MOCA : A Multiprocessor On-Line Competitive Algorithm for Real-Time System Scheduling*

Gilad Koren† Dennis Shasha

Department of Computer Science
Courant Institute, New York University
251 Mercer Street
New York, NY 10012-1185

November 14, 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., one in which 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. 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 within a small constant factor from the best possible guarantee in many cases. We consider two memory models: a distributed system having a centralized scheduler and a shared memory multiprocessor.

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.

---

*Supported by U.S. Office of Naval Research grants #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`
†Current address: Department of Computer Science, Bar-Ilan University, Ramat-Gan 52900, Israel (`koren@bimacs.cs.biu.ac.il`).

These range from multi-billion dollar gadgets like (the proposed) space station to relatively low-cost products like cars and airplanes. Real-time systems control production and safety in power plants, factories, labs and perhaps soon in our homes.

An 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 failures of some computational resources or as a transient condition (e.g., an overloaded communications circuit).

Recently, several groups of researchers have presented inherent bounds and algorithms that give guarantees for overloaded real-time systems. The current paper extends our previous *uniprocessor* results to parallel architectures. We consider two memory models: a shared memory model where thread migration is cheap and a distributed memory model where thread migration is impractical. In both cases, we assume a centralized scheduler.

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[1] (as in [6, 14]).

Multiprocessor real-time scheduling is an active field of research. Both shared memory [14, 17] and distributed memory [16, 20, 22, 18] architectures have been studied. *Static binding* of tasks to processors (i.e., no migration) is assumed in [16, 17, 18] while dynamic binding is assumed in [6, 14]. For a survey of scheduling issues for uniprocessor and multiprocessor systems see [1, 4].

Mok and Dertouzos [6] showed that in a multiprocessor environment an optimal algorithm must have an *a priori* knowledge of release times. Hence, no on-line optimal algorithm exists even when the system is underloaded. Locke [15, pp. 124–134] presented heuristics for the multiprocessor environment. Ramamritham and Stankovic [20] studied the question of scheduling firm deadline tasks in a distributed environment. They proposed a scheduler that assumes, at the design phase, that the system is underloaded for *critical* tasks. The non-critical tasks are scheduled dynamically and heuristically using any surplus processing power. Our algorithm could be used for the non-critical tasks in their environment.

Zhou et. al. presented an on-line algorithm [22][2] for distributed real-time systems. Our model resembles theirs 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).

In our system model, tasks are sporadic (aperiodic), independent (i.e., no inter-task precedence constraints) and can be preempted at any time. A preempted task can later resume its work. 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. Following [8, 13], we denote such deadlines as *firm.* (Other papers [4] denote such deadlines as *hard.*

[1] A reasonable assumption since real time kernels are designed to keep all tasks' code and data in memory thereby avoiding paging-induced faults during context switches; also, such kernels are built with short code path lengths.

[2] And additional references within.

The reader should therefore be aware of the definitional variations). The goal of the scheduler is to obtain as much value as possible.

As in [2, 3], 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 of tasks by k and will assume throughout the paper that this value is known before a given execution begins.

We quantify the performance guarantee of an on-line scheduler by comparing it with a *clairvoyant* [16, p. 39] scheduling algorithm (also called an off-line scheduler). 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 [3] .

As in [3, 9, 13, 19] 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 the best possible worst case performance guarantees; that is, the lowest competitive multiplier possible.

For uniprocessor environments with an importance ratio k, Baruah et. al. [2, 3] showed a lower bound of $(1+\sqrt{k})^2$ on the best competitive multiplier that any on-line scheduler can have. Wang and Mao [2, 21] first reported an algorithm that achieves this bound when k is 1. Having independently developed an algorithm for the $k = 1$ case [12], we were forced to use a different algorithmic strategy to achieve an algorithm called D^{over} [11, 13] that meets the Baruah et al. bound for all k.

For multiprocessor environments, Wang and Mao [2, 21] 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 that tasks are released with *no* slack time[4]. We present algorithms and lower bounds for tasks *having* slack time, executing on an arbitrary number of (identical) processors, and with arbitrary importance ratios.

This paper is organized as follows: Section 2 summarizes the main results. In section 3, we present our lower bound using a novel adversary argument. Section 4 describes the *MOCA Algorithm*, a new scheduling algorithm for multiprocessor real time systems. In section 5, we analyze the *MOCA Algorithm*. The paper ends with a brief conclusion and a discussion of some salient open problems.

[3] Finding the maximum achievable value for such a scheduler, even in the uniprocessor case, is reducible from the knapsack problem [7]; hence is NP-hard.

[4] We say that a task has *no slack time* if the computation time needed to execute the task to completion equals its deadline minus the current time (i.e., such a task must be scheduled immediately in order to complete).

2 Summary of Results

We present algorithms and lower bound results for multiprocessor scheduling of overloaded real-time systems. We consider two memory models: a shared memory model where thread *migration* is cheap and a distributed memory model where thread migration is impractical. In both cases, we assume a centralized scheduler. In the first model, tasks can *migrate* cheaply (and quickly) from one processor to another. Hence, if a task starts 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 cannot execute on any other processor. For both models, we assume that preemption within a processor takes no time.

- Inherent Bound on The Best Possible Competitive Multiplier

  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 $\frac{k}{(k-1)} \ln k$.

  This result holds even when migration is allowed.

- The *MOCA Algorithm*

  We present an algorithm that does not use migration called *MOCA: Multiprocessor On-line Competitive Algorithm.* For a system with $2n$ processors and importance ratio of $k > 1$, this algorithm has a competitive 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 small multiplicative constant factor from the lower bound for the same system.

- Scheduling Algorithms for Two-Processor Systems

  In [10], we present an algorithm called the *Safe-Risky* algorithm, for two-processor systems with uniform value density (i.e., $n = 2$ and $k = 1$) that achieves the best possible competitive multiplier of 2 even when tasks may **have slack time** but migration is allowed[5]. For the "No-Migration" model, a variant of this algorithm, called the *Safe-Risky-(fixed)*, achieves a competitive multiplier of 3.

[5] This was already known when tasks have no slack-time [2, 21].

3 The Lower Bound

We would first like to show that every on-line algorithm has a competitive multiplier of at least $\frac{k}{(k-1)}n(k^{\frac{1}{n}}-1)$ for a system with n processors and importance ratio of k. As usual in proofs of this kind, we assume that a game is played between an adversary and the on-line scheduler.

