Then, the effective value density (i.e, the value achieved divided by the period's length) obtained by the clairvoyant scheduler between t_l and t_{l+1} is at least $\frac{k}{(k-1)}n(k^{\frac{1}{n}}-1)$ times bigger than the effective value density obtained by the on-line scheduler for the same period.

PROOF.
Suppose $T_1, T_2, \cdots T_m$ are the tasks scheduled at t_l with value densities $i_1 \leq i_2 \leq \cdots i_m < select(t_l)$. Denote $select(t_l)$ by i_0.

The clairvoyant algorithm schedules n tasks of level i_0, there are no more tasks releases until the end of i_0's period. Hence the ratio between the effective value densities is at least:

$$\frac{nk^{\frac{i_0}{n}}}{(k^{\frac{i_1}{n}} + k^{\frac{i_2}{n}} + \cdots + k^{\frac{i_m}{n}} + (n-m)\epsilon)}$$

But, if we replace the value density of a task T by $f(T)$ (f as in lemma B.2) then the denominator does not decrease hence the ratio does not increase.

$$\begin{aligned} & \frac{nk^{\frac{i_0}{n}}}{(k^{\frac{i_1}{n}} + k^{\frac{i_2}{n}} + \cdots + k^{\frac{i_m}{n}} + (n-m)\epsilon)} \\ \geq & \frac{nk^{\frac{i_0}{n}}}{(k^{\frac{f(i_1)}{n}} + k^{\frac{f(i_2)}{n}} + \cdots + k^{\frac{f(i_m)}{n}} + (n-m)\epsilon)} \\ \geq & \frac{nk^{\frac{i_0}{n}}}{(k^{\frac{1}{n}} + k^{\frac{2}{n}} + \cdots + k^{\frac{m}{n}} + (n-m)\epsilon)} \end{aligned}$$

We saw, in lemma 3.1 above, that the above ratio is not smaller than

$$\frac{k}{k-1}n(k^{\frac{1}{n}}-1)$$

and the lemma is proved.

□

□

Lemma B.2 *At time t, let $T_1, T_2, \cdots T_m$ be the tasks of levels 0 to $select(t) - 1$ scheduled by the on-line scheduler. Then, there is a one to one correspondence f from these tasks to all the density levels from 0 to $select(t) - 1$ so that $f(T)$ is greater than or equal to the value density of T. In other words, we can promote all tasks with level density below $select(t)$ to a higher (or equal) value density (but still below $select(i) - 1$) so that after the promotion, only one task corresponds to any given value density level.*

PROOF.
Assume that the tasks $T_1, T_2, \cdots, T_m$ are ordered by non-decreasing order of value densities. Define f as follows,

$$f(T_r) = r - 1$$

(That is, $f(T_r)$ is the position in the list starting from zero)

We know that,

$$\begin{aligned} 1 &= \Phi(select(t)) \\ &= select(t) + 1 - \sum_{j=0}^{select(t)-1} present(j) \\ &= select(t) + 1 - m \end{aligned}$$

Therefore, $select(t) = m$. Hence, f is a one to one function onto $[0..(select(t) - 1)]$.

We still have to show that for every T_r, $f(T_r)$ is at least the value density level of T_r. If the level of T_r is 0 then there is nothing to prove since $f \geq 0$, so assume that T_r's level is $i + 1$ for some i.

$$0 \geq \Phi(i) = i + 1 - \sum_{j=0}^{i} present(j)$$

Therefore, $\sum_{j=0}^{i} present(j) \geq i + 1$. Hence, there are at least $i + 1$ tasks with value densities below T_r's value density. This means that T_r's position in the ordered list is not before the $i+2$'th place, so $f(T_r) \geq (i + 2) - 1 \geq i + 1$.

□

Now we are ready to state and prove the version of lemma 3.1 for the general case (i.e., when more than one level is missing).

Lemma B.3 *Suppose the on-line scheduler gets partial value for tasks that were executed but not to completion. In particular, for a task that executed for α $(0 \leq \alpha \leq 1)$ of its computation time the scheduler will get α times the task's value.*

B APPENDIX: The Complexity Bound; Few Lemmas

In section 3 we proved the complexity result in the special case that only one value density level is missing from the on-line schedule. But what will happen if more than one level is missing? In this appendix, we will show that this can not benefit the on-line scheduler Hence, the complexity bound holds in the general case.

Definition B.1 PRESENT(t), $\Phi(i)$, *select*(t)

For any time t, look at the tasks that are currently being executed by the on-line scheduler. For each density level $i, 0 \geq i \geq n$:

Define *present*(i) to be the number of tasks from level i that are executing at time t.

Define the following potential function Φ:

$$\Phi(i) = (i+1) - \sum_{j=0}^{i} present(j)$$

Define *select*(t) to be the minimal index i for which $\Phi(i) \geq 1$ (Note that, $\Phi(0) \leq 1$ and $\Phi(n) \geq 1$). □

We will start by proving two properties of Φ and *select*.

Lemma B.1 *For all t,*

1. *present*(*select*(t)) = 0 *(i.e., No task of level select*(t) *is executing at time t).*
2. Φ(*select*(t)) = 1 *(i.e, equals one rather then greater than 1).*

PROOF.
There are two cases: *select*(t) = 0 and *select*(t) > 0.

- *select*(t) = 0.

  From the definition of Φ, $\Phi(0) \leq 1$. But, *select*(t) = 0 hence $\Phi(0) \geq 1$. We conclude that $\Phi(0) = 1$ which implies that *present*(0) = 0.

- *select*(t) > 0.

  Let *select*(t) = $i+1$ (for $i \geq 0$). Note that by definition,

  $$\Phi(i+1) = \Phi(i) + 1 - present(i+1) \tag{6}$$

  Assume, in order to get a contradiction, that *present*($i+1$) ≥ 1 then $\Phi(i) \geq \Phi(i+1) \geq 1$. Therefore, $\Phi(i) \geq 1$. This is a contradiction to the fact that *select*(t) is the minimal index i with $\Phi(i) \geq 1$. We conclude that *present*($i+1$) = 0.

  Substituting *present*($i+1$) = 0 in (6) above gives $1 \leq \Phi(i+1) = \Phi(i) + 1$. Since $\Phi(i) \leq 0$ (otherwise $i+1$ is not minimal), $\Phi(i+1) = 1$.

PROOF.

All tasks that were abandoned by the *Safe-Risky-(fixed)* were diverted to RP. Let d be the maximal deadline among all tasks of F [21]. The best the clairvoyant can do, with tasks of F, is to schedule both processors continuously to get a value of $2d$. Lemma A.2 shows that the *Safe-Risky-(fixed)* gets a value of at least d.

□

Theorem A.9 *For any interval I the competitive multiplier of the Safe-Risky-(fixed) algorithm is less than or equal to 3. I.e,*

$$C(I) \leq 3V(I)$$

PROOF.
This follows from the Lost Value Lemma (4.1) and lemma A.8.

□

[21] If there is no such maximal deadline. Then the supremum is either finite or infinity. If it is finite a similar proof will work. if it is infinity, then the interval is infinitely long, in which case the value obtained by the on-line algorithm is infinity.

A.3 When Task Migration is Not Allowed

By task migration, we mean moving a task that has already partially executed from one processor to another. In the case that task migration is not allowed, we can use a simple variant of the *Safe-Risky* algorithm to get a competitive multiplier of 3 for tasks with slack time[20]. We call this variant the *Safe-Risky-(fixed)*.

A.3.1 The Safe-Risky-(fixed) Algorithm

The only place that the *Safe-Risky* algorithm used the possibility of task migration is in the task completion routine. For that reason, the *Safe-Risky-(fixed)* differs from the *Safe-Risky* algorithm only in this routine.

```
42  Task Completion :
    (* on SP; There is no special event when a task completes on RP *)

43      let T be the task with earliest deadline in Q_waiting;
44      if ( T has earlier deadline than the current task on SP
            and can be scheduled with all other privileged tasks) then
45          remove T from Q_waiting;
46          add T to Q_privileged;
47          (* which amounts to scheduling T on SP *)
48      endif (*T can not be added to Q_privileged *)
49      schedule on SP the privileged task with earliest deadline (if any);
50      if (SP is idle) then
51          switch roles between the SP and RP processors;
52      endif (*SP is idle *)
53  end (*task completion *)
```

A.3.2 Analysis

We define *latest_affected* and all the other definitions as before. Observe that lemmas A.1 and A.2 hold for the new algorithm as well.

Recall, that F denotes the set of tasks abandoned by the on-line algorithm

Lemma A.8 *For every interval, I,*

$$C(F) \leq 2V(I)$$

[20] When tasks have no slack time an optimal algorithm with competitive multiplier of 2 is known [12,3].

Lemma A.6 *For any interval I, the Safe-Risky algorithm guarantees that the competitive multiplier can not be greater than 2. I.e,*

$$C(I) \leq 2V(I)$$

PROOF.
The proof is by induction on the structure of the interval.

Basis: If the interval is *atomic_interval* then lemma A.4 ensures that the induction hypothesis holds.

