structures can be 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 if there are $n$ tasks in the structures.

Now, we need to argue only that the cost incurred by any task is $O(\log n)$. A task during its lifetime causes one TaskRelease event, at most one TaskCompletion event and at most one TimerInterrupt event. The rest follows from the observation that each event incurs a cost of $O(\log n)$. □

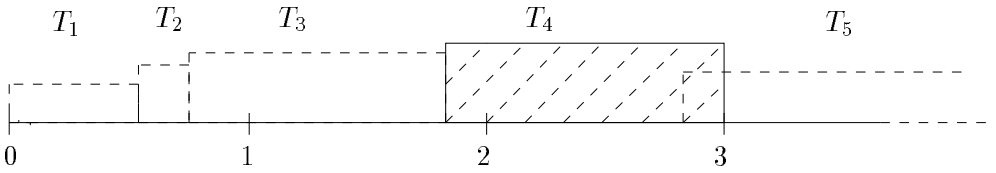
## 4 A Lower Bound on the Value Obtained by Any On-line Algorithm

In this section, we show that no on-line algorithm can obtain a value better than $1/(2 + \frac{\sqrt{6}}{2})$ ($\approx 1/3.224745$) times the value obtained by an off-line algorithm in the presence of overload. This lower bound shows that the algorithm of the previous section comes close to being competitively optimal.

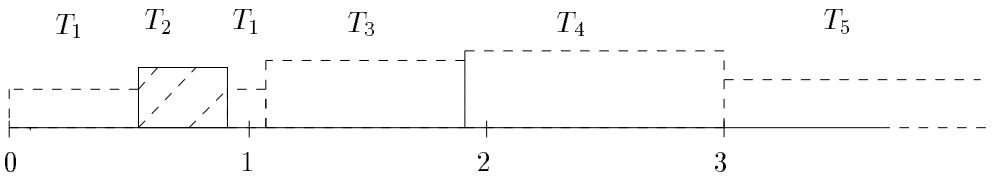
**Theorem 4.1** *No on-line algorithm can obtain a value better than $1/(2 + \frac{\sqrt{6}}{2})$ ($\approx 1/3.224745$) times the value obtained by an off-line algorithm in the presence of overload.*

PROOF.

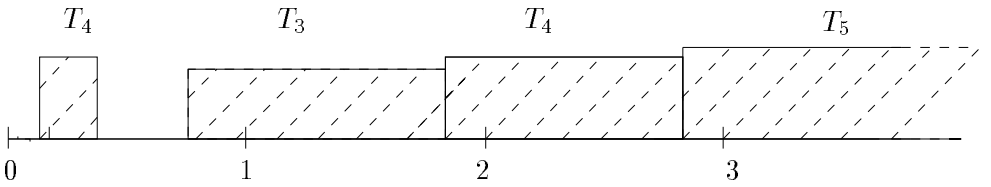
In proving lower bounds, one usually refers to the on-line algorithm under consideration as the player. Lower bounds are best described as a game between a player and an adversary who makes up part of the task set, observes the player's response to it, and then extends the history by creating a new task. 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.




(A) Behavior of the algorithm $D^*$.




(B) Behavior of the algorithm $D$.




(C) An optimal schedule.

Figure 4: THE ONE-FIFTH FACTOR CAN NOT BE IMPROVED.

$T_3$ is scheduled. Next, at $1 - \frac{3}{2}\epsilon$, $T_1$ reaches its $LST$ and causes a TimerInterrupt. However, its value is not big enough and $T_1$ is rejected . The preempted value does not change. $T_3$ continues to execute without interruptions till $T_4$ causes a TimerInterrupt at $2 - 3\epsilon$. This time, $T_4$'s value is big enough ($1 + 3\epsilon > 1 + 2\epsilon$), so $T_3$ is overthrown and the preempted value is set to $2 + 2\epsilon$ and $T_4$ is scheduled. The last event in this example occurs when $T_5$ reaches its $LST$ at $3 - \epsilon$ its value $2+4\epsilon$, is smaller than the minimum value needed to overthrow $T_4$ ( i.e. $2+5\epsilon = (1+3\epsilon)+(1+2\epsilon)$). $T_5$ is rejected and $T_4$ successfully completes at its deadline. The total value that D* achieves in this system is the value of $T_4$ that is $1 + 3\epsilon$ (All the $\delta_i$'s ,for $i = 1, 2, 3$ and 4 are sub-intervals of $\Delta_4$, see Figure 5).