We consider $n+1$ possible levels of value density $1, k^{\frac{1}{n}}, k^{\frac{2}{n}}, \ldots, k^{\frac{n}{n}} = k$, call them levels $0, 1, \ldots, n$. With each level we associate a period. A task of some value density level 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 $i+1$'st level task is only a small fraction of the i'th level task's value. In fact, we choose it so the $i+1$'st task's *effective value density*[6] taken over the i'th period is arbitrarily small (say ϵ for some small ϵ). A collection of tasks that has n identical tasks for each level, where all are released at the same time is called a *complete set*[7]. Figure 1 shows a complete set for a system with 3 processors and value density of 8.

The adversary controls the release of tasks, making decisions after observing the actions (schedule) of the on-line algorithm so far. In the following we describe the game played by the adversary and the on-line scheduler.

The game is played by stages, the first one beginning at time 0. At the beginning of each stage the adversary releases a complete set of tasks. The adversary releases tasks only in complete sets and only in the beginning of a stage. The behavior of the on-line scheduler dictates *when* the adversary releases the next complete set (i.e, the beginning of the next stage). Denote by t_l the beginning of the l'th stage. At time t_l (in particular at time 0), the on-line algorithm has to schedule a new complete set and possibly some previously released tasks. The number of possible scheduling decisions is vast. However, since the number of processors is smaller than the number of levels, at least one level is not represented in the on-line schedule (at time t_l). Let i_0 be an index of *some* level (to be specified later) that is not represented. 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 the adversary will release no new tasks. We will say that the stage starting at t_l is associated with level i_0. The game goes on in that manner for a big enough number of stages (see proof of theorem 3.3).

Suppose that the stage starting at t_l is associated with level i_0, then what can the clairvoyant scheduler do? One possibility is to 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.

The idea behind the lower bound game is that while the clairvoyant scheduler gets a value density of $k^{\frac{i_0}{n}}$ for the duration of the entire stage on *all* the processors. The on-line scheduler utilizes its processors either on lower value density tasks or on higher value density tasks that have very short duration (hence have little value). After the completion of these short high value density tasks,

[6] See definition 3.1.

[7] Hence, a complete set has $n(n+1)$ tasks.

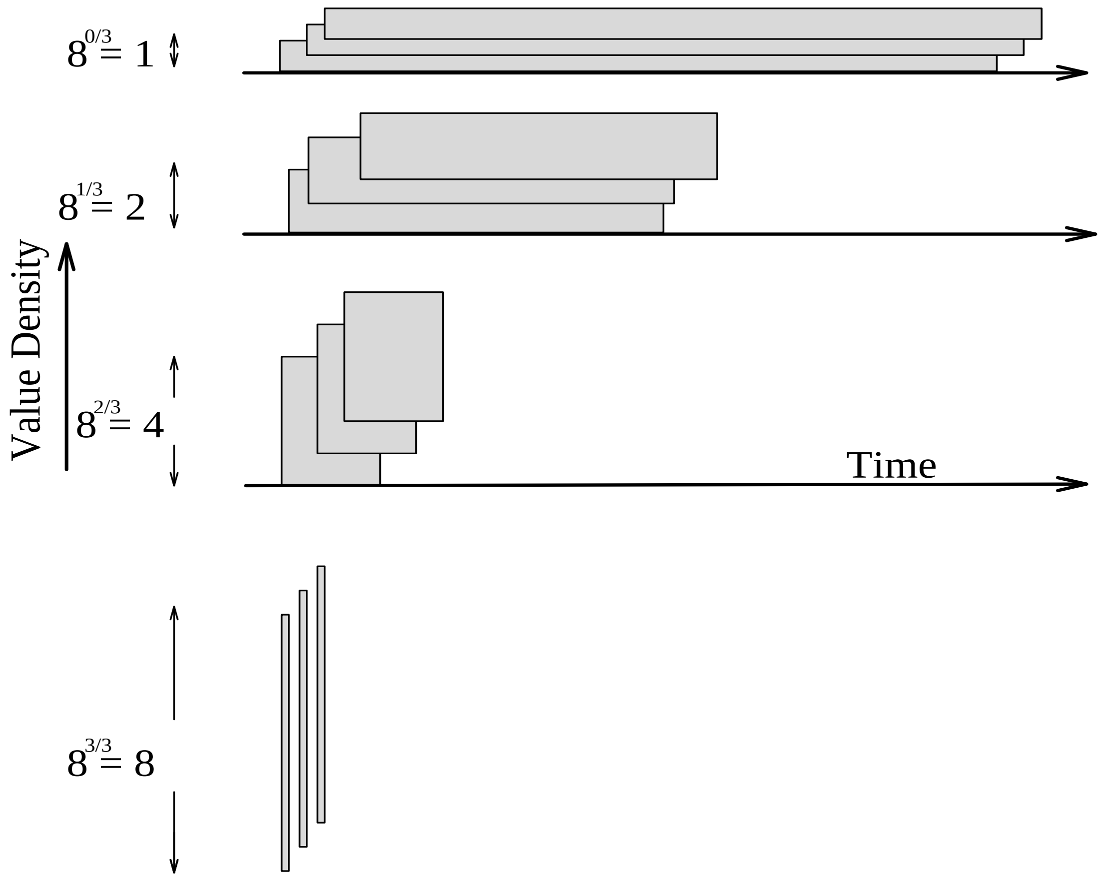

Figure 1: A complete set for $n = 3$ and $k = 8$.
Note that there is one more level than processors and the levels correspond to value densities k^0, $k^{1/3}$, $k^{2/3}$, and $k^{3/3}$. Each level has as many tasks as there are processors. Finally, each lower value density task T is so much longer than any higher value density task T' that the value of T is much greater than the value of T'.

the associated processors will be left idle because no more tasks are released before the end of the stage.

The question is how to choose the level associated with a given stage. When only one value density is missing from the on-line schedule then this level is the one. We will start by proving results assuming that only one level is missing. Later, we will show that these results hold in the general case, when more than one level of value density is missing.