Induction step: Suppose the induction hypothesis is known for all the sub-intervals $I_1, I_2, \cdots, I_m$ of I (after removing the tasks of *uncovered*(I)). By the induction hypothesis, $C(I_j) \leq 2V(I_j)$ for all j.

But[19],

$$\begin{aligned} C(I) &= C(I_1 \cup I_2 \cup \cdots \cup I_m \cup uncovered(I)) \\ &\leq C(I_1) + C(I_2) + \cdots + C(I_m) \cup C(uncovered(I)) \\ &\leq 2V(I_1) + 2V(I_2) \cdots + 2V(I_m) + C(uncovered(I)) \\ &= 2V(I_1) + 2V(I_2) \cdots + 2V(I_m) + value(uncovered(I)) \end{aligned}$$

Lemma A.5 shows that the tasks of $uncovered(I)$ do not affect the scheduling of tasks in the sub-intervals (by the *Safe-Risky* algorithm). Hence,

$$V(I) = V(I_1) + V(I_2) \cdots + V(I_m) + value(uncovered(I))$$

This shows that the induction hypothesis holds for I, hence the theorem is proved.
□

Theorem A.7 *For a system of uniform value density and 2 processors, the Safe-Risky algorithm achieves the best possible competitive multiplier of 2 (when migration of tasks is permitted).*

PROOF.
The intervals partition the entire set of tasks Γ into disjoint subsets $\Gamma_1, \Gamma_2, \cdots$.

$$\begin{aligned} C(\Gamma) &\leq C(\Gamma_1) + C(\Gamma_2) + \cdots \\ &\leq 2(V(\Gamma_1) + V(\Gamma_2) + \cdots) \\ &= 2V(\Gamma) \end{aligned}$$

The last equality is due the fact that tasks of one interval do not influence the scheduling decisions (of the *Safe-Risky* algorithm) in another interval. This might not be the case for the clairvoyant algorithm, that is why the first inequality is not an equality.
□

[19] Recall that all the tasks of $uncovered(I)$ are scheduled to completion by the *Safe-Risky* algorithm. Hence, $C(uncovered(I)) = value(uncovered(I)) = V(uncovered(I))$.

- *Suppose the tasks of* uncovered *were never released (i.e, only the tasks of* covered *were released). Then, the tasks of* covered *will be scheduled (on SP or RP), preempted, and completed or abandoned at exactly the same times as if the tasks of both* covered *and* uncovered *were released.*
- *Removing* uncovered *from the original set of tasks will break the interval into one or more sub-intervals separated by idle time.*

PROOF.

- A task T is diverted to RP (and possibly abandoned) only if it is involved in a conflict with the tasks of *Q_privileged*. A task T of *uncovered* can not be involved in any conflict (otherwise, *max_cover* will be at least *deadline*(T) which contradicts the fact that T is in *uncovered*). We conclude that T was scheduled on SP (hence also completed).
- Suppose that T_{rp} is executing on RP. T_{rp} was scheduled at its latest start time. Let T be the first task of *uncovered* to be executed (on SP) concurrently with T_{rp}. Note that all the tasks in *Q_privileged* (at that point) must be of *uncovered* because these tasks are scheduled on SP according to deadline order and the tasks of *covered* have earlier deadlines.

  When was T scheduled? It could not be before $LST(T_{rp})$ because this means that (at its latest start time) T_{rp} had a conflict with tasks of *uncovered*. So, T must have been scheduled after $LST(T_{rp})$ as a consequence of a task completion event[18]. But, if this is the case, all the tasks of *Q_privileged* could be scheduled on one processor along with T_{rp}. Hence the *Safe-Risky* algorithm (in the Task-Completion routine) would have scheduled T_{rp} and then all the privileged tasks on SP, a contradiction.
- The execution of a task of *covered* on SP can be delayed or interrupted (i.e, preempted) only by tasks of *covered* (because all the tasks of *uncovered* have later deadlines). Hence, the execution of *covered* tasks on SP is not affected by *uncovered* tasks.

  A task of *covered* can not be diverted to RP as a consequence of a conflict with a task of *uncovered* (because the tasks of *uncovered* had no conflicts). Also, no task of *uncovered* will be scheduled on RP. This implies that the execution of *covered* tasks on RP is not at all affected by *uncovered* tasks.
- We saw above that removing the tasks of *uncovered* will not change the execution history of all other tasks. Also, when a task of *uncovered* is executing on SP, RP is idle. Hence, periods in which only tasks of *uncovered* execute in the execution of (*covered*$\cup$ *uncovered*) will correspond to idle periods in the execution of *covered*.

□

[18] It can not be a consequence of a task release event because a task of *uncovered* can preempt only tasks of *uncovered*, in which case the task that precede T, on SP, is the first task of *uncovered* to coincide with T_{rp}.

later added to *Q_privileged*). The length of the interval is at least t_{actual}. Lemma A.1 shows that the value obtained on SP is at least this length.

As mentioned before, the value obtained by the *Safe-Risky* algorithm on RP is at least $value(T)$. Hence,

$$\begin{aligned} V(I) &= VSP(I) + VRP(I) \\ &\geq t_{actual} + computation_time(T) \\ &\geq deadline(T_{affected}) \\ &= latest_affected(T) \end{aligned}$$

The second inequality restates the definition of *latest_affected*(T).

□

Corollary A.3 *The value obtained by the Safe-Risky algorithm during an interval is at least*

$$\max\{the\ interval\ length, \max_cover\}$$

□

Definition A.6 COVERED, UNCOVERED, ATOMIC_INTERVAL

For any interval I, let $uncovered = uncovered(I)$ be the set of tasks (released in I) with deadlines after *max_cover*(I). Denote the remaining tasks by $covered = covered(I)$. I is called an *atomic_interval* if $uncovered(I) = \emptyset$ (the empty set).

□

Lemma A.4 *For an atomic_interval I, the Safe-Risky algorithm achieves a competitive factor of 2. That is,*

$$C(I) \leq 2V(I)$$

PROOF.

All tasks have deadlines before *max_cover*. So the best the clairvoyant algorithm can do is to schedule tasks on both processors from time 0 to *max_cover* to get a value of $2 \cdot max_cover$. Corollary A.3 ensure that the *Safe-Risky* algorithm gets a value of *max_cover* at least. □

Lemma A.5 *For every non atomic_interval, I,*

- *All the tasks of uncovered are executed to completion on SP.*
- *When a task of uncovered is executing on SP, RP is idle.*

Definition A.5 CONFLICTED TASK, MAX_COVER

We say that T is a *conflicted task if it conflicted with the tasks of Q_privileged (when it was released or when considered to execution at its latest start time) and hence was diverted to RP (and later was either scheduled to execution or was abandoned)*

Define max_cover as,

$$max_cover = max_cover(I) \stackrel{\text{def}}{=} \max_{T \text{ is a conflicted task}} cover_end(T)$$

If there is no conflicted task this max equals 0.
□

Lemma A.2 *For every task T, if T is a conflicted task (i.e, was diverted to RP by the Safe-Risky algorithm) then*[17]*, the total value obtained by the Safe-Risky algorithm is at least cover_end(T)*

PROOF.
Recall that,

$$cover_end(T) = \max\{deadline(T), latest_affected(T)\}$$

.

- First, let us show that $V(I) \geq deadline(T)$.

  T is diverted to RP at its latest start time which is at $deadline(T) - computation_time(T)$. This means that the interval length is at least $deadline(T) - computation_time(T)$ (recall that the interval starts at time 0). Lemma A.1 says that the value obtained by the *Safe-Risky* algorithm on SP is at least $deadline(T) - computation_time(T)$.

  If T does not complete, it can be abandoned only in favor of a bigger valued task (scheduled on RP), hence the value obtained on RP is at least $value(T)$. Hence,

$$\begin{aligned} V(I) &= VSP(I) + VRP(I) \\ &\geq (deadline(T) - computation_time(T)) + value(T) \\ &= (deadline(T) - computation_time(T)) + computation_time(T) \\ &= deadline(T) \end{aligned}$$

- Now, we are going to show that $V(I) \geq latest_affected(T)$.

  By definition, *latest_affected*(T) is the deadline of some task $T_{affected}$. Let t_{actual} be the time when $T_{affected}$ completed (according to the *Safe-Risky* algorithm). t_{actual} can not be any earlier than the estimated completion time of T computed when T was considered for execution (the actual time can be, in fact, later than the estimated time if some tasks were

[17] Recall that the interval starts at time 0.

By definition the tasks of *Q_privileged* constitute an *underloaded* subsystem (i.e, all these tasks can complete on one processor).

Now, let us consider one additional task, T, which has a deadline earlier than all the tasks in *Q_privileged*. Suppose that the system comprised of *Q_privileged* plus T is *overloaded*. We want to know which of the tasks of *Q_privileged* conflict with T.

Compute the *earliness* of all the tasks in *Q_privileged*. That is, suppose the tasks are scheduled according to *earliest-deadline-first* scheduling algorithm, the earliness of a task is how much before its deadline it completes (i.e, its deadline minus its completion time). Any task with earliness smaller than T's computation time in *Q_privileged* prevents T from being added to *Q_privileged*.