The behavior of the algorithm $D$ is shown in the case (B) of figure 4. One can see that only $T_2$ completes its execution on time. All other tasks miss their deadlines and hence have positive lateness. Note that all these tasks are scheduled up to their deadlines even when it is already clear that they will miss their deadlines.

What is the overloaded period in this case. The union of the $\delta_i$'s of all the tasks with positive lateness [10] yield $OL = [0, 5 + 3\epsilon]$ as the over load period. The quotient $\frac{1+3\epsilon}{5+2\epsilon}$ tends to $\frac{1}{5}$ as $\epsilon$ goes to zero. We conclude that D* can not guarantee more then one-fifth of the overloaded interval.

[10] Note that $T_2$ does not have positive lateness although its deadline in the overload period.

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[8] the following:

$$V = \sum_{j \in S} |\delta_j| = \frac{1}{5} \times \sum_{j \in S} |\Delta_j| \geq \frac{1}{5} \times \left| \bigcup_{j \in S} \Delta_j \right| \geq \frac{1}{5} \times \left| \bigcup_{i=1}^{q} \delta_i \right| \geq \frac{1}{5} \times |C|.$$

This is the desired result. □

**Remark 3.3** Notice that $V$, the value that D* obtained from all tasks with deadline in the overloaded interval $C$, can in fact be bigger than the length of C. For example, it is possible when some of those tasks were released well in advance with large slack time and used previous idle time during previous underload intervals; these tasks may not have positive lateness.

**Remark 3.4** Our algorithm allows *coincident interrupts*. That is, several interrupts may occur at one time. All of our claims about the behavior of D* hold also when we consider coincident interrupts (see appendix B for details).

### 3.3 Can the performance guarantee factor of one-fifth be improved?

We saw that D* obtains a value that is at least *one-fifth* the length of the overloaded period from all tasks with deadlines in the overloaded period. Is it possible to have a performance guarantee for D* with a bigger constant? The answer is no. We show that using the following counter example.

**Example 3.5** Consider the following overloaded system with five tasks labeled $T_1$, $T_2$, $T_3$, $T_4$ and $T_5$. (See figure 4.) [9]

| Task | *Release-Time* | *Computation-Time* | *Deadline* | *Latest start interval* ($\delta_i$) | $\Delta_i$ |
|---|---|---|---|---|---|
| $T_1$ | 0 | $1-\epsilon$ | $1+\frac{1}{2}\epsilon$ | $[\frac{3}{2}\epsilon, 1+\frac{1}{2}\epsilon]$ | |
| $T_2$ | $1-3\epsilon$ | $2\epsilon$ | $1-\epsilon$ | $[1-3\epsilon, 1-\epsilon]$ | |
| $T_3$ | $1-2\epsilon$ | 1 | $2-2\epsilon$ | $[1-2\epsilon, 2-2\epsilon]$ | |
| $T_4$ | 0 | $1+3\epsilon$ | 3 | $[2-3\epsilon, 3]$ | $[-9\epsilon, 5+6\epsilon]$ |
| $T_5$ | $3-\epsilon$ | $2+4\epsilon$ | $5+3\epsilon$ | $[3-\epsilon, 5+3\epsilon$ | |

The behavior of the algorithm D* is shown in the case (A) of figure 4. First, D* schedules $T_1$ to run at time 0. At time $1-3\epsilon$, $T_2$ is released into the system and, since the preempted value is zero, $T_2$ is scheduled for execution. At time $1-2\epsilon$ $T_3$ is released and since its deadline is after $T_2$'s deadline it is inserted to the D- and L-Structures but since $T_3$ has no slack time a TimerInterrupt immediately occurs. $T_2$ is overthrown and the preempted value is set to $2\epsilon$ and

[8] $|\cdot|$ means the length of the interval

[9] In this example $T_5$ can be released at any point from 0 to $3-\epsilon$ the D* scheduling will be affected.

task can be overthrown only in its *Lastest start interval* ) and that $c_k \geq \tau$ (otherwise $T_k$ could not be scheduled by TimerInterrupt). Hence,

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

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 $T_k = T_j$). 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 $(\tau - \tau_0)$, $Abandoned\,(T_i) - (d_i - c_i)) \leq c_i$ and $c_k \geq \tau$. Hence,

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

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 that

$$d_i - c_i \geq d_j - c_j \geq d_k - 3c_k.$$

- 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 [7].