Definition 3.1 EFFECTIVE VALUE DENSITY The effective value density obtained by a scheduling algorithm $\mathcal{A}$ during the period between time t_1 and t_2 is the sum of value densities scheduled during this period, weighted according to their length of execution during the period. Formally, for any task T, let $duration(T)$ denote the duration for which T was scheduled (by $\mathcal{A}$) between time t_1 and t_2. Then the effective value density of the algorithm $\mathcal{A}$ between t_1 and t_2 is:

$$\sum \frac{value\_density(T) \times duration(T)}{t_2 - t_1}$$

The sum is taken over all tasks T that have been scheduled for execution between t_1 and t_2.

We will say that the the effective value density of a task T between t_1 and t_2 is its contribution to the above sum. I.e.,

$$\frac{value\_density(T) \times duration(T)}{t_2 - t_1}$$

Lemma 3.1 *If only one density level, i_0, is missing from the on-line schedule at the beginning of some stage, $\mathcal{S}$, then the effective value density obtained by the clairvoyant scheduler during stage $\mathcal{S}$ is at least $\frac{k}{(k-1)} n(k^{\frac{1}{n}} - 1)$ times greater than the effective value density obtained by the on-line scheduler for the same period.*

PROOF.

Suppose the clairvoyant algorithm schedules to completion n tasks of level i_0. This corresponds to an effective value density of $nk^{\frac{i_0}{n}}$.

During stage $\mathcal{S}$ the on-line scheduler did not execute any task of level i_0 because no task of that level was scheduled at the beginning of the stage and the adversary releases no new tasks before the end of the stage. Instead, it scheduled tasks of lower or higher levels. The effective value density of any task of higher level is much smaller than its value density because of its short period. In fact all such tasks have effective value density of at most ϵ during $\mathcal{S}$. Hence, the effective value density achieved by the on-line scheduler is at most

$$1 + k^{\frac{1}{n}} + k^{\frac{2}{n}} + \cdots + k^{\frac{i_0-1}{n}} + \underbrace{\epsilon + \cdots + \epsilon}_{n-i_0\ times}$$

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

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

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

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

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 the following we show that this cannot benefit the on-line scheduler (for a "good" choice of i_0). Hence, the lower bound holds in the general case.

Actually, we look for a value density level (at time t_l) that has the following *single representative* property: *No task of that level is currently executing and all lower levels have only one representative in the on-line schedule*[8] A level having the single representative property will give us the desired result.

Still, it is possible that no such level exists. That is, it may be that some levels lower than the missing one have more than one representative. In that case we show that we always can help the on-line scheduler by the following *gift*: we promote some tasks upwards to higher value densities, i.e., giving them an additional value density during one stage. We choose the promotion in such a way that it leads to a situation that satisfies the above property. Then, we obtain the lower bound taking the gift into consideration. This bound surely applies for the weakened on-line scheduler (i.e., without the gift).

Here are the details of the promotion procedure. At the beginning of the stage, the on-line scheduler executes up to n tasks. The *promotion* works as follows: group the tasks currently executing according to their value density levels. Now, starting from level zero go up the levels until a level having the single representative property is found. If there is no task at level zero then level zero has the desired property. Otherwise, promote all *but one* of the tasks one level up to level one. Now, we repeat this procedure for level one: if there are no tasks at level one (taking into consideration tasks that were just promoted) then level one satisfies the desired property. If level one is not empty then we promote all but one of the tasks (if any) to the next level and repeat this process.

There are $n+1$ value density levels but only n (or fewer) tasks, hence this process must terminate producing a "promoted" schedule with a level that has the single representative property.

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 3.2 *For any stage $\mathcal{S}$, the effective value density obtained by the clairvoyant scheduler during $\mathcal{S}$ is at least $\frac{k}{(k-1)} n(k^{\frac{1}{n}} - 1)$ times greater than the effective value density obtained by the on-line scheduler for the same period.*

[8] If only one level is missing from the schedule then the single representative property is satisfied for that level. In general, however, recall that each level can have up to n representatives.

PROOF.
Suppose i_0 is the level having the single representation property for the period in question, possibly after performing the promotion procedure.

An easy lower bound on the value achieved by the clairvoyant algorithm is obtained by scheduling to completion n tasks of level i_0. This corresponds to an effective value density of $nk^{\frac{i_0}{n}}$.

Suppose $T_1, T_2, \cdots T_m$ with value densities $i_1 \leq i_2 \leq \cdots i_m < i_0$ are the tasks that were executing at the beginning of stage $\mathcal{S}$ [9]. 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 its promoted value density (denoted by $P(\cdot)$) then the denominator does not decrease hence the ratio does not increase.

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

The last equality is due to the fact that P is a one to one function from $\{i_1, i_2, \cdots i_m\}$ *onto* $\{1, 2, ...m\}$. 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. □

The lemma above demonstrates a ratio between the *effective value density* of any on-line scheduler and that of the clairvoyant scheduler during every stage. For an infinite game, this translates to a ratio between the *values* obtained by the algorithms during the entire game. However, we are interested in finite games: a problem arises with the end of the last stage. At the end of the last stage the on-line scheduler may still execute tasks from previous stages while the clairvoyant (according to our scenario) leave all the processors idle. The following theorem proves that after sufficient number of stages these "residual" tasks can be ignored.

Theorem 3.3 *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.

[9] It is possible that some of these tasks were released in previous stages.

Fix an on-line scheduling algorithm. Recall that t_l the beginning of the l'th stage, denote by $V(t_l)$ the value obtained by the on-line scheduler until time t_l. The ratio between the effective value densities as it appears in lemma 3.2 becomes a lower bound on the ratio between values, because the clairvoyant scheduler never abandons a task that started its execution while the on-line algorithm might. Hence, lemma 3.2 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 $nt_l \to \infty$). Hence we can assume that $V(t_l)$ tends to infinity. For arbitrarily small $\epsilon > 0$ there is a big enough l_0 such that

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

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 have 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,

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

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

Corollary 3.4 *As the number of processors n tends to infinity, no on-line algorithm can have a competitive multiplier smaller than* $\ln k$ *(natural logarithm).*

Remark 3.2 For $n = 1$ the lower bound is k which is not as good as the already known tight lower bound of $(1+\sqrt{k})^2$ [2, 13]. For $k = 1$ a different treatment is needed.

4 Algorithmic Guarantees

Having proved the lower bound on the best possible competitive multiplier, we would like to devise an on-line scheduler that achieves this bound. In the following we describe an algorithm that does so (up to a small multiplicative factor) in many cases. We assume that some upper bound on the largest possible importance ratio is known *a-priori.* This upper bound is provided to the scheduling algorithm and is denoted by k.