Define *latest_affected*(T) to be the latest deadline of any such task (i.e, a tasks whose earliness is smaller than T's computation time) and define *cover_end*(T) to be,

$$cover_end(T) \stackrel{\text{def}}{=} \max\{deadline(T), latest_affected(T)\}$$

□

Definition A.3 INTERVAL

An *interval* starts when a task is released to an idle system (the system becomes non-idle) and ends when the system becomes idle again (both processors are idle).

□

In the following, we are going to analyze the *Safe-Risky* algorithm interval by interval. For the sake of notational convenience let us assume that there was only one interval (call it I) and it started at time 0.

Lemma A.1 *The value obtained by the Safe-Risky algorithm from tasks executed on SP during an interval is at least the interval length (duration).*

PROOF.
From the beginning of the interval to its end SP is never idle (when SP is idle so is RP which means that the interval ends). A task that was scheduled on SP is guaranteed to complete. We conclude that at any given moment at least one processor is working on a task that will eventually complete. This proves the lemma.

□

Notation A.4 C(I), V(I), VSP(I), VRP(I)

For an interval I, let $C(I)$ and $V(I)$ denote the value obtained by the clairvoyant algorithm and the *Safe-Risky* algorithm (respectively) from tasks released during I. Similarly, let *VSP* and *VRP(I)* denote the portion of $V(I)$ obtained by the *Safe-Risky* algorithm on SP and RP respectively.

□

```
27  Task Completion :
    (* on SP; There is no special event when a task completes on RP *)

28      let T be the task with earliest deadline in Q_waiting;
29      if (T has earlier deadline than the current task on SP
            and can be scheduled with all other privileged tasks) then
30          remove T from Q_waiting;
31          add T to Q_privileged;
32          (* which amounts to scheduling T on SP *)
33      endif (*T can not be added to Q_privileged *)
34      if (all the privileged tasks can complete with the
            task currently executing RP) then
35          preempt the current task on RP;
36          add it to Q_privileged;
37          (* this task will be scheduled on SP by the following piece of code. This
               is the only place where migration is used *)
38      endif
39      schedule on SP the task with earliest deadline in Q_privileged (if any);
40  end (*task completion *)
41 end {loop}
```

Remark A.1 Note an important difference between the above algorithm (and for that matter also D^{over} [7]) and the version of the *Cascading Multi-Band Algorithm* presented earlier. In the task release routine a newly arrived tasks will be added to *Q_privileged* only if its deadline is earlier than the currently executing task and also, it can be added to *Q_privileged* without creating overload (we call that a *local schedulabilty test*. This stands in contrast to the *global schedulabilty task* used by the *Cascading Multi-Band Algorithm*. That is: a new task enters *Q_privileged if and only if* it does not create overload irrespectively of its deadline.

A.2 The Competitive Multiplier of the *Safe-Risky* algorithm

In this section we show that the *Cascading Multi-Band Algorithm* has a competitive multiplier of 2. Our approach is to partition the execution into disjoint intervals, the *Cascading Multi-Band Algorithm* gets at least half the optimal value in each interval. This is proved by means of "covering". We show that all intervals are covered; if a period is covered then the *Cascading Multi-Band Algorithm* uses at least one processor effectively during all that period. The clairvoyant scheduler could gain a factor of two by utilizing both processors.

Definition A.2 EARLINESS, LATEST_AFFECTED, COVER_END

```
4 loop :
5    Task Release :  (* T is released *)
6        if (SP is idle ) then
7            schedule T on SP;
8        else if ( T has earlier deadline than the current task on SP
                   and can be scheduled to completion with the current task
                   as well as with all other privileged tasks) then
9                    preempt current task;
10                   add the current task to Q_privileged;
11                   schedule T on SP;
             else    (*SP is not idle and T can not be scheduled on SP *)
12               add T to Q_waiting;
         endif
13   end (*task release *)

14   LST:
     (* T reached its LST; LST denotes the latest start time of T, i.e., the
     moment when the computation time remaining for T equals the time
     to its deadline
     Note that scheduling T on SP will cause overload (with all the other
     privileged tasks)                                                  *)
15       if (RP is idle) then
16           schedule T on RP;
17       else     (*RP is not idle and T can not be scheduled on SP *)
18           let T_RP be the current task on RP;
19           if (value(T) > value(T_RP)) then
20             abandon T_RP;
21             schedule T on RP;
22           else
23             abandon T;
24           endif (*comparing values *)
25       endif (*RP is not idle *)
26   end (*LST *)
```

A APPENDIX: Two Processor Systems

This section deals with systems having only two processors. We assume *uniform value density* (i.e., $k = 1$). With out loss of generality we can assume that the value density is normalized to 1. We consider two possible models of multiprocessor systems. In the first model, tasks can *migrate* cheaply (and quickly) from one processor to another. Hence, if a task started to execute on one processor it can later continue on any other processor (and migration takes no time). We present a scheduling algorithm called the *Safe-Risky* algorithm for this model. In the second model (the *fixed* model), once a task starts to execute on one processor it can not execute on any other processor. An on-line scheduler can do better when migration is possible.

A.1 The Safe-Risky Algorithm

In this algorithm, one processor is designated as the *Safe Processor* (SP) and the other as the *Risky Processor* (RP). A task that started to execute on SP is called "privileged" because it is guaranteed to complete (*Q_privileged* is a queue containing these tasks).

Ready tasks that are not privileged wait in *Q_waiting* until they become privileged or they reach their LST, at that point the LST task tries to be scheduled on RP. It will be scheduled if it has a bigger value than the task currently executing on RP. A task that started to execute on RP can be preempted and then resume on SP (i.e., migrate to SP). In this version of the algorithm the designation as safe or risky processor is fixed. In the no-migration version (the *Safe-Risky-(fixed)*) the processors may switch roles.

The following few boxes depict the code of the *Safe-Risky* algorithm:

```
(* the Safe-Risky algorithm: a competitive scheduling algorithm for two
processors systems in the uniform value density case                  *)

1   Initilzation :
    (* Q_waiting and Q_privileged are initialized to the empty queue. One
    processor is designated as the safe processor (SP) and the other one is
    designated as the risky processor (RP).
    In the beginning both processors are idle.                        *)

2   Q_privileged := φ;
3   Q_waiting    := φ;
```

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

[12] F. Wang and D. Mao, Worst Case Analysis for On-Line Scheduling in Real-Time Systems. *Technical Report 91-54, Department of Computer and Information Science,* University of Massachusetts at Amherst, 1991.

[13] Hongyi Zhou, Karsten Schwan and Ian F. Akyildiz Performance Effects of Information Sharing in a Distributed Multiprocessor Real-Time Scheduler, *In Proc. of 1992 IEEE Real-Time Systems Symposium* , Phoenix, Arizona, pp. 46-55, December 1992.

An important issue is how to account for migration overhead in such a system. For example, we modeled NUMA architectures by forbidding migration but that is clearly too strong a restriction. Permitting migration, but at a cost, would have been much more reasonable.

There are many other open issues that must be addressed for overloaded systems in a fault-tolerant context. An important issue is how to reallocate processors when a failure occurs. The fact that the algorithms here execute competitively for any number of processors does not mean they will execute competitively if the number of processors changes during the computation. Handling such changes in an environment where failures can occur with high frequency is still to be done.

8 Acknowledgments

We would like to thank Bud Mishra for helpful discussions and insights and Doug Locke for practical enlightenment.

References

[1] A. Bar-Noy, Y. Mansour and B. Schieber, I.B.M T.J. Watson Research Center private communication, 1992

[2] S. Baruah, G. Koren, B. Mishra, A. Raghunathan, L. Rosier and D. Shasha. On-line Scheduling in the Presence of Overload,. *Proc. of IEEE Foundations of Computer Science Conference, pp. 101-110*, San Juan, Puerto Rico, October 1991.

[3] S. Baruah, G. Koren, D. Mao, B. Mishra, A. Raghunathan, L. Rosier, D. Shasha and F. Wang. On The competitiveness of On-line Task Real-Time Task Scheduling,. *The Journal of Real-Time Systems*, 4, 124-144, 1992. *Also, in Proc. of 1991 IEEE Real-Time Systems Symposium* , San Antonio, Texas, pp. 106-115 December 1991.

[4] M. Dertouzos and A. Mok. Multiprocessor Scheduling in a Hard Real-Time Environment. In *Proc. 7th IEEE, Conference on Computing Systems, Texas* ,1978.

[5] 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*

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

[7] G. Koren, D. Shasha. D-over: An Optimal On-Line Scheduling Algorithm for Overloaded Real-Time Systems, *In Proc. of 1992 IEEE Real-Time Systems Symposium* , Phoenix, Arizona, pp. 290-299, December 1992.

[8] G. Koren, D. Shasha. An Optimal Scheduling Algorithm with a Competitive Factor for Real-Time Systems,. *Technical report no. 572, Courant Institute, NYU* , July 1991.