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

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

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

PROOF.

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

[7] 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.

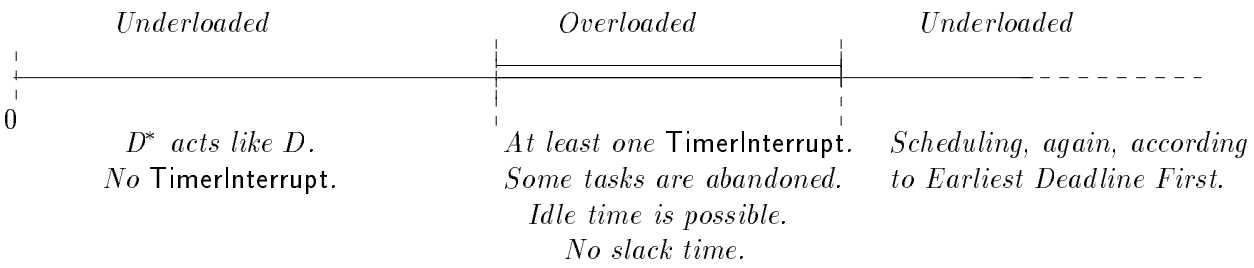



Figure 3: Overloaded and Underloaded Intervals.

positive. The last possibility is that $T_i$ executes with no interruption until its completion. In all cases the lemma holds. □

**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 $T = T_k$.*

PROOF.

It is clear that *Abandoned* $(T_i)$ cannot occur after $C$. Suppose $T_i$ was abandoned due to some task, say $T$. Lemma 3.6 tells us that $T$ has no slack time hence its deadline is in $C$ (Lemma 3.4). So, there is an index $j$ so that $T = T_j$. If $T_j$ completed its execution then $j = k$, the proof is complete, otherwise $T_j$ is abandoned due to another task , say $T_{j_1}$. Replacing $T$ by $T_j$ and $T_j$ by $T_{j_1}$ in the above yields that $d_{j_1}$ is in $C$. We can carry this argument inductively up to $T_k$ and conclude that $d_k$ is in $C$. □

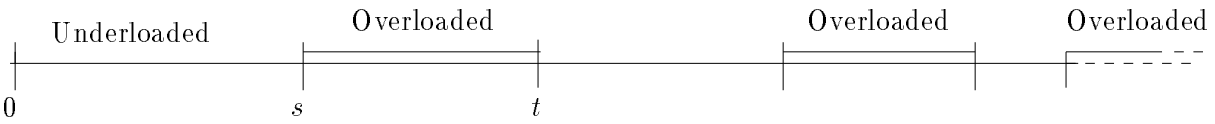
**Lemma 3.8** *Suppose $T_i$ was abandoned in $C$. Let $T_k$ be the the first task that completed execution after Abandoned $(T_i)$ then* [6] *$\delta_i \subseteq \Delta_k$.*

PROOF.

- First let us show that $d_i - c_i \geq d_k - 3c_k$.
  Let $\tau$ 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 $\tau_0$ 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 $\tau_0$. 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

thus, the new task will be overthrown and will have executed trivially. Preempted-Value will be strictly positive beyond this point.

[6] Remember that for a task $T_i$, $d_i$ denotes *Deadline*$(T_i)$ and $c_i$ denotes the initial *Computation-Time*$(T_i)$. Also, $\delta_i$ denotes the interval starting at $d_i - c_i$ and ending at $d_i$, $\Delta_i$ is the interval between $d_i - 3c_i$ and $d_i + 2c_i$.




(A) D schedules H, first overloaded interval is [s,t].

(B) D schedules $H'$.

Figure 2: The Overloaded Periods $OL$ and $OL_2$.

In the following lemma, a task executes *nontrivially* if it executes for some strictly positive amount of time[4].

**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.*

PROOF.

