Competitive On-Line Scheduling for Overloaded Real-Time Systems

by

Gilad Koren

A DISSERTATION SUBMITTED IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

DEPARTMENT OF COMPUTER SCIENCE
NEW YORK UNIVERSITY

SEPTEMBER 1993

Approved

Dennis Shasha
Advisor

Bhubaneswar Mishra
Co-Advisor

Dedication

I would like to dedicate this work to the memory of my beloved uncle Ignatz Brand who passed away while this work was in progress.

Acknowledgments

This work was made possible by a big investment on the part of my advisor Dennis Shasha both in time and talent. Dennis was constantly pushing forward, raising new questions and suggesting new ideas. I owe him a lot for showing me the way through graduate school (including crossing the Atlantic to Paris and back). Karen and Dennis Shasha became part of my extended family. I enjoyed their international Thursday night dinners in Paris (where I played the dish washer role) and many other occasions where we spent time together.

I would like to thank my co-advisor Bud Mishra for his guidance and encouragement. Bud spent many hours with me, always contributing a new and clear perspective of the problems and the solutions. It was Marc Donner that introduced my advisors to the field of *Real Time Scheduling* in a seminar he gave in NYU (which I regret deeply for having overlooked at that time). Marc has been very supportive throughout this work, willing to listen and to give advice always with a smile. Some of this work was done while visiting in INRIA, Rocouncourt, France. I would like to thank Patrick Valduriez and all the people of Project Rodin for their hospitality.

My aunt and uncle Frega and Ignatz Brand helped me in all possible ways through out my New York experience, always willing to listen and to support. I should not forget my good friends who had a big stake in my life during graduate school: Gadi, Eddie, Arnon, Andres and last but not the least Jenifer (with one 'n'). Vekamuvan hamishpahca ba'aretz she'azra li harbe: Ima, Neomi, Srulik, Shachar and Tali.

The illustration is by Marianna Trofimova.

Work on this dissertation was 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.

Abstract

We study *competitive on-line scheduling* in *uniprocessor* and *multiprocessor* real-time environments. In our model, tasks are sporadic and preemptible. Every task has a deadline and a value that the system obtains only if the task completes its execution by its deadline. The aim of a scheduler is to maximize the total value obtained from all the tasks that complete before their deadline.

An *on-line* scheduler has no knowledge of a task until it is released. The problem is to design an on-line scheduler with worst case guarantees even in the presence of overloaded periods. The guarantee is given in terms of a positive competitive factor. We say that an on-line algorithm has a competitive factor of r, $0 < r \leq 1$, when under all possible circumstances (i.e, task sets) the scheduler will get at least r times the best possible value. The best value is the value obtained by a *clairvoyant* algorithm. In contrast to an on-line scheduler, the clairvoyant algorithm knows the entire task set *a priori* at time zero.

When a uniprocessor system is underloaded there exist several optimal on-line algorithms that will schedule all tasks to completion (e.g., the Earliest Deadline First algorithm). However, under overload, these algorithms perform poorly. Heuristics have been proposed to deal with overloaded situations but these give no worst case guarantees.

We present an optimal on-line scheduling algorithm for uniprocessor overloaded systems called D-over. D-over is optimal in the sense that it has the best competitive factor possible. Moreover, while the system is underloaded, D-over will obtain 100% of the possible value.

In the multiprocessor case, we study systems with two or more processors. We present an inherent limit (lower bound) 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 factor from the best possible guarantee in many cases.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 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 station 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 homes.

1.1.1 Real-Time Systems

A real-time system is usually one that controls and/or monitors a physical (real-world) process. This means that the system gathers information from external sensors. It processes this information and then usually performs some action. The nature of the physical process might dictate a strict time limit for the system to respond. If this time limit is passed—for the monitoring functions— then the information from the sensors would be lost or outdated; for the controlling functions a missed deadline might mean that the action eventually taken is not appropriate any more.

This leads to the notion of *deadline* which is a common thread among all real-time system models and the core of the difference between real-time systems and time-sharing systems. The deadline of a task is the point in time before which the task must complete its execution.

1.1.2 Scheduling

An essential component of a computer system is the *scheduling mechanism*, that is the strategy by which the system decides which task should be executed at any given time. The problem of real-time scheduling is different from that of multiprogramming time-sharing scheduling because of the role of *timing constraints* in the evaluation of the system performance. Normal multiprogramming time-sharing systems are expected to process multiple job streams simultaneously, so the scheduling of these jobs has the goals of *maximizing throughput* and *maintaining fairness*. In real-time systems the primary performance is not to maximize throughput or maintain fairness, but instead to perform critical operations within a set of user-defined *critical time constraints* [29].

When a system can meet all its tasks' deadlines we say that this set of tasks is *schedulable* and the system is *underloaded.* Otherwise, if at least one task cannot meet its deadline then the system is *overloaded.* A common approach, in practical systems, to deal with overload is to try to prevent it. This is done by ensuring that an abundance of processing power is available and is sufficient to handle the *worst* possible situation (i.e. load). The problem with this approach is that such systems are extremely inefficient and therefore expensive. Moreover, overload can still arise either as the result of failures of some computational resources or as a transient condition (e.g., an overloaded communications circuit). As a result, some important deadlines might be missed, resulting in unpredictable failures. We would like to have *schedulers* that would minimize the need for processing resources with two properties: (i) in an underloaded environment they would schedule all tasks to completion and (ii) in the presence of overload, the damage to the overall system performance will be minimal and predictable.

In addition to its deadline a task can be characterized by the following parameters: its *release time* (sometimes referred to as *start-time* or *request time*), its *computation time*, its *period*[1] for a periodic task and its *priority* if priorities are used[2]. Additionally each task can be associated with some *value*; this value will be obtained if the task completes prior to its deadline. A scheduler uses the task's parameters in its decision making, it is said to be an *on-line scheduler* if its decisions do not depend on *a priori* knowledge of future requests. In other words, the parameters of a task are not known prior to its release time.