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

[10] Wei-Kuan Shih private communication, 1993

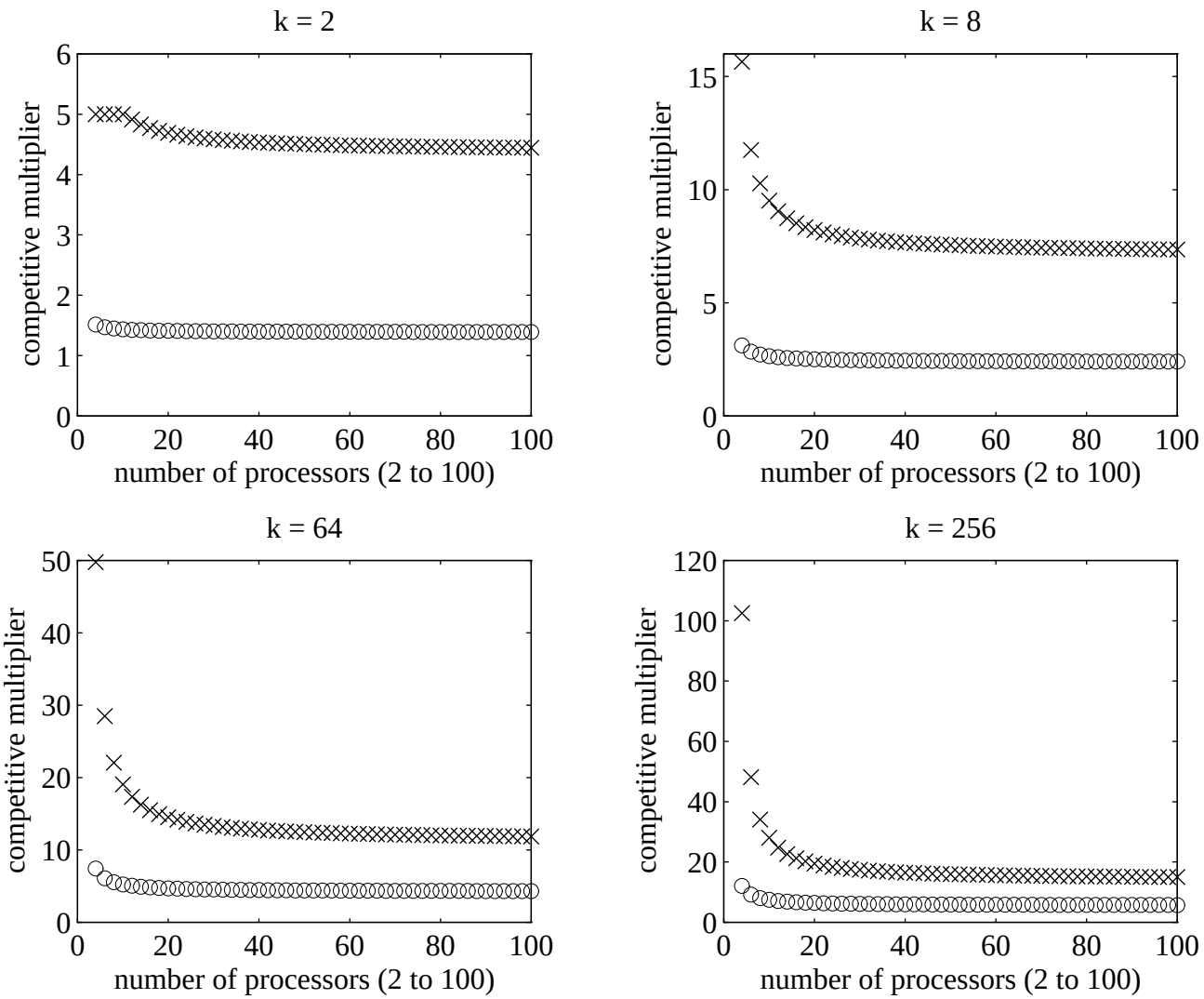


Figure 2: Comparison of competitive multipliers of complexity (x's) and algorithmic bounds (o's) for fix k's and varying number of processors.

7 Conclusion

number of processors	importance ratio	bounds: complexity	bounds: algorithmic	comments
1	: any $k \geq 1$	$(1+\sqrt{k})^2$	tight	tight bound achieved by D^{over} [7].
2	: 1	2	tight	tasks have *no* slack time and may not migrate between processors [3,12].
2	: 1	2	3 ‡	tasks may have slack time but may not migrate between processors.
2	: 1	2	tight ‡	tasks may have slack time and may migrate between processors.
n	: $k > 1$:	$\frac{k}{(k-1)} n(k^{\frac{1}{n}} - 1)$ ‡	$1 + n(k^{\frac{1}{\psi}} - 1)$ ‡ where $\psi = \frac{n}{2} \frac{\ln k}{\ln k + 1}$	general algorithmic guarantee can be seen in theorem 6.2
$n >> 2$	: $k > 1$	$\frac{k}{(k-1)} \ln k$ ‡	$2 \ln k + 3$ ‡	asymptotic behavior

The above table summarizes the current state of the art of competitive real time scheduling. Here, n is the number of processors in the system; k is the *importance ratio*, that is the highest possible value per unit of computation time that any task can possibly obtain (normalizing the lowest to 1). The bounds are expressed in terms of *competitive multipliers*. Results marked with ‡ are part of this paper.

A gap remains between the guarantees achieved by the *Cascading Multi-Band Algorithm* and the complexity bounds we have proved. The algorithmic guarantee is within a multiplicative factor of 2 from the complexity bound for large enough n and k (the graphs on the next page show that the asymptotic behavior is attained even for small values of n). When the importance ratio of a system (i.e., k) is close to 1, a different treatment is needed. Some work in this direction has been done by Bar-Noy et. al. [1].

It is possible that a better choice of ψ will lead to a better exact expression of the algorithmic guarantee for our algorithm. But it seems that, asymptotically we can not do better without changing our algorithmic techniques. The reason is that our basic block, the scheduling algorithm for a 2-processor band concentrates its efforts on one processor at a time (SP); the other processor RP, is essentially left idle. Hence, the *Cascading Multi-Band Algorithm* automatically loses a factor of 2 compared to a clairvoyant scheduling algorithm that utilizes *all* the processors concurrently. Of course, one can suggest heuristics that will use a processor whenever possible[16], the true challenge is to show that such a heuristic achieves a better worst case performance guarantee. Another way to improve the algorithmic guarantee will be to come up with a better algorithm for an m-processor band (for some $m \geq 2$).

[16] Heuristic improvements can be obtained by looking ahead in the waiting and privileged queues and by scheduling tasks on the Risky Processor before tasks arrive at their latest start times.

6.3 The *Cascading Multi-Band Algorithm* Scheduling Overhead

In the previous sections we analyzed the performance of our algorithms in the sense of their competitive multipliers. In this section we study the cost of executing the scheduling algorithms themselves.

What is the cost of testing whether a newly arriving task can be added to *Q_privileged* containing N tasks with out causing overload? This can be done in $O(\log N)$ operations using a 2-3 tree that holds slack times with sums of the slack times from left siblings held in interior nodes. If the task is to be added to *Q_privileged* the updating of the 2-3 trees involved takes also $O(\log N)$ time.

Let M be a bound on the the total number of ready tasks at any given moment in *Q_waiting* and any of the local queues.

When a task is released it may have to be checked against all bands (suppose the task cascades from the highest band all the way to the lowest) with a total cost of $O(n \log M)$.

A task in *Q_waiting* awaits its LST. Hence, *Q_waiting* is a 2-3 tree organized according to Latest Start Time. Inserting and removing a task from this queue costs $O(logM)$ operations.

A task during its lifetime causes exactly one task release event and at most one LST interrupt. Hence, the scheduling overhead per task is $O(nlogM)$.

6.4 Two Processor Systems

In this section we discuss the case of uniform value density (i.e., $k = 1$). Wang and Mao [3,12] showed a lower bound of 2 (on the competitive multiplier) and presented an algorithm that achieved this bound, assuming tasks have no slack time. Building on this work we have developed an algorithm, the *Safe-Risky* algorithm that achieves a competitive multiplier of 2 even when tasks may have slack time but migration is allowed. Hence, the *Safe-Risky* algorithm is optimal. For the "No-Migration" model, a variant of the *Safe-Risky* algorithm, called the *Safe-Risky-(fixed)*, achieves a competitive multiplier[15] of 3. We do not know whether the second algorithm is optimal. For reasons of space we omit here the details of these algorithms and their analysis, these can be found in appendix A at the end of this paper.

[15] Tasks may have slack time.

Recall that $\psi(k^{\frac{1}{\psi}}-1)$ tends to $\ln k$ as ψ approaches infinity. Hence, when the number of processors tends to infinity, equation (4) above tends to

$$1+2\lim_{\psi\to\infty}\psi\frac{\ln k+1}{\ln k}(k^{\frac{1}{\psi}}-1)=1+2\frac{\ln k+1}{\ln k}\ln k=1+2(\ln k+1) \tag{5}$$

Corollary 6.4 *The ratio between the complexity bound and the algorithmic guarantee is* at most

$$\frac{1+2n(k^{\frac{1}{\psi}}-1)}{\frac{k}{k-1}2n(k^{\frac{1}{2n}}-1)}$$

When k is held fixed and n tends to infinity this ratio tends to $\frac{k-1}{k}\left(2+\frac{3}{\ln k}\right)$. Which, when k tends to infinity tends to a constant of 2.

PROOF.