Some task must successfully complete its execution after *Abandoned* $(T_i)$, because each time a task is abandoned, a new task takes its place. Notice that the Current-Task at *Abandoned* $(T_i)$ has no slack time and that once the Preempted-Value is greater than 0, no task can be scheduled by TaskRelease, until one task completes execution successfully. So, once the Preempted-Value is greater than zero, every task, executed either *trivially* or *nontrivially*, is scheduled by TimerInterrupt (with no slack time). We now must establish what happens until Preempted-Value becomes greater than zero.

If $T_i$ is discarded, then Preempted-Value is greater than 0 at *Abandoned* $(T_i)$.

If $T_i$ is overthrown , then Preempted-Value is greater than 0 immediately after *Abandoned* $(T_i)$.

If $T_i$ is rejected due to $T_j$, then Current-Task at *Abandoned* $(T_i)$, $T_j$, has no slack time, since our algorithm says that $T_i$ cannot be rejected otherwise. It is still possible, however, that Preempted-Value is 0. What can happen later on with $T_j$? $T_j$ can be overthrown, the Preempted-Value would become positive and the lemma is proved. $T_i$ can be interrupted by TaskRelease, the new task would be executed *trivially* [5] , hence again the Preempted-Value would become

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

[5] Suppose that TaskRelease schedules a new task. This implies that the deadline of the new task precedes that of $T_j$; hence it has less value than $T_j$. Since $T_j$ has no slack time, it will cause a TimerInterrupt immediately;

**Definition 3.2** 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 amount of computation time still needed at time $t$, under D*, is no larger than the amount needed for a completion of $T$ at time $t$ under $D$. In particular, the above holds, if $T$ already completed at time t under D*.

**Lemma 3.4** *Using $D^*$, if at some point $x$ in [s,t] a task $T$ has no slack time then its deadline is in [s,t].*

PROOF.

Assume that $T$'s deadline is after $t$. Since $T$ has no slack time at time $x$ under D*, $T$ cannot have any slack time at time $x$ under $D$'s scheduling either, because $T$'s position is *no worse* in the schedule of D* . Now, in D at time $x$, either $T$ is the currently executing task or it has positive lateness. Lemma 3.2 tells us that $T$ cannot in fact be the currently executing task in D since $T$'s deadline is after t. Hence $T$ will have positive lateness. This implies that the overloaded interval must extend up to $T$'s deadline, a contradiction. □

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*.

Imagine that D* schedules the system up to time $t$ and at time $t$ the control is handed over to algorithm D, call the resulting history $H'$. The history $H'$ for D is different from history $H$; for every task that is currently in the system, its start time is taken to be $t$ and the computation time is set to the "remaining" computation time in the schedule of D*.

Let $OL$ be the overloaded period of $D$ for $H$ and $OL_2$ be the overloaded period of $D$ for $H'$. We assert that $OL_2 \subseteq OL$ (i.e. the "new" overloaded period is contained in the "original" overloaded period). This holds because at time $t$ the task set for D* is a subset [3] 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 $D$. Hence we conclude that a task that has positive lateness in D's schedule of $H'$ must also have positive lateness D's schedule of $H$.

Let $[s_2, t_2]$ be the first overloaded interval in D's schedule of $H'$. All previous claims will hold with $[s_2, t_2]$ playing the role of [s,t]. The process can now go on inductively, thus yielding the following lemma (see figure 2).

**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.

[3] 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*.

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 with positive lateness and deadline in $[s,t]$.

**Lemma 3.1** *There is no idle time between $s$ and $t$ when algorithm D schedules $H$.*
PROOF.
For any $x \in [s,t]$ there is an index $i_0 \in V$ such that $x \in \delta_{i_0}$. Since $T_{i_0}$ has positive lateness it did not complete by time $x$ in $D$. Furthermore, $d_{i_0} \geq x$, as the deadline of $T_{i_0}$ is yet to pass. Hence the system cannot be idling at $x$; if nothing else, it has to schedule $T_{i_0}$. □

**Lemma 3.2** *All tasks scheduled by D between $s$ and $t$ have deadlines at or before $t$.*
PROOF.
For any $x \in [s,t]$ there is an index $i_0 \in V$ such that $x \in \delta_{i_0}$. If $T_{i_0}$ is the Current-Task at time $x$ then $d_{i_0} \in [s,t]$ by definition. If $T_{i_0}$ is not the Current-Task at time $x$, the Current-Task has its deadline prior to the deadline of $T_{i_0}$. Since the deadline of $T_{i_0}$ is at or before $t$, the Lemma follows. □