We break the processors into *bands* (of 2 processors each) and one *central pool.* The main idea of the algorithm is to attempt to assign a task, upon its release, to the band corresponding to its value density. Tasks that are assigned to a band are guaranteed to complete and can all complete on a single processor. This means that they constitute a uniprocessor underloaded system and can be scheduled according to the *earliest-deadline-first* algorithm [5]. Suppose the new task cannot be added to the band that corresponds to its value density (because it will cause overload at that band). Then the scheduler will determine whether the new task can be scheduled on the next band below (i.e, a band corresponding to lower value density). If the band below cannot accept the new task, the task will continue to *cascade* downwards. If a task cascades to the lowest band but still cannot 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 (*Latest Start Time*)[10], at which time it tries again to be scheduled (details to follow).

Throughout this section we assume a system with $2n$ 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 $[k^{\frac{i-1}{\psi}}, k^{\frac{i}{\psi}})$. We allocate 2 processors (*a band*) for each of the ψ value density levels. The remaining 2ω processors are allocated as a *central pool* that will be used by tasks of all levels.

The algorithm has three major components:

1. Upon task release, assign a task to a band if possible (possibly after cascading) making that task privileged.
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 *MOCA: Multiprocessor On-line Competitive*

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

Algorithm. In this variant, the central pool is also broken into bands of two processors each[11]. The *MOCA Algorithm* schedules according to the following rules:

- 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 contains 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.
  2. If T was not accepted by any band (including all the bands in the central pool) it enters *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[12].

- 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[13].
  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 the earliest deadline among all the tasks executing on these RPs.
     If T has a later deadline than T^* then abandon T^* and schedule T in its place; otherwise, abandon T [14].

- If, at a task completion event, the SP of a band becomes idle then the two processors should switch roles; the *safe-processor* becomes the *risky-processor* and vice versa[15]. Relabelling processors in this way does not require task migration.

Figure 2 is a schematic description of the *MOCA Algorithm*. The bands structure as described above gives priority to 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:

[11] 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 it awaits its LST.

[12] Using idle RPs and scheduling tasks of *Q_waiting* before they reach their LST is a heuristic that can improve the average case behavior of the scheduler.

[13] Heuristics can be used to choose the processor in case more than one RP is idle.

[14] If T is to be abandoned while there is an idle processor (above T's own band), scheduling T on an idle processor (with or without guaranteeing its completion) can only improve the average case behavior of the scheduler.

[15] The current task on SP (that was RP) becomes privileged.

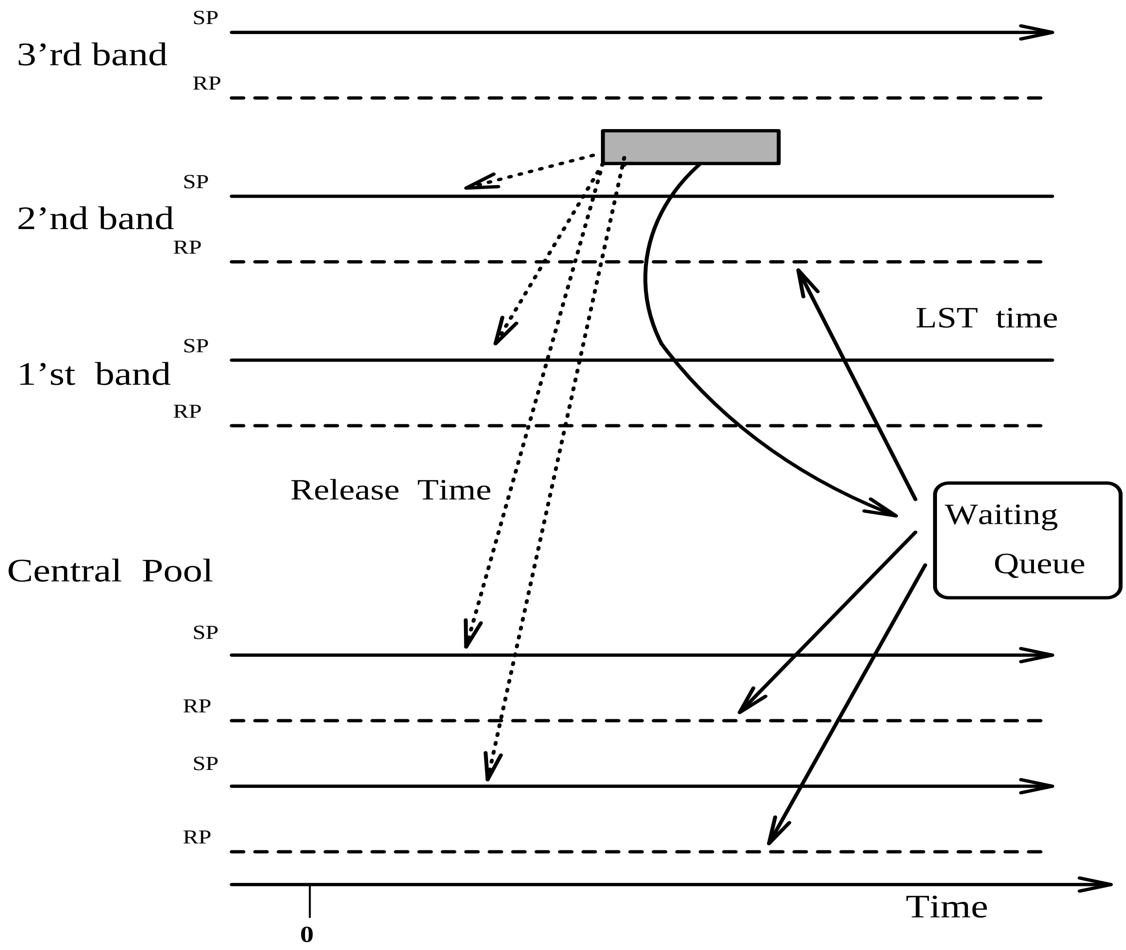

Figure 2: A schematic description of the *MOCA Algorithm*.
In this figure, the system has 10 processors divided into 3 bands and a central pool. At release time a task tries to be scheduled on one of the SPs starting with its own value density band. If unsuccessful, it awaits its *LST* in *Q_waiting*. At *LST* the task tries to be scheduled on one of the RPs, again starting from its own value density band.