Naturally, one cannot predict the entire system behavior at the system design stage. For that reason on-line scheduling suggests itself as a viable and important field of research. The basic problems that we address in this context are: The *feasibility problem*: given set of tasks, how can we test that this set is *schedulable*? What is the complexity of such a test? Which on-line schedulers are optimal for overloaded and underloaded systems? What are the time and storage complexities of these schedulers? And, most important, what performance guarantees can a scheduler give in a overloaded system? Of course, the answers to the above-mentioned questions vary greatly depending on the assumptions of the model under consideration. In the next sections we describe some models that were

[1]A task is called *periodic* if it has regular request times i.e. there is a constant time interval between consecutive request for that task. This interval is the *period* of this task.

[2]Another parameter is the *laxity* of a task (also called the *slack time*), distance to its deadline minus its remaining-computation-time. Hence, the laxity of a task is a measure of its urgency— a task with small laxity would have to be scheduled soon in order to meet its deadline.

studied, later we present our model.

1.2 Background

The literature presents a wide variety of real-time models corresponding to abstractions of real-world real-time systems. Different models have different, sometimes contradictory, assumptions. In the sequel we will list the main characteristics (parameters) of real-time systems. Different models can be characterized by the choices made for each of these parameters.

- Hard, Soft and Firm Real-time Systems.

  In a real-time system, when a task is requested to do some service there is a time limit associated with this request. If this time limit elapses before the task completes its execution, the task has failed. This failure might lead to a total collapse of the system in which case we say that this is a *hard real-time* system. For example, in a nuclear power plant, a delay in the response of the task that is responsible for cooling the overheated reactor can have catastrophic results. Systems in which deadlines may occasionally be missed with only degradation in performance of the entire system but not a complete failure are called *soft real-time* systems. Sometimes, in a soft real-time system a task that missed its deadline should nevertheless be completed i.e. its service though late is still valid and helpful. For example suppose an aircraft's position must be computed every 100 milliseconds to ensure a positional accuracy of 25 meters. A delayed position update might result in a loss of positional accuracy, while missing it altogether would exacerbate the loss—Locke [29]. In a special kind of soft real-time, called *firm real-time*[3], if a task missed its deadline its response has no value–it is not helpful at all [4,11]. For example, suppose a task is responsible for collecting the characters received by an antenna. This antenna has very limited storage, hence if the task is late some characters would be missed and the transmission is lost. Our work here assumes **firm deadlines**.

  In a hard real-time system overload should never occur, hence one must use an optimal scheduler and supply it with enough processing power for the worst-case

[3]Other papers [5] denote such deadlines as *hard*. The reader should therefore be aware of the definitional variations.

scenario. Optimal schedulers (for the underloaded case) are described in Liu and Layland [28], Dertouzos [7] and Mok [30]. In a soft real-time system some tasks might miss their deadlines. The scheduler has the difficult task of deciding which tasks should be aborted in order to maximize the overall throughput of the system. In a system with mixed hard and soft deadlines the tasks with hard deadlines are called *critical tasks* [37]. In this case, the scheduler is required to meet the deadlines of these tasks even in the presence of overload. A scheduler that satisfies this condition is called *stable* [33].

- Periodic vs. Aperiodic Tasks.

  Periodic tasks are common in many practical real-time systems. If a system is comprised of only periodic tasks then the scheduling problem becomes easier because of the regularity of events. Optimal algorithms for pure periodic systems were the first to be found (Liu and Layland [28]). Unfortunately a purely periodic system is an unrealistic model—some tasks in any system are non-periodic (such as tasks that handle emergency situations, operator commands etc.). If the non-periodic tasks are of low importance (i.e. have soft deadlines) they can be treated as *background tasks*. They will be scheduled whenever the processor is not being utilized by the periodic tasks [28] or will be scheduled using more sophisticated *sporadic servers* [36]. However, in a realistic system some of the sporadic (aperiodic) tasks will definitely have high importance and hard deadlines, in this case one tries to translate aperiodic tasks into equivalent periodic tasks based on the worst case frequency and computation demands of the aperiodic tasks [30]. In this work we assume that all tasks are **aperiodic**.

- Preemptive vs. Nonpreemptive Tasks.

  We assume that a task can be preempted at any time. This is a realistic assumption since most real-time operating systems enable preemption. If preemption is not possible the scheduling problem is easier since there is less the scheduler can do [14].

  However, among the studies that assume preemption there are differences as to the treatment of the cost associated with preemption (i.e. task-switching). Some assume that task-switching takes virtually no time at all (e.g., [8,23,28] and our current work);

others try to account for the task switching time by adding it to the processing time of the preempted tasks [8,28,29,37].

- Uniprocessor vs. Multiprocessor Systems

  For the uniprocessor system a variety of optimal schedulers were presented as well as heuristics for scheduling an overloaded system. The problem becomes much more difficult in a multiprocessor system (see also sections 1.2.4 and 1.4.1). Mok and Dertouzos [8] showed that optimal on-line scheduling algorithm does not exist in multiprocessor environment.

- Knowledge of Task's Parameters *Start-time, Computation-time, Deadline, Period and Value.*

  Virtually all schedulers assume some knowledge of the task's parameters. This knowledge can be exact or stochastic. If parameters are known *a-priori* then an optimal scheduling sequence can be found at compile time (Mok and Dertouzos [8] showed that this a-priori knowledge is necessary in a multiprocessor system). In practice, however, there are many occasions where the parameters of the tasks are not known beforehand. Even if such information is available the computation might be NP-hard [8].

- Dynamic vs. Static Scheduling.

  This dichotomy is manifested in priority-driven algorithms. If static priority assignment is used then once a task is released a priority is assigned to it. This priority cannot be changed[4]. If a dynamic scheduler is used, then the priority assignment of a task can be changed at any time.

  Liu and Layland [28] presented a static priority-driven algorithm for a purely periodic system. They also describe a *mixed* scheduling algorithm—some priorities are fixed and some are dynamic. Sha, Rajkumar and Lehoczky [34] study the use of dynamic priority-driven schedulers for the *priority inversion problem.* The feasibility and complexity of fixed [27] and dynamic [23,26] priority scheduling have been studied.