Recall our complexity bound of $2n\frac{k}{k-1}(k^{\frac{1}{2n}}-1)$. This bound tends to $\frac{k}{k-1}\ln k$ when n tends to infinity. The limit of the ratio is the ratio of the limits which is:

$$\frac{1+2(\ln k+1)}{\frac{k}{k-1}\ln k}=\frac{k-1}{k}(2+\frac{3}{\ln k})$$

Which gives the desired result.

□

Remark 6.5 In the discussion above we have chosen to ignore the fact that ω and ψ must be integers. We can take care of that by setting ψ as the nearest integer to $\frac{\ln k}{\ln k+1}$

6.2 Distributed vs. Centralized Scheduler

We discuss here architectures with large number of processors. Hence, it is necessary to see which portions of the scheduler are centralized and which are distributed. The *Cascading Multi-Band Algorithm* uses a central scheduler in order to assign a task to a band (at task release time and LST). This means that the centralized scheduler has all the information regarding tasks assigned to each band and their parameters[14]. Once a task is assigned to a band it is left in the hands of the local scheduler (which basically employs *Earliest-Deadline-First*).

It is desirable for reasons of fault-tolerant and efficiency ([13]) to distribute the functionality of the centralized scheduler among the processors. This is an interesting and important extension to the work presented here.

[14]Since all the tasks go through the central scheduler this is not difficult to do.

Remark 6.2 Note that the *Cascading Multi-Band Algorithm* does not use migration, hence the previous result holds both whether migration is allowed or not.

Remark 6.3 When $k = 1$, there is no need for the bands' structure, hence the central pool consists of all the processors ($\omega = n$ and $\psi = 0$). This leads to a competitive multiplier of $2 + 1$.

Remark 6.4 When the number of processor is odd, a similar result can be obtained.

6.1 Setting ψ

In the following, we will estimate the upper bound in theorem 6.2 by setting[12] $\psi = n\frac{\ln k}{\ln k+1}$ (hence $\omega = n\frac{1}{\ln k+1} = \frac{\psi}{\ln k}$).

The bound in (2) above becomes:

$$2n \max_{1\le i\le\psi} \frac{k^{\frac{i}{\psi}}}{\frac{\psi}{\ln k} + \frac{k^{\frac{i}{\psi}}-1}{k^{\frac{1}{\psi}}-1}} = 2n(k^{\frac{1}{\psi}} - 1) \max_{1\le i\le\psi} \frac{k^{\frac{i}{\psi}}}{\frac{\psi}{\ln k}(k^{\frac{1}{\psi}} - 1) + (k^{\frac{i}{\psi}} - 1)}$$

The left hand side is obtained by multiplying both numerator and denominator by $(k^{\frac{1}{\psi}} - 1)$. Define $f_a(x)$ to be $\frac{x}{a+(x-1)}$. When $a > 1$, this function is monotone increasing with x ($x \ge 0$). For $a = \frac{\psi}{\ln k}(k^{\frac{1}{\psi}} - 1)$, a is bigger than[13] 1. This means that the maximum above is attained at $i = \psi$ and the upper bound (equation 2) is:

$$2n(k^{\frac{1}{\psi}} - 1)\frac{k}{\frac{\psi}{\ln k}(k^{\frac{1}{\psi}} - 1) + (k - 1)} < 2n(k^{\frac{1}{\psi}} - 1)\frac{k}{1 + (k - 1)} = 2n(k^{\frac{1}{\psi}} - 1) \tag{3}$$

We have just proved the following lemma:

Lemma 6.3 *The Cascading Multi-Band Algorithm has a competitive multiplier of* at most

$$1 + 2n(k^{\frac{1}{\psi}} - 1), \quad \textit{where } \psi = n\frac{\ln k}{\ln k + 1} \tag{4}$$

(recall that the complexity bound is bigger than $2n(k^{\frac{1}{2n}} - 1)$*)* □

[12] ln is the natural logarithm

[13] Because $\psi\frac{k^{\frac{1}{\psi}}-1}{\ln k}$ is a monotone decreasing function of ψ tending to 1 when ψ goes to infinity.

The facts that a task of level $high(t)$ was abandoned and that t is in its executable interval imply that at time t, all bands up to (and including) $high(t)$ were productive. This means that the on-line scheduler has a cumulative value density of at least[11]:

$$\omega + 1 + k^{\frac{1}{\psi}} + k^{\frac{2}{\psi}} + \cdots + k^{\frac{high(t)-1}{\psi}} = \omega + \frac{(k^{\frac{high(t)}{\psi}} - 1)}{(k^{\frac{1}{\psi}} - 1)}$$

This leads to the following theorem.

Theorem 6.2 *For a system with $2n$ processors and maximal value density of $k > 1$ the Cascading Multi-Band Algorithm has a competitive multiplier of* at most

$$1 + 2n \min_{(0 \le \psi \le n; n = \omega + \psi)} \left\{ \max_{1 \le i \le \psi} \frac{k^{\frac{i}{\psi}}}{\omega + \frac{(k^{\frac{i}{\psi}} - 1)}{(k^{\frac{1}{\psi}} - 1)}} \right\} \tag{1}$$

PROOF.

The discussion above demonstrated that,

$$\frac{C(F)}{V(\Gamma)} \le \max_{1 \le i \le \psi} \frac{2nk^{\frac{i}{\psi}}}{\omega + \frac{(k^{\frac{i}{\psi}} - 1)}{(k^{\frac{1}{\psi}} - 1)}} = 2n \max_{1 \le i \le \psi} \frac{k^{\frac{i}{\psi}}}{\omega + \frac{(k^{\frac{i}{\psi}} - 1)}{(k^{\frac{1}{\psi}} - 1)}} \tag{2}$$

Since this is true for any setting of ψ (provided that $n = \omega + \psi$), hence we get

$$\frac{C(F)}{V(\Gamma)} \le 2n \min_{(0 \le \psi \le n; n = \omega + \psi)} \left\{ \max_{1 \le i \le \psi} \frac{k^{\frac{i}{\psi}}}{\omega + \frac{(k^{\frac{i}{\psi}} - 1)}{(k^{\frac{1}{\psi}} - 1)}} \right\}$$

Using the Lost Value lemma we get the desired result.

□

[11] For the case $k = 1$, the *uniform value density case*, see remark 6.3 below.

PROOF.

If SP does not become idle before time d then by definition the band is productive between t and d. Otherwise, suppose SP becomes idle at time s, $t < s < d$, then there must be a task executing on RP at time s (because a task on RP can be abandoned only in favor of another task with a later deadline and no slack time). So, at time s RP becomes SP and it would not become idle before time d because the deadline of the current task is at least d (and it has no slack time).

□

Here are a few things to notice about the *Cascading Multi-Band Algorithm*:

1. At any band i, only tasks of level i or higher can be executed.
2. If a task T of level i is abandoned then band i and all lower bands (including the central pool) are productive during the entire executable period of T.

   PROOF.

   Let T be $T(r, c, d)$[10]. Upon T's release it was not accepted by any of the levels on or below i. This means that for each of these bands, the tasks currently in (the local) *Q_privileged* will execute at least until $d - c$ (otherwise T could become privileged). This proved that all bands are productive between r and $d - c$.

   However, $d - c$ is the LST of T. At its LST, T would not be scheduled only if every band (on or below the i'th) has a task currently executing on its RP with deadline after d. This means that all bands are productive between $d - c$ and d (lemma 6.1) Combining the two gives the desired result.

   □
3. Once a task starts to execute on some processor, it will never migrate to another processor.

At any given time t, consider all the tasks of F for which t is in their executable period. Let $high(t)$ be the value density level corresponding to the task with the highest value density among all these tasks.

Suppose the clairvoyant scheduler has to schedule only the tasks of F, and suppose it had chosen some optimal schedule for these tasks. At time t, the best the clairvoyant scheduler can hope for (looking only at time t) is to have all $2n$ processors executing tasks of level $high(t)$, i.e, with value density not greater than $k^{\frac{high(t)}{\psi}}$. We conclude that the cumulative value density of the clairvoyant schedule at time t is bounded by $2nk^{\frac{high(t)}{\psi}}$.

[10] I.e., released at time r with deadline d and computation time c.

second band, but at time 4 it is abandoned in favor of T_7 which arrived to its LST and has a later deadline.

At time 5, the SP of the first band becomes idle, which creates a switch of the roles between the SP and RP of that band. Later at time 6, T_{10} is released; it can not be scheduled on its own band's SP but after cascading it is scheduled on the (new) SP of the first band.

All in all, the *Cascading Multi-Band Algorithm* completed all the tasks but T_5 and T_9. A clairvoyant scheduler could schedule all the tasks (T_5 can be scheduled on the idle SP and T_9 can be scheduled before its LST on the same processor). Figure 1 shows the schedule created by the *Cascading Multi-Band Algorithm.*

6 The Algorithm's Competitive Multiplier

In this section we would like to study the behavior of the *Cascading Multi-Band Algorithm* and to compute its competitive multiplier. The final result is stated in theorem 6.2.