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

Table 1: The tasks for example 4.1.

downwards[16]. An algorithm that uses the "pure" bands structure (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 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 *MOCA Algorithm*. An analogy to everyday life is to a well-balanced corporate research and development policy: a few researchers should work on high risk/high value research whereas most employees should work on bread and butter concerns.

Example 4.1 The following is a small example of the *MOCA Algorithm*'s scheduling. Assume that the highest possible value density is 16, number of processors is 6 of which 2 are allocated as a central pool and the rest constitute 2 bands (i.e., $k = 16, 2n = 6, \psi = 2$ and $\omega = 1$). 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 table 1. Figure 3 shows the schedule created by the *MOCA Algorithm*.

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 cannot 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 unless we invoke heuristics).

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 cannot 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 later deadline). T_9 cannot be scheduled on any of the SPs; it reaches its LST and is scheduled on the RP of the

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

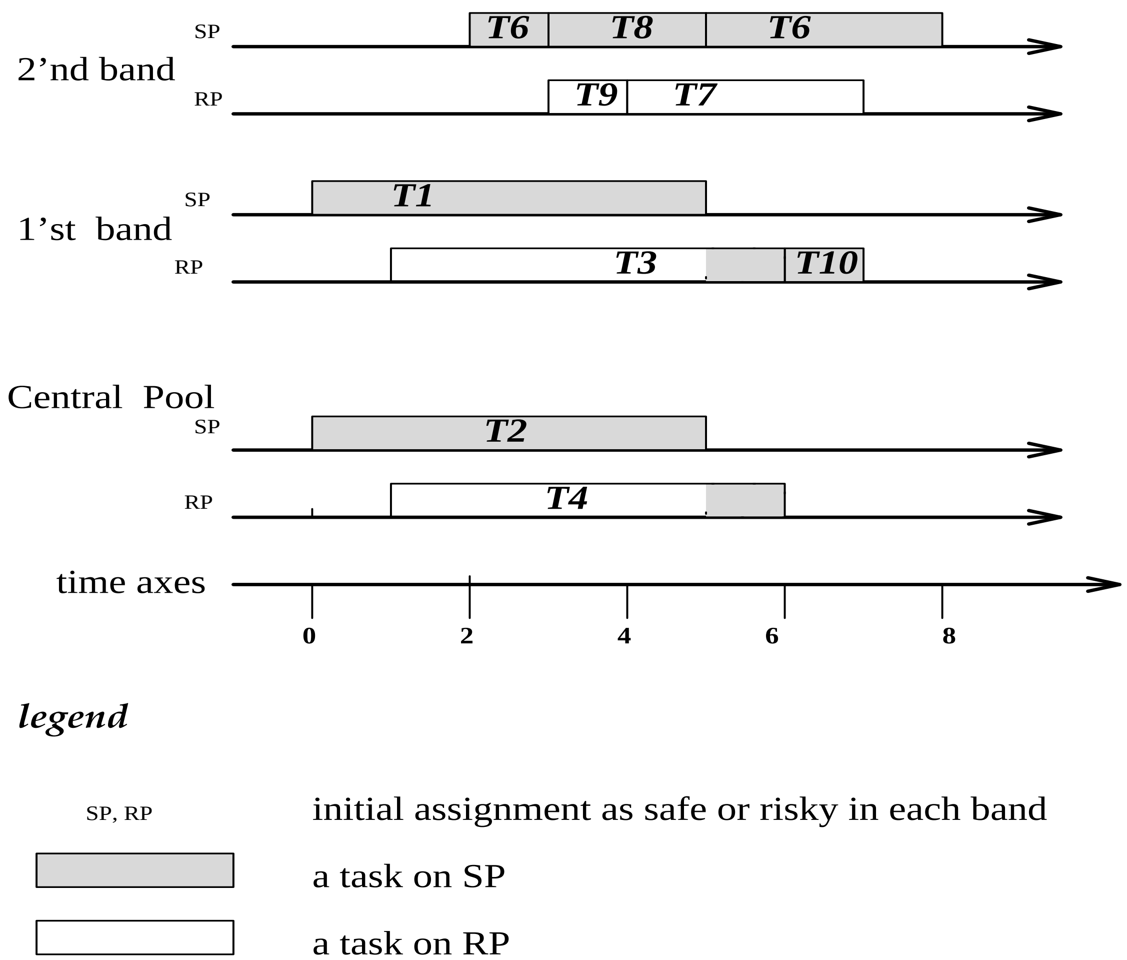

Figure 3: The *MOCA Algorithm* scheduling for example 4.1.

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

At time 5, the SP of the first band becomes idle, which creates a switch of roles between the SP and RP of that band. Later at time 6, T_{10} is released; it cannot 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 *MOCA 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). As we have pointed out in the notes, heuristics should be used to increase the performance in practice. We have downplayed these heuristics because they do not improve the worst case guarantees.

5 The Algorithm's Competitive Multiplier

In this section we would like to study the behavior of the *MOCA Algorithm* and to compute its competitive multiplier. The final result is stated in theorem 5.3. Before we start we introduce the *lost value lemma* as well as some notation and definitions.

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, respectively.

Lemma 5.1 *The Lost Value Lemma*

If for some constant α and every set of tasks Γ,

$$C(F) \leq \alpha V(\Gamma)$$

Then,

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

PROOF.

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

□

Definition 5.1

- PRODUCTIVE BAND: A band is said to be *productive* at time t if at that time, its SP is not idle.

  Recall that tasks that start executing on SP are never abandoned. This mean that whenever SP is not idle it "generates" value (i.e., is productive).

- 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 [17] .

Recall that Γ is the entire set of tasks to be scheduled. We partition the tasks of Γ according to the behavior of the *MOCA Algorithm*: tasks that never completed (F) and tasks that completed successfully (S). Denote by $V(\Gamma)$ the value achieved by the *MOCA Algorithm*.

We would like to show that for any task that was abandoned by the *MOCA Algorithm* there are other tasks with "enough value" that were completed. This will show that $C(F) \leq \alpha C(\Gamma)$ (for some constant α). Using the Lost Value Lemma (5.1) we will get a competitive multiplier of $\alpha + 1$. The following technical lemma is used in item 2 below.