- Independent vs. Dependent tasks.

[4]In addition, all requests for a specific periodic task always have the same priority (Liu and Layland [28]).

Tasks are said to be *independent* if the request for a certain task does not depend on the initiation or completion of requests for other tasks and also no task need wait for an action to be taken by another task in order to continue its execution. Otherwise, tasks are said to be *dependent*. A possible source of dependency between tasks is the need for synchronization between tasks. For example, suppose a semaphore is used to force mutual exclusion between tasks making access to some shared data object. Once one task holds the lock for this semaphore, all other tasks that need access to this shared data object must wait and are said to be *blocked*. Another example of dependency occurs when a task requests some service (e.g. I/O) and this service is given according to a FIFO queue, hence a task must wait for the completion of all earlier requests.

Sha, Rajkumar and Lehoczky [34] investigate the *priority inversion problem* that arises from the use of semaphores to utilize mutual exclusion. Mok [30] showed that with mutual exclusion constraints it is impossible to find an optimal *on-line* scheduler. He also showed that the following problem is NP-hard: deciding whether a set of periodic tasks that use semaphores is feasible.

We have presented eight parameters in which real-time models can differ. Of course, there are other important issues (e.g., fault-tolerance) that are beyond the scope of this work. For a survey of scheduling issues for uniprocessor and multiprocessor systems see Audsley and Burns [1] and Cheng et. al. [5].

1.2.1 Optimal Scheduling Algorithms

First, let us describe optimal schedulers for the uniprocessor environment, later we will discuss multiprocessor scheduling.

1.2.2 Rate Monotonic Scheduling

Liu and Layland [28] presented the *rate monotonic priority assignment scheduling algorithm*. They assumed the following:

- Uniprocessor environment.
- All tasks are periodic.

- The deadline for each task coincides with the end of its period.
- Deadlines are hard.
- The computation time for each task is constant for that task and does not vary with time.
- Tasks are independent.
- Tasks are preemptible and task-switching takes no time.

Recall that a priority-driven scheduler is *static* if the priority assignment of a task is fixed throughout its computation. The rate-monotonic (RM) scheduling algorithm is a static priority-driven scheduler. A task is assigned a priority according to the *length of its period* so that tasks with shorter periods are assigned higher priorities. At any given moment the task with the highest priority is executed. Thus, the priority assignment is independent of the semantic importance and the computation time of the tasks. Liu and Layland proved that RM is optimal among all *static* preemptive scheduling algorithms for periodic tasks with hard deadlines. This means that a task set which cannot meet its deadlines with RM will not be able to be scheduled with any *fixed priority assignment* scheduler. The work of Liu and Layland has been extended to include:

- Aperiodic tasks [6,24,30,36].
- Deadlines occurs before the end of the period (see the *deadline monotonic* priority assignment [36]).
- Dependency between tasks (e.g., using semaphores [30,34]).
- Multiprocessor environments [9,23,31].

1.2.3 Time-driven Optimal Schedulers

Liu and Layland [28] studied a *dynamic* scheduling algorithm for periodic tasks system - the *earliest-deadline-first* (D) algorithm. This algorithm schedules at every instant the task with the nearest deadline. They proved that earliest-deadline-first is an optimal scheduling algorithm in their model (periodic tasks). Note that RM is optimal only among the *static* schedulers so D may schedule all tasks in a case where RM cannot[5]. Dertouzos [7] showed that D is optimal even in the presence of non-periodic tasks. He assumed (i) arbitrary request and deadline times for each task. (ii) arbitrary and unknown to the scheduler execution time for each request and (iii) underload. Mok [30] proved that the *least-slack* (LS) algorithm is also an on-line optimal algorithm. LS schedules at any time the task with the least slack time (see footnote 2 for definition of slack time). However, one additional assumption is needed - all time parameters are non-negative integers. All requests times, computation times and deadlines are integers; also, preemption is possible only at integral time instants[6]. This assumption is not needed for D.

D has two major advantages over LS. The first is that D is driven by deadlines alone and does not require knowledge of the computation time while LS needs both. The second is that LS tend to generate frequent preemptions[7].

It might be impossible to find an on-line optimal algorithm when any of the above assumptions is relaxed. For example, Mok [30] showed that with mutual exclusion constraints (i.e. tasks are not independent) it is impossible to find an optimal on-line scheduler for uniprocessor.

1.2.4 Multiprocessor Optimal Scheduling

D and LS are not optimal in the multiprocessor case. Their optimality proofs do not transfer from the uniprocessor setting, since they lead to situations in which the same task is scheduled in two or more processors at one time instant (Dertouzos and Mok [8]).

Multiprocessor real-time scheduling is an active field of research. Both shared memory [23,31] and distributed [30,37,39] architectures have been studied. *Static binding* of tasks to processors (i.e., no migration) is assumed in some studies [30,31] while dynamic binding is assumed in others [8,23].

Dertouzos and Mok [8] proved that for two processors or more, an optimal scheduling algorithm must have *a priori* knowledge of the request times, hence *no* on-line optimal algorithm is possible in the multi-processor case. They also showed that once a task is

[5]An example can be found in Liu and Layland's paper [28, p. 188].

[6]This assumption can be justified since in practice, time parameters are presumably given in integral multiples of a basic time unit e.g., a processor instruction cycle.

[7]For example, look at the case where two tasks both have the least laxity. Each task will execute for one time unit and then will be preempted by the other.

released an optimal scheduler must know its deadline and computation time. Hong and Leung [13] showed that for the special case where all tasks share the same deadline an optimal on-line scheduler exists. Henn [12] studies the problem of scheduling tasks with precedence constraints in uniprocessor and multiprocessor systems. In his model, all tasks are released at time zero. Leung [25] and Lawler and Martel [23] studied the feasibility and complexity of multiprocessor scheduling for periodic task sets.

1.3 Scheduling in the Presence of Overload

Overload is a necessary evil of real-time systems. An ideal scheduler would schedule all tasks to completion in an underloaded environment and would minimize the overall damage to the system performance in the presence of overload. Scheduling itself should incur little overhead.