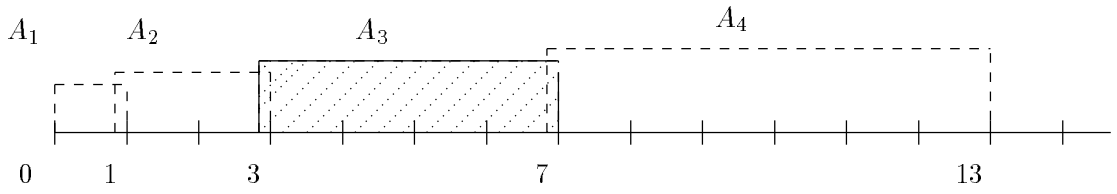
Now, let D* schedule the same history $H$. D* operates exactly like D till the first TimerInterrupt . (Note that the TaskRelease of D* is different from that of D only if the preempted value is not zero, indicating that a TimerInterrupt event had previously occurred. See appendix A). We now assert that 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*).*
PROOF.
This is a consequence of the above observation. Until $s$, D* behaves exactly as does D, since no TimerInterrupts occur before $s$. Since all tasks with deadlines before $s$ complete in the schedule of D, the result follows. □

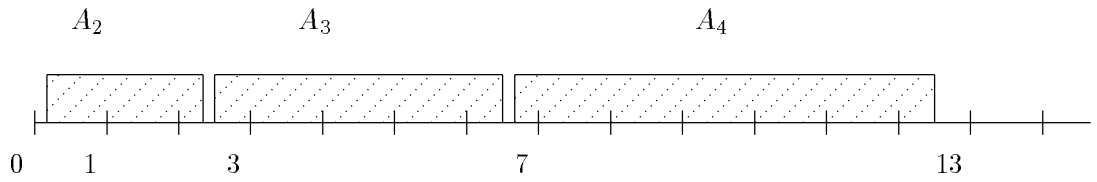
Let us look at the schedule of D* in $[s,t]$. D* might have abandoned some tasks but we assert that all such tasks have deadlines prior to $t$.

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"; in other words, either $T$ completes by $t$ or its "remaining execution time" is less than or equal to the "remaining execution time" in D's schedule. Is it possible that $T$ is abandoned? The following lemma shows that the answer is no.

**Example 3.1**




(A) Behavior of the algorithm $D^*$.




(B) An optimal schedule.

Figure 1: An example of an Overloaded System.

of the overloaded interval (i.e., $4.1 > 12.1/5$). In the case (B) of figure 1, we show a schedule (possibly obtained by a clairvoyant scheduler), in which $A_2$ runs during the period $[0.1, 2.3]$, $A_3$, during the period $[2.3, 6.4]$ and $A_4$, during the period $[6.4, 12.5]$. This schedule obtains a total value of 12.4. □

### 3.2 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

```
        TimerInterrupt:
            (* See figure 8 for a flow chart of 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 An Example

Consider the following example of an overloaded system with four tasks labeled $A_1$, $A_2$, $A_3$ and $A_4$, (See figure 1.)

| Task | *Release-Time* | *Computation-Time* | *Deadline* | *Latest start interval* ($\delta_i$) | $\Delta_i$ |
|---|---|---|---|---|---|
| $A_1$ | 0.0 | 1.0 | 1.0 | $[0.0, 1.0]$ | |
| $A_2$ | 0.1 | 2.1 | 3.0 | $[0.9, 2.1]$ | |
| $A_3$ | 0.2 | 4.1 | 7.0 | $[2.9, 7.0]$ | $[-4.3, 15.2]$ |
| $A_4$ | 0.3 | 6.1 | 13.0 | $[6.9, 13.0]$ | |

The behavior of the algorithm D* is shown in the case (A) of figure 1: First, D* schedules $A_1$ to run at time 0.0. At time 0.9, D* tries to schedule task $A_2$, and notices a conflict between $A_1$ and $A_2$. But, since the value of $A_2$ is 2.1 and since it exceeds the preempted value (which is zero) plus the value of $A_1$, $A_2$ wins. At this point $A_1$ is overthrown and $A_2$ executes. A similar situation occurs at time 2.9 and $A_2$ yields to $A_3$. However, at time 6.9, D* tries to schedule task $A_4$ and notices that $A_4$ has a smaller value than the cumulative preempted value, so far; it rejects $A_4$ and lets $A_3$ to complete. The total value that D* obtains during the period $[0, 13]$ is given by the task $A_3$ and is equal to 4.1.

When D tries to schedule this system all tasks but $T_1$ have positive lateness hence the overloaded period is $OL = [0.9, 13.0]$ We see that D* obtains more than one-fifth the length

far since the termination of the last task that completed execution. The function *value* is equal to the (original) computation time of a task. Figure 8 and figure 9 at the end of this paper are flow charts of the TimerInterrupt and TaskRelease routines.

---

On-Line Scheduling Algorithm, $D^*$:

```
D-Structure := ∅;   L-Structure := ∅;   Preempted-Value := 0;

loop
    (* TaskCompletion interrupts have higher priority than TaskRelease and TimerInterrupt.
    In the algorithm, the statement "Execute Current-Task" schedules the task to run;
    however, this task may be interrupted before it finishes. *)

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

    TaskRelease:
        (* See figure 9 for a flow chart of 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
            (* 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 TimerIn-
            terrupt, the new task is discarded .
            Note that in this case T cannot have as much value as the current-task, since
            its deadline is earlier and the current-task has no slack time. Otherwise, T
            is scheduled to execution. *)
            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}
```

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 preempted tasks 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

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

We assume that $LST(T) \geq Release\text{-}Time(T)$ (otherwise the task was infeasible when it was released). Recall that $Computation\text{-}Time(T)$ 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.

When the system is underloaded, the tasks will execute in their deadline order. However, during an overloaded interval, a task $T'$ may be executing at the latest start time of some other task $T$. Since the scheduling decision for $T$ cannot be postponed beyond $LST(T)$, in this case, the algorithm must decide which of the two tasks, $T$ and $T'$, should win, i.e., whether $T'$ must be abandoned in favor of $T$, or *vice versa*.

Before we describe the full algorithm, we need some notation. In the algorithm described below, there are three kinds of *events* (each causing an associated interrupt) considered: TaskCompletion, TaskRelease, and TimerInterrupt. TaskCompletion interrupts have higher priority than TaskRelease and TimerInterrupt interrupts. TaskRelease and TimerInterrupt interrupts share the same priority. Thus if several interrupts happen simultaneously, then the TaskCompletion interrupt is handled before the TaskRelease and TimerInterrupt interrupts. It may happen that a TaskCompletion event can remove the condition for a lower priority interrupt, e.g., by removing a task from a data structure.

- 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 of the structures and is then executed.
- The external event TaskRelease occurs when a task is released into the system; this causes insertions into the D- and L-Structures.
- The system event TimerInterrupt occurs when the wall-clock time equals the minimal value in the L-Structure. These interrupts indicate overloaded situations, 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

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 enters the system, $T_e$, 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.

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

where

$$F(T_{i_j}) = min\{S(T_{i_j}) + \mathit{Computation\text{-}Time}(T_{i_j}), \mathit{Deadline}(T_{i_j})\}, \quad (j \geq 1).$$

$\mathit{Computation\text{-}Time}(T_{i_j})$ 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 bigger than zero. In other words, a task that did not complete by its deadline in the schedule of the algorithm D has positive lateness. 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 computation time of $T_e$; however, $c_e$ continues to denote the original length of computation or the value of $T_e$. Let us also call the interval starting at $d_e - c_e$ and ending at $d_e$ the *Lastest start interval* of $T_e$ and denote it by $\delta_e$. 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 $\Delta_e$.

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 necessary 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 Main Algorithm

As noted earlier, our algorithm D* is a variation of the well-known earliest-deadline-first (D) algorithm (see appendix A). During the period when the system is underloaded, the algorithm D* behaves like the algorithm D. However, when the system is overloaded, the algorithm D*

systems, called a *firm* real-time system, in which there is no value for a task after its deadline is missed, but there is no catastrophe either.

Also, we shall make the following assumption:

**Assumption 2.1**

1. UNIFORM VALUE-DENSITY: The value of a task is directly proportional to its computation time and every task is of finite length. This value is obtained provided the task is completed before its deadline; there is no value to completing a task once its deadline has passed.
2. TASK MODEL: 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, then, to make *on-line* scheduling decisions that will maximize the overall value of the resulting computation, even in the presence of overload. Note that maximizing the value may not minimize the number of missed deadlines, since many low-value tasks may be abandoned in favor of a single high-value task.