Recall that Γ is the entire set of tasks to be scheduled. We partition the tasks of Γ according to the behavior of the *Cascading Multi-Band Algorithm*: tasks that never completed (F) and tasks that completed successfully (S). Denote by $V(\Gamma)$ and $C(\Gamma)$ the value achieved by the *Cascading Multi-Band Algorithm* and the clairvoyant scheduler from scheduling Γ, respectively. We will show that $C(F) \leq \alpha C(\Gamma)$ (for some constant *alpha*), using the Lost Value Lemma (4.1) we will get a competitive multiplier of $\alpha + 1$.

Before we start we must introduce some notation and definitions.

Definition 6.1

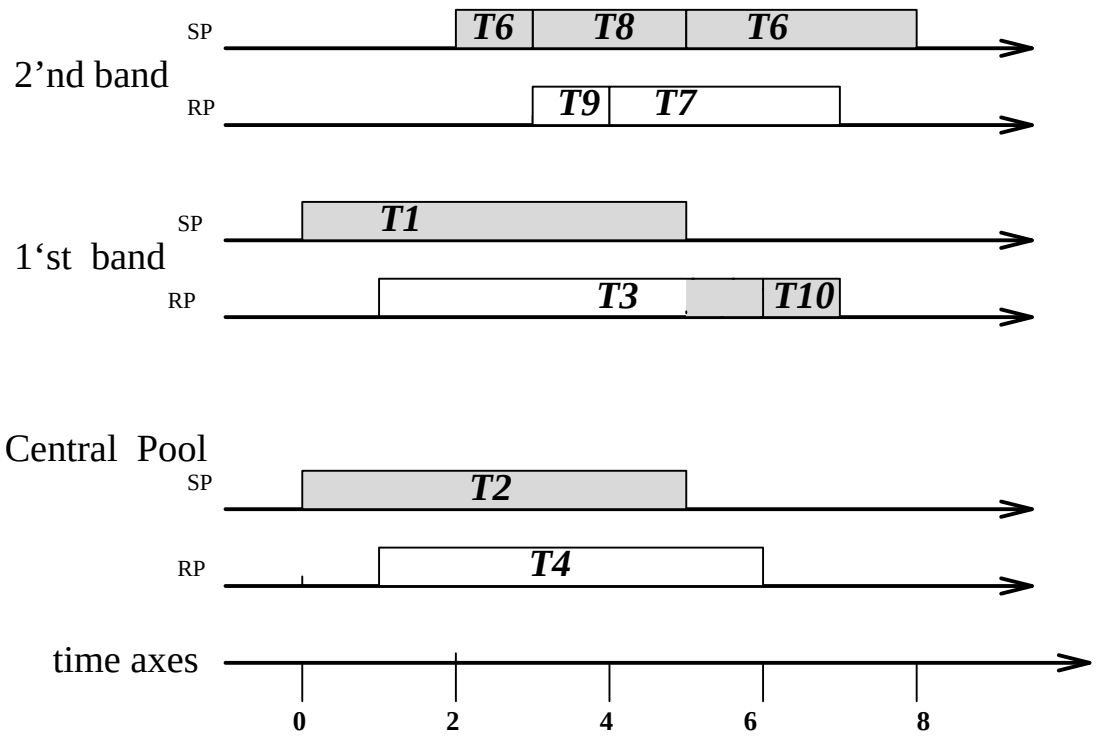
- PRODUCTIVE BAND: A band is said to be *productive* at time t if at that time, its SP is not idle.
- EXECUTABLE PERIOD: The *executable period*, of a task is the interval between its release time and its deadline.

  By definition, a task may be scheduled only during its executable period.
- CUMULATIVE VALUE DENSITY (CVD): Suppose some schedule is chosen, the *cumulative value density* at time t is the sum of the value densities of all tasks executing at time t. [9]

□

Lemma 6.1 *If at time t a task with deadline d is executing on RP of a band (i.e this task was scheduled by an LST interrupt) then that band will be productive between t and d.*

[9] For example, for a system with $2n$ processors, if all processors are idle at time t then $CVD(t) = 0$. If half of the processors (i.e, n) execute tasks with unit value density and the others execute tasks of value density k then the cumulative value density is $n + kn$. In no case can $CVD(t)$ be bigger than $2nk$.



legend

SP, RP	initial assignment as safe or risky in each band
	a task on SP
	a task on RP

Figure 1: The *Cascading Multi-Band Algorithm* Scheduling

(i.e., with no central pool) can be crippled when the task set consists of mostly low value density tasks since all the higher bands will be left idle. In order to minimize the loss of such cases we add the central pool to the bands structure. If all the tasks are of low value density then all high bands would still be left idle but the bands in the central pool would be utilized.

A big enough central pool will offset the damage caused by higher idle bands. However, making the central pool too big can cause another problem—weakening the advantages of the higher value density tasks. We conclude that choosing the right size of the central pool is a delicate and important aspect of the the *Cascading Multi-Band Algorithm*.

The following is a small example of the *Cascading Multi-Band Algorithm*'s scheduling.

Example 5.1

Assume that the highest possible value density is 16 ($k = 16$), number of processors is 6 from which 2 are allocated as a central pool and the rest constitute 2 bands. The first band will be for tasks with value density below 4 and the second for tasks with value density of 4 and above. For this example, consider the tasks depicted in the following table:

Task	*Release Time*	*Computation Time*	*Slack Time*	*Deadline*	*Value Density*
T_1	0	5	0	5	1
T_2	0	5	0	5	2
T_3	1	5	0	6	3
T_4	1	5	0	6	3
T_5	1	1	0	2	3
T_6	2	4	2	8	16
T_7	2	3	2	7	10
T_8	3	2	1	6	10
T_9	3	2	0	5	10
T_{10}	6	1	1	8	16

The first two tasks to be released are scheduled on the SP of the first band and the central pool (T_2 cascades into the central pool). When T_3 is released it can not be scheduled on an SP, so it is inserted into *Q_waiting* only to create an LST interrupt immediately. Then, it is scheduled on the RP of the first band. In the same way T_4 is scheduled on the RP of the central pool. But when T_5 arrives it can be scheduled neither on any of the SPs nor on any of the RPs, hence is abandoned (in the LST routine). Note that T_5 is abandoned even though the second band is idle (a task can cascade only downwards).

All the remaining tasks have value density high enough to be scheduled on the second band. T_6 is scheduled on the SP. T_7 can not be scheduled on any of the SPs and it enters *Q_waiting* (with LST at 4). T_8 can be added to the SP of the second band preempting T_6 (which has a latter deadline). T_9 can not be scheduled on any of the SPs; it reaches its LST and is scheduled on the RP of the

2. At LST (of a non-privileged task), decide whether and where a task should be scheduled or maybe abandoned.

3. The method used in scheduling each band (and the central pool).

Different choices for these three components would create different variants of the algorithm. In this paper we describe one specific variant that we call the *Cascading Multi-Band Algorithm*. In this variant, the central pool is also broken into bands of two processors each[6]. All the pairs (i.e, bands) execute the *same* two processor scheduling algorithm.

At each moment, every band has one of its processors designated as the *Safe Processor* (SP) and the other as the *Risky Processor* (RP). Each band has its own queue called *Q_privileged*, the tasks in *Q_privileged* are guaranteed to complete. In addition to the local *Q_privileged* queues there is one global queue called *Q_waiting*. This queue includes all the ready tasks that are not privileged.

When a new task T is released, it is assigned to a band as follows:

1. It is added to the *Q_privileged* of its own band if this does not create overload (i.e, all tasks including the new task can complete on SP). Otherwise, T cascades downward as described above.

2. If T was not accepted by any band (including all the bands in the central pool) it enters the *Q_waiting* where it waits until its LST occurs.

So, at release time only the SPs are examined. A task might not be scheduled even if an RP is idle. A task T that reached its LST is assigned to a processor as follows:

1. If there is any idle RP among all the lower level bands (including T's own level) then schedule T on one of these processors[7].

2. If there is no idle RP among lower level bands, we might abandon a task executing on one of these RPs in order to schedule T, depending on the following rule:

   Let T^* be the task with earliest deadline among all the tasks executing on these RPs.
   Abandon the task with earlier deadline among T and T^*.

If, at task completion event, SP of a band becomes idle while RP is not then the two processors should switch roles; the *safe-processor* becomes the *risky-processor* and vice versa. This does not require task migration.

Using idle RPs and scheduling tasks of *Q_waiting* before they reach their LST are heuristics for improving the average case behavior of the scheduler.

The bands structure as described above prioritize high value density tasks over low value density tasks. Higher value density tasks start their cascading at a higher point and cascading is possible in only one direction - downwards[8]. However, an algorithm that uses the "pure" bands structure

[6]The bands of the central pool are ordered so that a task that reaches the pool start with the first band in the pool and if not accepted it cascades to the second band and onwards. If the task is not accepted by the last band in the pool ("end-of-the-world") it enters *Q_waiting*.

[7]Heuristics can be used to choose the processor in case that there are more than one idle RP. Examples might be the one of the lowest band, or maybe the highest.