When overload occurs, a scheduling algorithm *must* discard some tasks. This should be done in a way that maximizes the overall value of the system. Locke [29] suggested a heuristic called *best effort* (BE) in an attempt to approximate such a scheduler.

Locke assumed that tasks are independent, preemptible and have arbitrary arrival times. The execution time of a task is known only *stochastically.* Each task has a value associated with it, which is given as a *value function.* A value function is a continuous function of the task's completion time. Value functions can model various kinds of time-constraints, in particular firm and soft deadlines[8]. The *distributions* of the task parameters become known to the scheduler only upon the task release.

When the system is underloaded, BE operates like the earliest-deadline-first algorithm; however, if an overload condition is detected, BE abandons the tasks with the lowest *value density*[9] in order to bring the system back to normal load. Since the task parameters are known only stochastically all evaluations are probabilistic (e.g. the probability that the system is overloaded, the expected value-density of a task etc.). This makes Locke's algorithm much more complicated than the above description.

To evaluate the performance of BE, Locke executed a battery of elaborate tests. The

[8]For example, Locke considers value functions that increase to a point (referred to as the *critical point*) and then decrease corresponding to tasks for which completion should be delayed.

[9]The *value density* of a task is its value divided by its remaining-execution time. Tasks with higher value density produce more value per execution-time-unit.

tests concluded that BE performs very well in a wide range of environments and is comparable or better than the other schedulers it was tested against in most cases. The results suggests that BE is a practical heuristic. However, these are only statistical results and there are pathological situations where BE performs very poorly. These result brought us and other researches to study the question of on-line scheduling with worst-case guarantees even when the system is overloaded.

1.4 Our Model of Real-Time System

Here, we informally describe our assumptions. The formal model definitions and assumptions will come later in chapter 2.

We study on-line scheduling of systems of sporadic (aperiodic) tasks. Tasks are independent (i.e., no precedence constraints) and can be preempted at any time. A preempted task can later resume its work. We assume that preemption and resumption take no time[10] and scheduling algorithm incurs no overhead[11]. In our basic[12] 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. Hence, we assume a *firm* on-line real-time model. The goal of the scheduler is to obtain as much value as possible.

In the studies of *competitive analysis* [4,15,35], one can quantify the performance of an on-line algorithm by comparing it with a clairvoyant (off-line) algorithm. A clairvoyant scheduler [30] has complete *a priori* knowledge of all the parameters of all the tasks. A clairvoyant scheduler can therefore choose a “scheduling sequence” that will obtain the maximum possible value achievable by any scheduler[13]. We say (as in [4,15,35]) 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

[10]This is assumed for example by Liu and Layland [28], Lawler and Martel [23] and Dertouzos and Mok [8]. This is 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.

[11]This can be done be a special dedicated processor for the operating system scheduling activities

[12]Some extensions are considered, see appendixes C and B.

[13]Finding the maximum achievable value for such a scheduler, even in the uniprocessor case, is reducible from the knapsack problem [10]; hence is NP-hard.

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. Inherent bounds on the best possible competitive multiplier are devised (in Baruah et. al. [3] and in section 4.1). Our goal is to devise on-line algorithms with worst case performance guarantees as close as possible to the inherent bounds.

1.4.1 Competitive On-Line Schedulers

We need some terminology in order to state the known results in competitive on-line scheduling:

Notation 1.4.1

- VALUE DENSITY The *value density* of a task is its value divided by its computation time.
- IMPORTANCE RATIO The *importance ratio* of a collection of tasks is the ratio of the greatest value density to the least 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.

Koren et. al. [4,16] suggested the first on-line scheduling algorithm with a performance guarantee for an overloaded system. They assumed a simplified variation of the task model that assumes *uniform value density*. This algorithm is called *D-star* (D^*) since it behaves like *earliest-deadline-first* (D) in an underloaded situation. D^* executes to completion all the tasks with deadlines in underloaded intervals[14]. D^* also guarantees that all the tasks with a deadline in an overloaded interval will achieve a cumulative value of at least *one-fifth* of the *length* of the overloaded interval. However, D^* is not competitive (i.e., it has infinite competitive multiplier).

Baruah et. al. [4,3] demonstrated, using an adversary argument that, in the uniform value density setting, there can be *no* on-line scheduling algorithm with a competitive multiplier smaller than 4.

[14]The definition of an underload interval appears there [4,16].

Koren and Shasha described [19] an algorithm called *DD-star* (DD*), that has a competitive multiplier of 4 in the uniform value density case and offers 100% of the possible value in the underloaded case. This showed that the bound of 4 is tight in the uniform value density case. Wang and Mao [38] independently reported a similar guarantee.

On the lower bound side, Baruah et. al. [3,4] showed for environments with an importance ratio k, a bound of $(1+\sqrt{k})^2$ on the best possible competitive multiplier of an on-line scheduler. This result and some pragmatic considerations reveal the following limitations of the competitive scheduling algorithms described above:

1. The algorithms all assume a uniform value density, yet some short tasks may be more important than some longer tasks.
2. The algorithms all assume that there is no value in finishing a task after its deadline. But a slightly late task may be useful in many applications.
3. The algorithms all assume that the computation time is known upon release. However, a task program that is not straight-line may take different times during different executions.

D^{over}, the on-line scheduling algorithm presented in chapter 3 (and its extensions in appendices B and C) addresses all these limitations.

Multiprocessor Environments

Locke [29, pp. 124-134] presented a simple heuristic extending his *best effort* scheduling for multiprocessor environments. Ramamritham and Stankovic [37] 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.

Zhou et. al. presented an on-line algorithm [39][15] 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).

[15] And additional references within.

Wang and Mao [33] 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.

1.5 Main Results

This dissertation presents results for uniprocessor systems and for systems with two or more processors.

1.5.1 Uniprocessor Environments

We present an on-line scheduling algorithm called D^{over} that has an optimal competitive multiplier of $(1+\sqrt{k})^2$ for environments with importance ratio k. Hence we show that the bound of Baruah et. al. [34] is tight for all k. D^{over} also gives 100% of the value obtainable by a clairvoyant scheduler when the system is underloaded.