We must note that obtaining a scheduler that maximizes the overall value is a difficult problem. Even for a clairvoyant scheduler, i.e. one that satisfies the assumption 2.1 but knows all the parameters (including the request times) of the task set *a priori*, the problem of finding the maximum achievable value for such a scheduler can be shown to be reducible from the knapsack problem[4]; hence NP-hard. Thus, we shall aim for an on-line scheduler that only guarantees a performance bound.

## 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 be able 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 it exists. 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. Note that a task $T_{i_j}$ as above need not correspond to the original task set, because in D, preempted tasks may be rescheduled, and thus considered as new tasks.

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. D* 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. Our experimental data indicate that the proposed algorithm is comparable to Locke's best effort scheduler for a wide class of environments, while guaranteeing the bound even for the pathological cases that would cripple Locke's scheduler.

As in [5, 14], we say that an on-line algorithm has a *competitive factor* of $r$, $0 \leq r \leq 1$, if and only if it is guaranteed to achieve a cumulative value at least $r$ times the cumulative value achievable by a clairvoyant algorithm on *any* sequence of task requests. We show that, in the *uniform value-density* case[2], no on-line algorithm can have a *competitive factor* greater than 1/3.224745. Baruah et. al.[2] prove a tighter bound on the *competitive factor*. The bound, for an *importance ratio* of $k$, is $1/[(1+\sqrt{k})^2]$; in the *uniform value-density* case ($k = 1$), the corresponding lower bound is thus $1/4$, improving the bound of 1/3.224745. Also in [2], a further refined lower-bound has been obtained when one accounts for the *loading factor*, $b$; specially, when the loading factor is in the range $1 \leq b < 2$. Note that D*'s overload-guarantee does not correspond to a *competitive factor*; in fact, D*'s *competitive factor* can be shown to be equal to 0. (See the conclusion section at the end of the paper).

In the literature, there have been a few other methods proposed to deal with overloading. One translates aperiodic tasks into equivalent periodic tasks based on the worst case frequency of events. This approach yields a schedule with the equivalent periodic tasks included. (See, Sha, Lehoczky and Rajkumar [13].) But this model still requires knowing the worst case frequency of events which may lead to very pessimistic algorithm designs. Another approach suggested by Baker and Shaw [1] is to allow some slack time in each cycle of the cyclic executive. This approach fails to provide any performance guarantee, though it may be a good heuristic.

The rest of the paper is organized as follows: In section 2, we introduce the notations, assumptions and definitions used in the paper. In section 3, we describe the main algorithm, D*, with a proof of the theorem that demonstrates that our algorithm achieves the guarantees mentioned earlier. Section 4 provides a lower bound of 3.224745 for the competitive factor for any on-line algorithm. Section 5 describes the experimental results, and is followed by a brief conclusion.

## 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 [11]). Hard real-time systems are those whose deadlines must *absolutely* be met or the system will be considered to have failed (and whose failure might be catastrophic); soft real-time systems are those whose deadlines may occasionally be missed with only a degradation in performance, but not complete failure. Thus in a soft real-time system, a task that completes after its deadline (sometimes, called its *critical time*[10]) still has some (perhaps, nonzero) value. In this paper, we consider a special case of soft real-time

[2]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. When the importance ratio is 1, tasks are said to have *uniform value density*.

# 1 Introduction

There has been a renewed interest in the problem of process scheduling in real-time computer systems, primarily motivated by such demanding applications as spacecraft control, robotics systems and factory automation. Currently available operating systems for these applications assign priorities to tasks based on either their frequencies (the strategy used in rate-monotonic schedulers) or their perceived importance, and the scheduler attempts to execute the currently ready highest priority task. The hope is then that given fast computer systems and an appropriately chosen priority scheme, the system will be able to meet the deadlines of all tasks. However, even for a very well modeled system, this hope may not be realized.