[8]Hence, higher value density tasks have more bands that can possibly accommodate them.

$$C(\Gamma) \leq (c+1)V(\Gamma)$$

PROOF.

$$\begin{aligned} C(\Gamma) &= C(F \cup S) \leq C(F) + C(S) \\ &= C(F) + V(S) = C(F) + V(\Gamma) \\ &\leq (c+1)V(\Gamma) \end{aligned}$$

□

5 The Cascading Multi-Band Algorithm

Suppose a system has $2n$ processors. We break the processors into two disjoint groups: 2ψ processors will constitute a "band structure" and the other 2ω processors will constitute a "central pool" as described below ($n = \psi + \omega$; and $n > \omega \geq 0$).

We consider ψ intervals (*levels*) of value density $[1..k^{\frac{1}{\psi}}), [k^{\frac{1}{\psi}}..k^{\frac{2}{\psi}}), \cdots, [k^{\frac{\psi-1}{\psi}}..k]$, call these levels $1, \cdots, \psi$ respectively. The i'th band is said to be "lower" than the $i+1$'st band.

Suppose the entire set of tasks to be scheduled is Γ. We partition this set according to the value density of the tasks: $\Gamma = \Gamma_1 \cup \Gamma_2 \cdots \cup \Gamma_\psi$ where Γ_i contains all tasks with value density in the range [4] $[k^{\frac{i-1}{\psi}}, k^{\frac{i}{\psi}})$.

We allocate 2 processors (*a band*) for each of the ψ value density levels. In addition, the remaining 2ω processors are allocated as a *central pool*, that will be used by tasks of all levels.

The main idea of the algorithm is to assign a task, upon its release, to the band corresponding to its value density. However, if the band is already overloaded and can not accommodate the new task this task will "try" the band one below. If the band below can not accept the new task, the task will continue to *cascade* downwards. If a task cascades to the lowest band but still can not be scheduled there it can go into the central pool.

If a newly released task is accepted by one of the bands or by the central pool it is guaranteed to complete before its deadline (these tasks are called "privileged"). If it is not, it awaits its LST[5] (*Latest Start Time*). at which time it tries again to be scheduled (details to follow). The algorithm has three major components:

1. Upon task release, assign a task to a band (possibly after cascading).

[4] All but the last interval is half open half closed. The last level corresponds to the closed interval $[k^{\frac{\psi-1}{\psi}}, k]$.

[5] Definition: LST = (deadline - remaining computation time). If a task is not scheduled at its LST, it will not complete.

$$\geq \frac{\frac{k}{(k-1)} n(k^{\frac{1}{n}} - 1) V(t_{l_0})}{V(t_{l_0}) + \epsilon V(t_{l_0})}$$
$$\geq \frac{\frac{k}{(k-1)} n(k^{\frac{1}{n}} - 1)}{1 + \epsilon}$$

This holds for every positive ϵ hence the Theorem is proved.

□

Corollary 3.3 As the number of processors n tends to infinity, no on-line algorithm can have a competitive multiplier smaller than $\ln k$ (natural logarithm).

4 Algorithmic Guarantees

We consider two possible models of multiprocessor systems. In the first model, tasks can *migrate* cheaply (and quickly) from one processor to another. Hence, if a task started to execute on one processor it can later continue on any other processor (and migration takes no time). In the second model (the *fixed* model), once a task starts to execute on one processor it can not execute on any other processor. An on-line scheduler can do better when migration is possible.

4.1 The Lost Value Lemma

Let $\mathcal{A}$ be an on-line scheduler and Γ a set of tasks to be scheduled. We can partition the tasks of Γ according to the behavior of $\mathcal{A}$

1. Tasks that never completed (F), the "lost" ones.
2. Tasks that completed successfully (S)

$$\Gamma = F \cup S$$

Denote by $V(\Gamma)$ and $C(\Gamma)$ the value achieved by $\mathcal{A}$ and the clairvoyant scheduler from the tasks of Γ.

Lemma 4.1 *The Lost Value Lemma*

If for some multiplier c and every set of tasks Γ,

$$C(F) \leq cV(\Gamma)$$

Then,

The above term monotonically decreases when i_0 increases, hence the minimum is obtained when $i_0 = n$ and its value is

$$\frac{nk}{1 + k^{\frac{1}{n}} + k^{\frac{2}{n}} + \cdots + k^{\frac{n-1}{n}}}$$

The sum of the geometric sequence in the denominator is $\frac{k-1}{k^{\frac{1}{n}}-1}$ hence the smallest possible ratio is,

$$\frac{k}{k-1} n(k^{\frac{1}{n}} - 1)$$

and the lemma is proved. □

The preceding lemma dealt with the special case that only one value density level is missing from the on-line schedule. But what will happen if more than one level is missing? In appendix B we show that this can not benefit the on-line scheduler (for a "good" choice of i_0). Hence, we get the following theorem.

Theorem 3.2 *For a system with n processors and maximal value density of k, there is no on-line scheduling algorithm with competitive multiplier* smaller than $\frac{k}{(k-1)} n(k^{\frac{1}{n}} - 1)$.

PROOF.

Fix an on-line scheduling algorithm. Denote by $V(t_l)$ the value obtained by the on-line scheduler until time t_l. Lemma B.3 shows that the value obtained by the clairvoyant algorithm is at least $\frac{k}{(k-1)} n(k^{\frac{1}{n}} - 1)V(t_l)$.

Note that t_l tends to infinity as l goes to infinity. If $V(t_l)$ does not tend to infinity as l goes to infinity then the competitive multiplier of the on-line algorithm is not bounded (because the clairvoyant algorithm gets a value of at least $nt_l \to \infty$). Otherwise, $V(t_l)$ tends to infinity, hence for arbitrarily small $\epsilon > 0$ there is a big enough l_0 such that

$$V(t_{l_0}) \geq \frac{1}{\epsilon} kn \Rightarrow kn \leq \epsilon V(t_{l_0})$$

Suppose the game ends at t_{l_0} (i.e., no more task releases). The total value obtained by the on-line scheduler is not greater than $V(t_{l_0}) + kn$ (because all the tasks not yet completed has length at most 1 and value density at most k). The clairvoyant scheduler gets a value of at least $\frac{k}{(k-1)} n(k^{\frac{1}{n}} - 1)V(t_{l_0})$.

Hence,

$$\begin{aligned} & \frac{\textit{value obtained by the clairvoyant scheduler}}{\textit{value obtained by the on-line scheduler}} \\ \geq \quad & \frac{\frac{k}{(k-1)} n(k^{\frac{1}{n}} - 1)V(t_{l_0})}{V(t_{l_0}) + kn} \end{aligned}$$

at least one level is not chosen by the on-line scheduler. Let i_0 be an index of *some* level (to be specified later) that is not chosen. Then, t_{l+1} is set to be the end of the current i_0'th level period (this means that up to that time there will be no new task releases). We will say that t_l is associated with level i_0.

Suppose that t_l is associated with level i_0, then what can the clairvoyant scheduler do? It can execute n tasks of level i_0 to completion between t_l to t_{l+1}. In this scheme, the clairvoyant scheduler schedules all the processors in the same way, no processor is ever idle and all current tasks complete immediately before a new set is released.

At least one but possibly several value densities could be missing from the on-line schedule. We will start by proving results assuming that only one level is missing. Later, we will produce a general scheme to choose i_0 among all absent density levels.

Lemma 3.1 *Suppose the on-line scheduler gets partial value for tasks that were executed but not to completion. In particular, for a task that executed for α $(0 \leq \alpha \leq 1)$ of its computation time the scheduler will get α times the task's value.*

If only one density level is missing from the schedule at time t_l, then the effective value density (i.e, the value achieved divided by the period's length) obtained by the clairvoyant scheduler between t_l and t_{l+1} is at least $\frac{k}{(k-1)}n(k^{\frac{1}{n}}-1)$ times bigger than the effective value density obtained by the on-line scheduler for the same period.

PROOF.
Let i_0 be the level associated with the period in question. The clairvoyant algorithm schedules to completion n tasks of level i_0 and gets an effective value density of $k^{\frac{i_0}{n}}$ on all processors.

We know that during this period the on-line scheduler did not execute any task of level i_0. Instead, it scheduled tasks of lower and higher levels. The effective value density of any tasks of higher level is much smaller than its value density because of their short period. In fact all these tasks have effective value density of at most ϵ. Hence, the effective value density achieved by the on-line scheduler is at most

$$\begin{aligned} & 1 + k^{\frac{1}{n}} + k^{\frac{2}{n}} + \cdots + k^{\frac{i_0-1}{n}} + \underbrace{\epsilon + \epsilon + \epsilon}_{n-i_0 \ times} \\ = \quad & 1 + k^{\frac{1}{n}} + k^{\frac{2}{n}} + \cdots + k^{\frac{i_0-1}{n}} + (n-i_0)\epsilon \end{aligned}$$

We are looking for the smallest possible ratio between the effective value densities of the clairvoyant and the on-line scheduler. That is,

$$\min_{0 \leq i_0 \leq n} \frac{nk^{\frac{i_0}{n}}}{(1 + k^{\frac{1}{n}} + k^{\frac{2}{n}} + \cdots + k^{\frac{i_0-1}{n}} + (n-i_0)\epsilon)}$$

2. We present an algorithm called the *Cascading Multi-Band Algorithm*. For a system with $2n$ processors and importance ratio of $k > 1$, this algorithm has an algorithmic guarantee of at most

$$1 + 2n \min_{(0 \le \omega < n; n = \omega + \psi)} \left\{ \max_{1 \le i \le \psi} \frac{k^{\frac{i}{\psi}}}{\omega + \frac{(k^{\frac{i}{\psi}} - 1)}{(k^{\frac{1}{\psi}} - 1)}} \right\}$$

When n tends to infinity this bound is at most $2 \ln k + 3$, which is within a multiplicative factor of 2 from the complexity bound for the same system.