D^{over} can be implemented using balanced search trees and runs at an *amortized* cost of $O(\log n)$ operations per task where n bounds the number of tasks in the system at any instant.

We also investigated two important extensions to the task model presented earlier.

- Gradual Descent:

  We relax the *firm deadline* assumption. Tasks that complete after their deadline can still have a positive value though less than their initial value. As in Locke [29] the task's value is given by a *value function* which depends on its completion time.

  We show that under a variety of value functions an appropriate version of D^{over} has a competitive multiplier of $(1+\sqrt{k})^2$ for environments with importance ratio k.

- Situations in which the exact computation time of a task is not known:

  Suppose the on-line scheduling algorithm does not know the exact computation time of a task upon its release. However for every task T an upper bound on its possible computation time denoted by c_{max} is given and the actual computation time of T denoted by c satisfies:

$$(1-\epsilon)\cdot c_{max} \leq c \leq c_{max}$$

for some $0 \leq \epsilon < 1$.

We show that in that case D^{over} has a competitive multiplier of:

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

We also show that in this setting no on-line scheduler can guarantee 100% of the value obtainable by a clairvoyant algorithm for underloaded systems.

1.5.2 Multiprocessor Environments

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 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 small multiplicative factor from the lower bound for the same system.

- Scheduling Algorithms for Two-Processor Systems

  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[16]. For the "No-Migration" model a variant of this algorithm called the *Safe-Risky-(fixed)* achieves a competitive multiplier of 3.

1.6 Dissertation Overview

In chapter 2 we present some notation as well as formal definitions and assumptions of our model. In chapter 3 we present our uniprocessor results while chapter 4 gives the multiprocessor results. These chapters correspond to and extend the material in Koren and Shasha [17 18 20 22]. The main body of the dissertation ends with a short conclusion chapter. It includes a summary of the current state of the art in real-time on-line scheduling and a collection of open problems.

The dissertation is supplemented by four appendices. In appendix A we study the exact guarantee given by D^{over} for systems with occasional overloaded and underloaded periods. In appendices B and C we present the *gradual-descent* and *unknown-computation-time* extensions to our uniprocessor model. In appendix D we present our results for two processor environments.

[16]This was already known when tasks have no slack-time [3,38].

Chapter 2

Notation and Assumptions

We are given a collection of tasks $T_1, T_2 \cdots T_n \cdots$ denoted by Γ For each task T_i its value is denoted by v_i its release time is denoted by r_i its computation time by c_i and its deadline by d_i.

Definition 2.0.1

- UNDERLOADED AND OVERLOADED SYSTEMS: A system is *underloaded* if there exists a schedule that will meet the deadline of every task and *overloaded* otherwise.
- EXECUTABLE PERIOD: The *executable period* Δ_i of the task T_i is defined to be the following interval: $\Delta_i = [r_i, d_i]$.
  By definition T_i may be scheduled only during its executable period.

Suppose a collection of tasks is being scheduled by some scheduler S.

- COMPLETED TASK: A task (successfully) *completes* if before its deadline the scheduler S gives it an amount of execution time that is equal to its computation time.
- PREEMPTED TASK: A task is *preempted* when the processor stops executing it but then the task might be scheduled again and possibly complete at some later point.
- A READY TASK:
  A task is said to be *ready* at time t if its release time is before t its deadline is after t and it neither completed nor was abandoned before t (the current executing task if any is always a ready task).

The earliest deadline first algorithm (hereafter D) is described in figure 2.

At any given moment
schedule the ready task with the earliest deadline.

Figure 2.1: *D* The Earliest Deadline First scheduling algorithm.

We shall make the following assumption:

Assumption 2.0.2

- TASK MODEL: Tasks may enter the system at any time; their computation times and deadlines are known exactly at their time of arrival (we weaken this assumption of exact knowledge later in appendix C). Nothing is known about a task before it appears.

We do assume however that an upper bound on the possible importance ratio is known *a priori* and can be used by the on-line scheduler (this bound is denoted by k). In the uniprocessor case this assumption can be relaxed [32].

- TASKS SWITCHING TAKES NO TIME: A task can be preempted and another one scheduled for execution instantly.

Suppose that a collection of tasks Γ with importance ratio k is given.

- NORMALIZED IMPORTANCE: Without loss of generality assume that the smallest importance of a task in Γ is 1. Hence if Γ has *importance ratio* of k the highest possible value density of a task in Γ is k.

In uniprocessor environments we add the following assumption:

- NO OVERLOADED PERIODS OF INFINITE DURATION: We assume that overloaded periods of infinite duration will not occur. This is a realistic assumption since overload is normally the result of a temporary emergency or failure.

  Indeed in the uniprocessor case Baruah et. al. [3] showed that there is no competitive on-line algorithm when overloaded periods of infinite duration are possible[1]. Note that the number of tasks in Γ may be infinite as long as no infinite overload period is generated[2].

In multiprocessor environments we add the following assumption:

- IDENTICAL PROCESSORS: All processors have the same speed and all tasks can be scheduled on any of the processors.

[1]Intuitively, the adversary can generate a sequence of tasks with ever growing values. This will force any competitive scheduler to abandon the current task in favor of the next one and so on. If the competitive scheduler attempts to complete a task in favor of a new larger one, then the adversary completes the larger one. In either case, the on-line schedule will result in a small value compared with an arbitrarily large value for a clairvoyant scheduler

[2]For the definition of overloaded periods see section 3.3.

Chapter 3

Uniprocessor Environments

In this chapter we describe D^{over} an optimal competitive scheduler for uniprocessor environments.

3.1 D^{over}

In the algorithm described below there are three kinds of *events* (each causing an associated interrupt) considered:

- **Task Completion**: successful termination of a task. This event has the highest priority.
- **Task Release**: arrival of a new task. This event has low priority.
- **Latest-start-time Interrupt**: the indication that a task must immediately be scheduled in order to complete by its deadline that is the task's remaining computation time is equal to the time remaining until its deadline. This event has also low priority (the same as task release).