Lemma 5.2 *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.*

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 since it was scheduled by an LST interrupt). □

Here are a few things to notice about the *MOCA 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 $T(r, c, d)$[18]. Upon T's release it was not accepted by any of the levels on or below

[17] 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 $2nk$.

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

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 5.2). Combining the two gives the desired result. $\square$

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 on the tasks of F is bounded by $2nk^{\frac{high(t)}{\psi}}$.

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[19]:

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

This leads to the following theorem.

Theorem 5.3 *For a system with $2n$ processors and maximal value density of $k > 1$ the MOCA 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,

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

[19] For the case $k = 1$, the *uniform value density case*, see remark 5.2 below.

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

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

Using the Lost Value lemma we get the desired result. $\square$

Remark 5.2

- Note that the *MOCA Algorithm* does not use migration, hence the previous result holds both whether migration is allowed or not.
- When $k = 1$, there is no need for the bands structure, hence the central pool consists of all the processors (in our notation $\omega = n - 1$ and $\psi = 1$). This leads to a competitive multiplier of $2 + 1$ (when some tasks may have slack time). For $n = 2$ this corresponds to our results for two processor systems [10].
- When the number of processor is odd, a similar result can be obtained. For a system with $2n+1$ processors, create bands and pool from the first $2n$ processors. The left over processor can be used, for example, as a second SP for one of the bands. This leads to a bound of:

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

  However, this result does not specialize to a uniprocessor system because at least two processors are needed to create a band.

5.1 Setting ψ

We will estimate the complex expression for the upper bound on the competitive multiplier given by theorem 5.3 by setting[20] [21] $\psi = n\frac{\ln k}{\ln k+1}$ (hence $\omega = n\frac{1}{\ln k+1} = \frac{\psi}{\ln k}$). The bound in (2) above becomes:

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

[20] ln is the natural logarithm

[21] Numerical experiments have shown that this setting of ψ is a close approximation to the optimal setting.

The left hand side is obtained by multiplying both numerator and denominator by $(k^{\frac{1}{\psi}} - 1)$. The maximum[22] denoted by the equation 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 5.4 *The MOCA Algorithm has a competitive multiplier of at most*

$$1 + 2n(k^{\frac{1}{\psi}} - 1) \tag{4}$$

where $\psi = n\frac{\ln k}{\ln k+1}$
(recall that the lower bound is bigger than $2n(k^{\frac{1}{2n}} - 1)$*)* □

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

$$1 + 2\lim_{\psi\to\infty} \psi\frac{\ln k + 1}{\ln k}(k^{\frac{1}{\psi}} - 1) = 1 + 2\frac{\ln k + 1}{\ln k}\ln k = 2\ln k + 3 \tag{5}$$

Corollary 5.5 *For given n and k, the ratio between the lower bound and the algorithmic guarantee is at most*

$$\frac{1 + 2n(k^{\frac{1}{\psi}} - 1)}{\frac{k}{k-1}2n(k^{\frac{1}{2n}} - 1)}, \quad \textit{where } \psi = n\frac{\ln k}{\ln k + 1} \tag{6}$$

When k is held fixed and n tends to infinity this ratio tends to $\frac{k-1}{k}\left(2 + \frac{3}{\ln k}\right)$. *This is less than* 3.2 *for all $k > 1$ and tends to 2 as k tends to infinity.*

PROOF.
Recall our lower 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:

$$\frac{2\ln k + 3}{\frac{k}{k-1}\ln k} = \frac{k-1}{k}\left(2 + \frac{3}{\ln k}\right) \tag{7}$$

This gives the desired result. □

Figure 4 gives a graphical representation of the above result.

Remark 5.3 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 $n\frac{\ln k}{\ln k+1}$

[22] Define $f_a(x)$ to be $\frac{x}{a+(x-1)}$. When $a > 1$, this function is monotone increasing with x ($x \geq 0$). Let a be $\frac{\psi}{\ln k}(k^{\frac{1}{\psi}} - 1)$. Then a is bigger than 1, 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.

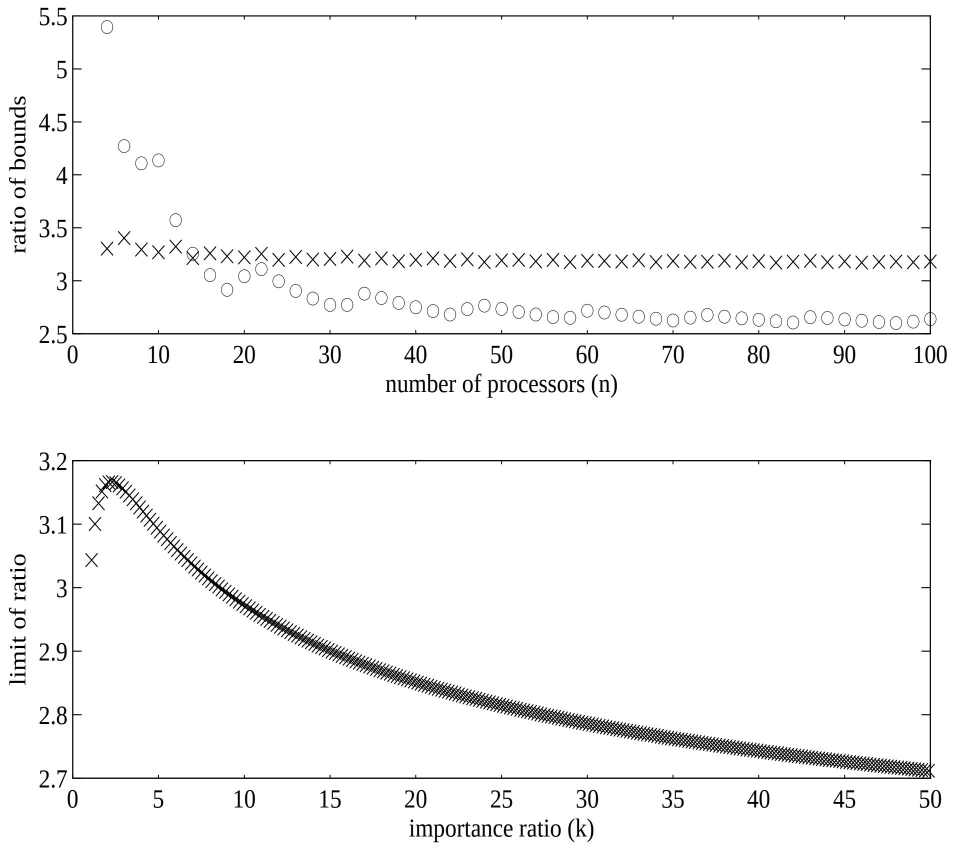

Figure 4: The ratio between the value guaranteed to be obtained by the algorithm and the lower bound for varying number of processors and importance ratios.
The upper graph shows the ratio (equation 6 above) for $k = 2$ (the 'x's) and $k = 256$ (the 'o's) for varying number of processors.
The lower graph shows the *limit* as the number of processors n tends to infinity of the ratio between the algorithmic guarantee of the *MOCA Algorithm* and the lower bound (equation 7 above) for varying importance ratios.