3. We present an algorithm called the *Safe-Risky* algorithm, for two-processor systems with uniform value density that achieves the best possible competitive multiplier of 2 even when tasks may **have slack time** but migration is allowed [3]. For the "No-Migration" model, a variant of this algorithm, called the *Safe-Risky-(fixed)*, achieves a competitive multiplier of 3.

3 The Complexity Bounds

We would first like to show that no on-line algorithm can have a competitive multiplier smaller than $\frac{k}{(k-1)} n (k^{\frac{1}{n}} - 1)$. As usual in proofs of this kind, we assume that an adversary can choose a set of tasks. The set depends upon the on-line algorithm's behavior.

We consider $n+1$ possible levels of value density $1, k^{\frac{1}{n}}, k^{\frac{2}{n}}, \cdots, k^{\frac{n}{n}} = k$, call them levels $0, 1, \cdots, n$.

With each level we associate a period. A task of some level of value density will have a computation time and deadline equal to the corresponding period. Hence, the value of a task of level i equals the length of the i'th period times the i'th value density. The length of the 0'th level's period is set to 1. We choose all other periods in such a way that the value of an $i+1$'th level task is only a small fraction of the i'th level task's value. In fact, we choose it so the $i+1$'th task's "effective value density" taken over the i'th period is arbitrarily small (say ϵ for some small positive ϵ).

A collection of tasks that has n identical tasks for each level is called a *complete set*. In this game, the adversary release tasks only in complete sets. The behavior of the on-line scheduler dictates *when* the next complete set is to be released.

Let t_l be the time when the l'th set was released (hence, always $t_1 = 0$). At time t_l (in particular at time 0), the on-line algorithm has to schedule a complete set. The number of possible scheduling decisions is vast. However, since the number of processors is smaller than the number of levels,

[3] This was already known when tasks have no slack-time [3,12].

Zhou et. al. presented an on-line algorithm [13][2] for distributed real-time systems. Their model resembles ours but our goal is to give worst case guarantees for value obtained (even for overloaded systems) while their goal is to generate a schedule efficiently when the system is underloaded (i.e, all tasks can be scheduled).

Wang and Mao [3,12] showed a lower bound of 2 (on the competitive multiplier) and presented an algorithm that achieved this bound for an arbitrary even number of processors, assuming uniform value density and **no** slack time.

We present here algorithms and complexity results for tasks having slack time, executing on an arbitrary number of processors, and with arbitrary importance ratios. The gap between the performance guarantees of our algorithms and the complexity results shows that there is still work to be done. Our adversary arguments and algorithms offer, however, three useful insights:

1. A parallel on-line scheduling algorithm improves its competitive multiplier by devoting some of its processing resources to every possible value density.

2. The lower bound on the best possible competitive multiplier (as measured by our adversary arguments) converges to $\ln k$ as the number of processors approaches infinity. Our current algorithm gives a guarantee that converges to $2 \ln k + 3$ as the number of processors approaches infinity.

3. It should be easier to design a competitive on-line scheduler when tasks are assumed to have no slack time. However, in all the multi-processors scheduling algorithms that we studied, the introduction of slack time entails only a fixed deterioration of the competitive multiplier. The same fixed deterioration is observed when comparing systems with cheap migration vs. systems where migration is impossible.

This paper is organized as follows: Section 2 summarize the main results. In section 3, we present our complexity results using a novel adversary argument. From section 4 onwards, we describe our algorithmic results. Section 4.1 states the *Lost Value Lemma* that we use frequently to relate results in the case where migration is both permitted and cheap to the case where it is not possible. In sections 5 and 6 we describe and analyze the *Cascading Multi-Band Algorithm*. Our results for two-processors systems are presented in appendix A. The paper ends with a brief conclusion and a discussion of some salient open problems.

2 Summary of Results

We present algorithms and complexity results for multiprocessor scheduling of overloaded real-time systems.

1. For a system with n processors and maximal value density of $k > 1$, there is no on-line scheduling algorithm with competitive multiplier smaller than $\frac{k}{(k-1)} n(k^{\frac{1}{n}} - 1)$.

   When n tends to infinity this lower bound tends to $\frac{k}{(k-1)} \ln k$.

[2] And additional references within.

failures of some computational resources or as a transient condition (e.g., in telecommunications and multimedia). In a parallel setting, it is possible to tolerate failures and still meet the deadlines of many high-value tasks. To do this, we must have algorithms that can adapt to the number of processors available and give performance guarantees.

Recently, several groups of researchers (including us) have presented complexity results and algorithms that give guarantees for overloaded real-time systems. The current paper extends our previous *uniprocessor* results to parallel architectures. We discuss both uniform shared memory models (where thread migration is cheap and scheduling is global) and non-uniform shared memory models (where thread migration is impractical but scheduling is still global). By "migration" we mean the ability to move a thread that has already begun execution from one processor to another. For both models, we assume that preemption within a processor takes no time (a reasonable assumption for real-time kernels).

In our work-flow model, the scheduler is given no information about a task before its release time. When a task is released, its value, computation time and deadline are known precisely. If a task completes before its deadline, then the system acquires its value. Otherwise, the system acquires no value for that task. This is known as a *firm* on-line real-time model. The goal of the scheduler is to obtain as much value as possible.

The *value density* of a task is its value divided by its computation time. The *importance ratio* of a collection of tasks is the ratio of the largest value density to the smallest value density. For convenience, we normalize the smallest value density to be 1. When the importance ratio is 1, the collection is said to have *uniform value density*, i.e., a task's value equals its computation time. We will denote the importance ratio of a collection by k.

We choose to quantify the performance guarantee of an on-line scheduler by comparing it with a *clairvoyant* (also called off-line) scheduling algorithm. A clairvoyant scheduler has complete *a priori* knowledge of all the parameters of all the tasks. A clairvoyant scheduler can choose a "scheduling sequence" that will obtain the maximum possible value achievable by any scheduler [1].

As in [2,6,11] we say that an on-line algorithm has a *competitive factor* r, $0 \leq r \leq 1$, *if and only if* it is guaranteed to achieve a cumulative value of at least r times the cumulative value achievable by a clairvoyant algorithm on *any* set of tasks. For convenience of notation, we use *competitive multiplier* as the figure of merit. The *competitive multiplier* is defined to be "one over the competitive factor". The smaller the competitive multiplier is the better the guarantee is. Our goal is to devise on-line algorithms with worst case performance guarantee.

For uniprocessor environments with an importance ratio k, Baruah et. al. [2,3] showed a lower bound of $(1+\sqrt{k})^2$. Wang and Mao [12] first reported an algorithm that achieves this bound when k is 1. Having independently developed an algorithm for the $k = 1$ case, we were able to generalize this to an algorithm called D^{over} that meets the Baruah et al. bound for all k [7].

For multi processor environments, Mok and Dertouzos [4] showed that no optimal algorithm exists even when the system is underloaded. Locke ([9] pp. 124-134) presented heuristics for this case.

[1] Finding the maximum achievable value for such a scheduler, even in the uniprocessor case, is reducible from the knapsack problem [5]; hence is NP-hard.

Competitive Algorithms and Lower Bounds for On-Line Scheduling of Multiprocessor Real-Time Systems *

Gilad Koren Dennis Shasha

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

June 23, 1993

Abstract

We study competitive on-line scheduling in multi-processor real-time environments. In our model, every task has a deadline and a value that it obtains only if it completes by its deadline. A task can be assigned to any processor, all of which are equally powerful. The problem is to design an on-line scheduling algorithm (i.e.,the scheduler has no knowledge of a task until it is released) with worst case guarantees as to the total value obtained by the system.

We study systems with two or more processors and with uniform or non-uniform value density. We present an inherent limit on the best competitive guarantee that any on-line parallel real-time scheduler can give. Then we present a competitive algorithm that achieves a worst case guarantee which is only within a factor of 2 from the best possible guarantee in many cases. These are the most general results yet known for parallel overloaded real-time scheduling.

1 Introduction

In modern life, real-time computer systems are gaining importance at a rapid pace. Once limited to exotic applications, real-time applications now can be found in many civilian and military products. These range from multi-million dollar gadgets like (the proposed) space stations to relatively mundane products like cars and airplanes. Real-time systems control the production and safety in power plants, factories, labs and perhaps soon in our home alarm systems.

A firm overloaded real-time system is one for which even a clairvoyant scheduler cannot meet all deadlines with the available computational resources. Overload can arise either as the result of

*Supported by U.S. Office of Naval Research #N00014-91-J-1472 and #N00014-92-J-1719, U.S. National Science Foundation grant #CCR-9103953.
Authors' e-mail addresses : koren@cs.nyu.edu and shasha@cs.nyu.edu