If several interrupts happen simultaneously they are handled according to their priorities. A task completion interrupt is handled before the task release and latest-start-time interrupt interrupts which are handled in random order. It may happen that a task completion event suppresses a lower priority interrupt e.g. the task completion handler schedules the next task if this task had just reached its *LST* then the latest-start-time interrupt is removed.

At any given moment the set of ready tasks[1] is partitioned into two disjoint sets. *privileged* tasks and *waiting* tasks. Whenever a task is preempted it becomes a *privileged* task. However whenever some task is scheduled as a result of latest-start-time interrupt <u>all</u> the ready tasks (whether preempted or never scheduled) become *waiting* tasks.

D^{over} maintains a special quantity called **availtime**. Suppose a new task is released into the system and its deadline is the earliest among all ready tasks. The value of **availtime** is the maximum computation time that can be taken by such a task without causing the current task or any of the privileged tasks to miss their deadlines.

D^{over} requires three data structures called **Q_privileged** **Q_waiting** and **Qlst**. Each entry in these data structures corresponds to a task in the system. **Q_privileged** contains exactly

[1] Excluding the currently executing task.

the privileged tasks and Q_waiting contains the *waiting* tasks. These two structures are ordered by the tasks' deadlines. In addition the third structure Qlst contains all tasks (again not including the current task) but this time they are ordered by their latest-start-times (LST).

These data structures support Insert Delete Min and Dequeue operations.

- The Min operation for Q_privileged or Q_waiting returns the entry corresponding to the task with the earliest deadline among all tasks in Q_privileged or Q_waiting. For Qlst the Min operation returns the entry corresponding to the task with the earliest LST among all tasks in the queue. The Min operation does not modify the queue.
- A Dequeue operation on Q_privileged (or Q_waiting) deletes from the queue the element returned by Min in addition it deletes this element from Qlst. Likewise a Dequeue operation on Qlst will delete the corresponding element from either Q_privileged if it is a *privileged* task or from Q_waiting if it is an *waiting* task.

An entry of Q_waiting and Qlst consists of a single task whereas an entry of Q_privileged is a 3-tuple (T Previous-time Previous-avail) where T is a task that was previously preempted at time Previous-time. Previous-avail is the value of the variable availtime at time Previous-time. All of these data structures are implemented as balanced trees (e.g. 2-3 trees).

D^{over}'s code is depicted in figures 3.1-3.4. The following is an intuitive description of the algorithm: as long as no overload is detected (i.e. there is no lst interrupt) D^{over} schedules in the same way as D. Tasks that are preempted during this phase in favor of a task with an earlier deadline become privileged tasks. The task with the earliest deadline (either a newly released task or a *waiting* task) will be scheduled provided it does not cause overload when added to the privileged tasks. This proviso is always met in situations of underload.

During overload when a *waiting* task reaches its LST it will cause a latest-start-time interrupt. This means that some task must be abandoned: either the task that reached its LST or some of the privileged tasks. The latest-start-time interrupt routine compares the value of that task against the *sum* of the values of all the privileged tasks. If its value is greater than $(1 + \sqrt{k})$ times that sum then this task will execute on the processor while all the privileged tasks will lose their privileged status to become *waiting*

In the following code $Now()$ is a function that returns the current time. $Schedule(T)$ is a function that gives the processor to task T. $Laxity(T)$ is a function that returns the amount of time the task has left until its deadline less its remaining computation time. That is $laxity(T) = deadline(T) - (now() + remaining\_computation\_time(T))$. ϕ denotes the empty set.
This code includes lines manipulating *intervals*. The notion of an interval is needed for purpose of analysis only so these lines are commented.

```
recentval    := 0     (* This will be the running value of privileged tasks. *)
availtime    := $\infty$
                      (* Availtime will be the maximum computation time that can
                      be taken by a new task without causing the current task or
                      the privileged tasks to miss their deadlines. *)
Qlst         := $\phi$     (* All ready tasks, ordered according to their latest start time.
                      *)
Q_privileged:= $\phi$      (* The privileged tasks ordered by deadline order *)
Q_waiting    := $\phi$     (* All the waiting tasks ordered by their deadlines. *)
idle         := true  (* In the beginning the processor is idle *)


loop
 task completion :
    if (both Q_privileged and Q_waiting are not empty) then
        (* Both queues are not empty and contain together all the ready tasks.
        The ready task with the earliest deadline will be scheduled unless it is a
        task of Q_waiting and it cannot be scheduled with all the privileged tasks.
        The first element in each queue is probed by the Min operation. *)

        $(T_{Q\_privileged}, t_{prev}, avail_{prev})$ := Min(Q_privileged);

        (* Next, compute the current value of availtime. This is the correct value
        because $T_{Q\_privileged}$ is the task last inserted of those tasks currently in
        Q_privileged. The available computation time has decreased by the time
        elapsed since this element was inserted to the queue. *)

        availtime := $avail_{prev} - (now() - t_{prev})$;
        (* Probe the first element of Q_waiting and check which of the two tasks
        should be scheduled. *)
```

Figure 3.1: D^{over}- A Competitive optimal on-line scheduling algorithm.

```
            $T_{Q\_waiting}$ := Min(Q_waiting);
            if $d_{Q\_waiting} < d_{Q\_privileged}$ and
            availtime$\geq$ remaining_computation_time($T_{\mathsf{Q\_waiting}}$) then
              (* Schedule the task from Q_waiting. *)
              Dequeue(Q_waiting);
              availtime:= availtime $-$ remaining_computation_time($T_{Q\_waiting}$);
              availtime:= min(availtime laxity($T_{Q\_waiting}$));
              Schedule $T_{Q\_waiting}$;
            else
              (* Schedule the task from Q_privileged. *)
              Dequeue(Q_privileged);
              recentval := recentval $-$ value($T_{Q\_privileged}$);
              Schedule $T_{Q\_privileged}$;
            endif (*which task to schedule. *)
        else if (Q_waiting is not empty) then
            (* Q_privileged is empty. The current interval is closed here, $t_{close}$ =
            now(). The first task in Q_waiting is scheduled *)

            $T_{current}$ := Dequeue(Q_waiting);
            availtime:= laxity($T_{current}$);
            (* A new interval is created with $t_{begin}$ = now().*)

            Schedule $T_{current}$;
        else if (Q_privileged is not empty)
            (* Q_waiting is empty. The first task in Q_privileged is scheduled *)

            ($T_{current}, t_{prev}, avail_{prev}$) := Dequeue(Q_privileged);
            recentval := recentval $-$ value($T_{current}$);
            availtime := $avail_{prev} - (now() - t_{prev})$;
            Schedule $T_{current}$;
        else
            (* Both queues are empty. The interval is closed here, $t_{close}$ = now(). *)

            idle := true;
            availtime:= $\infty$;
        endif
    end (*task completion *)
```