5.2 How Distributed is the 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 *MOCA 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[23]. 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 [22] to distribute the functionality of the centralized scheduler among the processors even further. This is an interesting and important extension to the work presented here.

5.3 The Scheduling Overhead

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

What is the cost of testing whether a newly arriving task can be added to *Q_privileged* containing N tasks without 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 Conclusion

Table 2 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 *MOCA Algorithm* and the lower bounds we have proved (see figure 5). The algorithmic guarantee is within a small multiplicative factor

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

number of processors	importance ratio	bounds: lower bound	bounds: algorithmic	comments
1	: any $k \geq 1$	$(1+\sqrt{k})^2$ [3, 2]	tight [13]	tight bound achieved by D^{over}.
2	: 1 : (uniform)	2	tight	tasks have *no* slack time and may not migrate between processors [2, 21].
2	: 1 : (uniform)	2	3 [10]	tasks may have slack time but may not migrate between processors.
2	: 1 : (uniform)	2	tight [10]	tasks may have slack time and may migrate between processors.
$n \geq 2$	: $k > 1$:	$\frac{k}{(k-1)} n(k^{\frac{1}{n}} - 1)$ ‡	$1 + n(k^{\frac{1}{\psi}} - 1)$ ‡ where $\psi = \frac{n}{2} \frac{\ln k}{\ln k+1}$	bounds are tight within constant coefficient for many cases. Exact algorithmic guarantee can be seen in theorem 5.3
$n >> 2$	: $k > 1$	$\frac{k}{(k-1)} \ln k$ ‡	$2 \ln k + 3$ ‡	asymptotic behavior

Table 2: State of the art of competitive real time scheduling.

(about 3.2) from the lower bound for large enough n and k (figure 4 and figure 5 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.

It is possible that a better choice of ψ will lead to a better exact expression of the algorithmic guarantee for our algorithm (section 5.1). But we cannot do better, asymptotically, 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 *MOCA Algorithm* automatically loses a factor of 2 compared to a clairvoyant scheduling algorithm that utilizes *all* the processors concurrently. Of course, one can introduce heuristics extending those we have suggested that will use a processor whenever possible. The theoretical 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 $m \geq 2$).

Our adversary arguments and algorithms offer two useful insights:

1. A parallel on-line scheduling algorithm achieves a competitive guarantee by allocating some processing resources according to tasks' value density. This is a qualitative difference from our uniprocessor scheduling algorithm D^{over} [11, 13] which made its decisions based on total value only. Moreover, high value density tasks in the *MOCA Algorithm* have priority over lower value density tasks in the sense that they have more processors on which they can be scheduled due to the cascading.

2. The lower bound on the best possible competitive multiplier (as measured by our adversary arguments) converges to $\frac{k}{k-1} \ln k$ 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. In both cases, the competitive multiplier far better than it is for unipro-

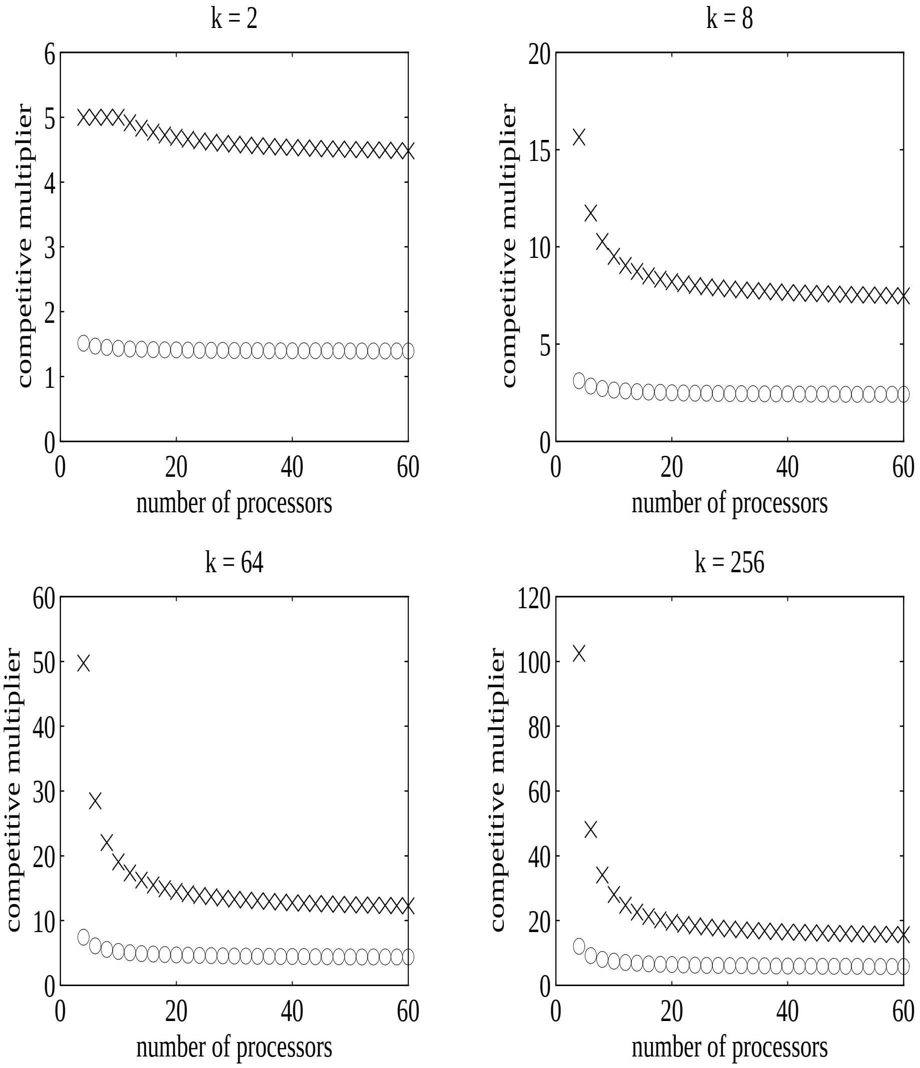

Figure 5: Comparison of competitive multipliers of lower bounds (o's) and algorithmic bounds (x's) for fix k's and varying number of processors.

cessors (i.e. $(1+\sqrt{k})^2$). Since effective utilization usually deteriorates in parallel situations, this was suprising to us. (see figures 4 and 5).