Most practical systems are prone to intermittent overloading, primarily caused by cascading of exceptional situations (i.e., the situations that are left unmodeled): For instance, if a robot arm carrying a pay-load hits an unknown obstacle, several sensors in the arm will simultaneously indicate exceptionally high values, all requiring immediate attention. While the real-time system may be more than adequate for the normal operation of the system, it may fail to meet most of the deadlines associated with the tasks under such a strenuous situation, with a catastrophic outcome. The same holds true even for the so-called soft real-time systems, such as telephone networks or financial markets.

In fact, the theoretically sound scheduling algorithms that have been proposed for real-time systems work well only when the system is underloaded. Examples of such algorithms include: *shortest-process-first* (SPT), *earliest-deadline-first* (D), *smallest-slack-time* (SL), *oldest-process-first* (FIFO), *maximum-value-density-first* (VD), and several others (see, [6]). There have been some attempts made to devise heuristics to handle the overloading situations intelligently: the most important results in this direction come from Locke's work in the CMU Archons project[10]. While Locke's heuristic has been shown to perform well in practice, there has been no clear explanation of its power. In fact, no guarantee of its ability to deal with overload conditions has been put forth.

Roughly, Locke's *best effort* (BE) heuristic works as follows: if the system is underloaded then BE operates like the earliest-deadline-first algorithm; however, if an overloaded condition is detected then BE attempts to abandon some of the less important tasks in an attempt to bring the system back to the underloaded state. The complete algorithm is much more sophisticated, since it deals with a stochastic system and provides a policy-mechanism-separation by letting the user influence how some tasks are abandoned.

A particularly interesting scenario occurs when we assume an on-line deterministic model (the computation-time of each task is known precisely when the task is released), where the importance of every task is proportional to its running time. In such a situation, the BE algorithm may be considered to be simply a hybrid of an earliest-deadline-first scheduler and a random scheduler. Even for such a simple model, the BE heuristic does not provide a performance guarantee.

In this paper, we formalize the notions of when a system is overloaded and when it is underloaded. We devise a scheduling algorithm called D$^*$, that provides the following performance guarantee. Informally [1] 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

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

# On-Line Schedulers for Overloaded Real-time Systems*

G. Koren † B. Mishra † A. Raghunathan ‡ D. Shasha †

Abstract

*There are several scheduling algorithms proposed for real-time systems with sporadic tasks that exhibit optimal behavior if the system is underloaded. Examples of such scheduling algorithms include* earliest-deadline-first *(D),* smallest-slack-time-first *(SL), etc. However, these algorithms perform poorly when the system is overloaded. This is a serious problem, since even the best-modeled systems may not be able to avoid intermittent overloading situations.*

*In this paper, we propose an on-line scheduling algorithm for a uniprocessor in an environment of sporadic tasks that addresses this issue. We assume that the value of each task is proportional to its computation time provided it completes by its deadline; if it misses its deadline, its value is zero. An optimal scheduling algorithm, then, maximizes the sum of the values of the completed tasks. Even for a clairvoyant model, this problem is NP-hard. In this paper, we formalize the notions of when a system is overloaded and when it is underloaded. We devise a scheduling algorithm called $D^*$, that provides the following performance guarantee. Informally 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 the best of our knowledge, this is the only algorithm that makes such a non-trivial guarantee for a real-time system that may face intermittent overloading. We implement $D^*$, using simple data structures, at a cost of $O(\log n)$ time per task, where $n$ bounds the number of tasks in the system at any instant.*

*We also show that no on-line algorithm can guarantee a value that is better than 1/3.224745 times the value obtained by a clairvoyant algorithm.*

*Our experimental data indicate that the proposed algorithm (referred to as $D^*$ algorithm) is comparable to Locke's* best effort scheduler *(BE) in the uniform arrival case, while guaranteeing the one-fifth bound even for the pathological cases that would cripple Locke's scheduler.*

April 30, 1991

*Supported by ONR grants #N00014-89-J3042, #N00014-90-J-1110, ONR N00014-91-J-1472 and NSF grants #CMU-406349-55586, #CCR-9002819, #IRI-89-01699, #IRI-9003986.

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

‡ University of California at Davis, CA 95616 and Xerox PARC, Palo Alto, CA 94304.