Figure 3.2: D^{over} (cont.)

```
task release : (* $T_{arrival}$ is released. *)
    if (idle ) then
        Schedule $T_{arrival}$;
        availtime:= laxity($T_{arrival}$);
        idle := false;
        (* A new interval is created with $t_{begin} = now()$.*)
    else (*$T_{current}$ is executing *)
        if $d_{arrival} < d_{current}$ and
        availtime$\geq$ computation_time($T_{arrival}$) then
          (* No overload is detected, so the running task is preempted. *)
          Insert $T_{current}$ into Qlst;
          Insert ($T_{current}$, $now()$, availtime) into Q_privileged;
          (* The inserted task will be, by construction, the task with the earliest
          deadline in Q_privileged*)
          availtime:= availtime − remaining_computation_time($T_{arrival}$);
          availtime:= min(availtime  laxity($T_{arrival}$))
          recentval := recentval + value($T_{current}$);
          Schedule $T_{arrival}$;
        else (* $T_{arrival}$ has later deadline or availtime is not big enough.*)
          (* $T_{arrival}$ is to wait in Q_waiting *)
          Insert $T_{arrival}$ into Qlst and Q_waiting;
        endif
    endif (*idle *)
end (*release *)

latest-start-time interrupt :
    (* The processor is not idle and the current time is the latest start time
    of the first task in Qlst. *)

    $T_{next}$ = Dequeue(Qlst);
    if ($v_{next} > (1 + \sqrt{k})$ ($v_{current}$ + recentval)) then
        (*$v_{next}$ is big enough; it is scheduled. *)
        Insert $T_{current}$ into Qlst and Q_waiting;
        Remove all privileged tasks from
        Q_privileged and insert them into Qlst and Q_waiting;
        (* Q_privileged = $\phi$ *)
        recentval := 0;
        availtime:= 0
```

Figure 3.3: D^{over} (cont.)

```
            Schedule $T_{next}$;
        else (*$v_{next}$ is not big enough; it is abandoned. *)
            Abandon $T_{next}$;
        endif
    end (*$LST$ *)
    end{loop }
```

Figure 3.4: D^{over} (cont.)

tasks (these tasks might later be successfully rescheduled). Otherwise the task reaching its LST is abandoned. A task T that was scheduled by a latest-start-time interrupt can be abandoned in favor of another task T' that reaches its LST but only if T' has at least $(1+\sqrt{k})$ times more value than T. D^{over} returns to schedule according to D when some task scheduled by its latest-start-time interrupt completes.

The reader may be curious to know why D^{over} compares *values* rather then *value densities* and why the values are compared using the magic factor of $(1+\sqrt{k})$? The lower bound proof [3 4 shows why value density cannot be a good criterion for choosing which task to abandon[2]. The factor of $(1+\sqrt{k})$ happened to be the one that gave the desired result since it yields the correct ratio between the minimal value gained by D^{over} and the maximal value that might have been missed.

3.2 Analysis of D^{over}

In order to facilitate the analysis of D^{over} it is convenient to introduce the notation of intervals.

Definition 3.2.1

- Intervals: The intervals are created (opened) and closed according to the scheduling decisions of D^{over} and this process is depicted in the code of D^{over} in section 3.1.

[2]In that proof going after high value density tasks (the short *teasers*) will give the on-line scheduler minuscule value compared to the clairvoyant scheduler that will schedule a low value density task that has long computation time and hence big value.

When an interval is created (comments 37 and 59 of D^{over}) it is considered *open* meaning that it may be extended it is closed when a task completes whileQ_privileged is empty (comments 33 and 48). A new interval would be opened when the next task is scheduled. Initially there is no open interval. Hence the first interval is opened when the processor first becomes non-idle.
The interval consists of the time between the point it was opened and the point it was closed. We will denote by $I = [t_{begin}, t_{close}]$ an interval I that was opened at t_{begin} and closed at t_{close}.
Note: Two intervals may overlap only at their end points.

- BUSY: Suppose D^{over} schedules a collection of tasks. Let $BUSY$ denotes the time during which the processor is not idle during the execution of these tasks. For simplicity the length of$BUSY$ will also be denoted by $BUSY$.
Note that $BUSY$ equals the union of all intervals created by D^{over}.

Suppose that a collection of tasks Γ with importance ratio k is given. and D^{over} schedules this collection. When a task is scheduled it can have zero or positive slack time. A task may be preempted and then re-scheduled several times. We will be mainly concerned with the last time a task was scheduled. For the purposes of analyzing D^{over} we will partition the collection of tasks according to the question of whether the task had completed exactly at its deadline or before its deadline or failed.

- Let F (for fail) denote the set of tasks that were abandoned.
- Let S^p (for successful with *p*ositive time before the deadline) denote the set of tasks that completed successfully and that ended some positive time before their deadlines.
- Let S^0 (for successful with *0* time before the deadline) denote the set of tasks that completed successfully but ended exactly at their deadlines.

Call a task *order-scheduled* if it was scheduled by the task completion or task release handlers. Call a task *lst-scheduled* if it was scheduled as a result of a latest-start-time interrupt. (As mentioned above a latest-start-time interrupt is raised on a waiting task when it reaches its latest start time (LST) i.e. the last time when it can start executing and still complete by its deadline).