We assumed here that an *a priori* bound on the importance ratio of a system is known to the scheduler. What algorithmic guarantee can be proven when this assumption is relaxed?

An important issue is how to account for migration overhead in multiprocessor environments. For example, we modeled distributed memory architectures by forbidding migration, but that is clearly too strong a restriction. Permitting migration at a cost is a subject for further research.

The *MOCA Algorithm* is described as a multiprocessor algorithm with a static number of processors. Fault tolerance issues can be addressed by the following techniques: First, one can keep some processors in reserve and introduce them as other ones fail. While in reserve the processors can be used as a secondary pool for tasks that were not accepted by the primary structure of bands and pool. Another way to utilize additional reserve processors is to add a third processor to a two-processor band. The third processor can be a mirror processor for the safe processor waiting to take over in case that one of the band's processor fails. Second, as processors fail, one can statically reset the algorithm to have a different number of bands and/or pool of shared processors. Combinations of these techniques and additional heuristics may give rise to promising algorithms.

In the absence of failures, but in the face of a large load, our algorithm gives *worst case guarantees* which is an important aspect of dependable real-time system.

Acknowledgments

We would like to thank Bud Mishra for helpful discussions and insights and Doug Locke for practical enlightenment. We would also like to thank the anonymous referees for their many helpful comments.

References

[1] N. Audsley and A. Burns. Real time system scheduling. Computer Science Department Technical Report No. YCS134, York University, UK, Jan. 1990.

[2] 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(2):124–144, 1992. Conference version appeared in Proceedings of the 12th Real-Time Systems Symposium ,pages 106-115, San Antonio, Texas, Dec. 1991.

[3] S. Baruah, G. Koren, B. Mishra, A. Raghunathan, L. Rosier, and D. Shasha. On-line scheduling in the presence of overload. In *Proceedings of the 32nd Annual Symposium on the Foundations of Computer Science*, pages 101–110, San Juan, Puerto Rico, Oct. 1991. IEEE.

[4] S.-C. Cheng, J. A. Stankovic, and K. Ramamritham. Scheduling algorithms for hard real-time systems: A brief survey. In J. A. Stankovic and K. Ramamritham, editors, *Hard Real-Time Systems: Tutorial*, pages 150–173. IEEE, 1988.

[5] M. L. Dertouzos. Control robotics: the procedural control of physical processes. In *Proceedings IFIF Congress*, pages 807–813, 1974.

[6] M. L. Dertouzos and A. K.-L. Mok. Multiprocessor on-line scheduling of hard-real-time tasks. *IEEE Transactions on Software Engineering*, 15(12):1497–1506, Dec. 1989.

[7] 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.

[8] J. R. Haritsa, M. J. Carey, and M. Livny. On being optimistic about real-time constraints. In *Proceedings of the PODS Conference*, pages 331–343, Nashville, TN, Apr. 1990. ACM.

[9] A. Karlin, M. Manasse, L. Rudolph, and D. Sleator. Competitive snoopy caching. *Algorithmica*, 3(1):79–119, 1988.

[10] G. Koren. *Competitive On-Line Scheduling for Overloaded Real-Time Systems*. PhD thesis, Computer Science Department, Courant Institute, NYU, New York, NY, Sept. 1993.

[11] G. Koren and D. Shasha. D-over: An optimal on-line scheduling algorithm for overloaded real-time systems. *SIAM Journal on Computing*. to appear.

[12] G. Koren and D. Shasha. An optimal scheduling algorithm with a competitive factor for real-time systems. Computer Science Department Technical Report No. 572, Courant Institute, NYU, New York, NY, July 1991.

[13] G. Koren and D. Shasha. D-over: An optimal on-line scheduling algorithm for overloaded real-time systems. In *Proceedings of the 13th Real-Time Systems Symposium*, pages 290–299, Phoenix, Arizona, Dec. 1992. IEEE.

[14] E. L. Lawler and C. U. Martel. Scheduling periodically occurring tasks on multiple processors. *Information Processing Letters*, 12(1):9–12, Feb. 1981.

[15] C. D. Locke. *Best-Effort Decision Making for Real-Time Scheduling*. PhD thesis, Computer Science Department, Carnegie-Mellon University, Pittsburgh, PA, 1986.

[16] A. K.-L. Mok. *Fundamental Design Problems of Distributed Systems for the Hard Real-Time Environment*. PhD thesis, Department of Electrical Engineering and Computer Science, M.I.T, Boston, MA, May 1983.

[17] R. Rajkumar, L. Sha, and J. P. Lehoczky. Real-time synchronization protocols for multiprocessors. In *Proceedings of the 9th Real-Time Systems Symposium*, pages 159–269. IEEE, Dec. 1988.

[18] L. Sha and S. S. Sathaye. A systematic approach to designing distributed real-time systems. *IEEE Computer*, pages 68–78, September 1993.

[19] D. Sleator and R. Tarjan. Amortized efficiency of list update and paging rules. *Communications of the ACM*, 28:202–208, Feb. 1985.

[20] J. A. Stankovic and K. Ramamritham. The spring kernel: A new paradigm for real-time systems. *IEEE software*, pages 62–72, May 1991.

[21] F. Wang and D. Mao. Worst case analysis for on-line scheduling in real-time systems. Department of Computer and Information Science Technical Report No. 91-54, University of Massachusetts, Amherst, MA, June 1991.

[22] H. Zhou, K. Schwan, and I. F. Akyildiz. Performance effects of information sharing in a distributed multiprocessor real-time scheduler. In *Proceedings of the 13th Real-Time Systems Symposium*, pages 46–55, Phoenix, Arizona, Dec. 1992. IEEE.