The first task in each interval is order-scheduled. The subsequent tasks (if any) in this interval may be order-scheduled or lst-scheduled. Proposition 3.2.1 shows that once a task is lst-scheduled all subsequent tasks of this interval must be lst-scheduled. During an interval several order-scheduled tasks may complete but only one lst-scheduled task can complete (this task will also be the last task that executes in the interval).

Proposition 3.2.1 *According to the scheduling of* D^{over} *once a task is lst-scheduled, then all subsequent tasks, in the current interval, are lst-scheduled.*

PROOF.
Suppose the current task $T_{current}$ is lst-scheduled and a task $T_{arrival}$ is released. $T_{arrival}$ will not be scheduled by the task release handler because when the current task is lst-scheduled **availtime** equals zero (see statement 86 of D^{over}) hence no task can be scheduled by the task release handler (see statement 61 of D^{over}). $\square$

Let **recentval**(t) denote[3] **recentval** at time t and **achievedvalue**(t) denote[4] the value achieved during the current interval before t. For an interval I **achievedvalue**(I) is the total value obtained during I.
We partition the value obtained during I in two different ways:

- **ordervalue** vs. **lstvalue**: **ordervalue**(I) is the total value obtained by order-scheduled tasks that completed during I. The value obtained by lst-scheduled tasks is denoted by **lstvalue**(I) (there is at most one such task in any interval I).
- **zerolaxval** vs. **poslaxval**: **zerolaxval**(I) denotes the total value obtained by tasks that completed at their deadlines during I (tasks in S^0). The value obtained by tasks that completed before their deadlines is denoted by **poslaxval**(I).

Hence for every interval

$$\mathsf{achievedvalue}(I) = \mathsf{ordervalue}(I) + \mathsf{lstvalue}(I) = \mathsf{zerolaxval}(I) + \mathsf{poslaxval}(I)$$

When the index (I) is omitted we refer to the entire execution. For example **ordervalue** denotes the total value obtained by order-scheduled tasks summing over all intervals.

[3]In the following only recentval is a variable explicitly manipulated by D^{over}. All the others: **zerolaxval**, **poslaxval**, **ordervalue** and **lstvalue** are introduced here to facilitate the analysis. This is why they do not reference algorithm statements.

[4]See statements 1,29,44,68 and 85.

Example 3.2.2 Before the detailed analysis let us first study an example of D^{over}'s scheduling. Consider the overloaded collection of six tasks depicted in table 3.1. For notational convenience we will denote the tasks by their deadlines hence for exampleT_{20} is a task with deadline at time 20. In this example we assume uniform value density (i.e. $k = 1$). D^{over} schedules the above collection as follows: In the beginning availtime is ∞

Task	*Release-Time*	*Computation-Time*	*Deadline*	Δ_i
T_{20}	0	6	20	$[0, 20]$
T_{34}	1	26	34	$[1, 34]$
T_{24}	1	20	24	$[1, 24]$
T_{18}	2	5	18	$[2, 18]$
T_{17}	3	2	17	$[3, 17]$
T_5	4	1	5	$[4, 5]$

Table 3.1: The tasks for example 3.2.2.

and Q_privileged is empty. First D^{over} schedules T_{20} to run at time 0. Availtime is set to 14 since this is T_{20}'s laxity.

At time 1 T_{34} is released into the system. Since T_{34}'s deadline is not earlier than the current task's (T_{20}) T_{34} is inserted into Q_waiting (and Qlst with LST equal 8). Also at time 1 T_{24} is released. Again since its deadline is after 20 this task is inserted into Q_waiting and Qlst with LST equals 4.

At time 2 T_{18} is released. This time the current task is preempted. T_{20} is inserted into Q_privileged and Qlst with LST equals 16. Availtime is decremented by the computation time of T_{18}. Its new value is 9. The value of recentval is set to the value of T_{20} (6).

T_{18} executes for one time unit until time 3 whenT_{17} is released. T_{17} is scheduled since its computation time (2) is smaller then availtime (9). Availtime is decremented by the computation time of T_{17}. Its new value is 7. The value the value of T_{18} (5) is added to recentval which becomes 11.

At time 4 two events occur: T_{24} reaches its LST and T_5 is released. These events can be handled in any order and we choose to handle the latest-start-time interrupt first. T_{24} reaches its LST but its value is smaller than twice ($1 + \sqrt{k} = 2$) the value of the current task plus recentval ($2 + 11$). Hence T_{24} is abandoned. T_5 is released and its deadline is earlier than the current task's (T_{17}). T_5 is scheduled since its computation time is smaller then availtime($1 < 7$). T_5 has laxity of zero which is smaller than the current availtime minus the computation time of T_5 (6). Hence availtime is now set to 0 and recentval becomes $11 + 2 = 13$.

At time 5 T_5 completes and since T_{17} is the task with the earliest deadline it is scheduled. Availtime is now 6 because this the value of availtime when T_{17} was executing (7) minus the time elapsed since it was inserted to Q_privileged (1). The value of T_{17} is subtracted from recentval which becomes $13 - 2 = 11$.

The remaining computation time of T_{17} is one unit hence at time 6 it completes. The next task in Q_privileged is T_{18} which has a remaining computation time of 4 units. Availtime is set to 6 which is value of availtime when T_{18} was executing (9) minus the time elapsed since it was inserted to Q_privileged $((6 - 3) = 3)$ (the value of T_{18} is subtracted from recentval which becomes $11 - 5 = 6$). However T_{18} will execute only until 8 when T_{34} reaches its LST. The value of T_{34} is big enough to preempt the current task. All tasks from Q_privileged are moved to Q_waiting and availtime as well as recentval are reset to zero.

The LST of T_{18} is 16 and of T_{20} (the only other task in Qlst) is 15. These tasks will generate latest-start-time interrupt in these respective times both will be abandoned.

At time 34 T_{34} completes its execution and D^{over} finished scheduling this history. Table 3.2 summarizes the scheduling decisions of D^{over}. In this example $S^0 = [T_5, T_{34}], S^p = [T_{17}]$ and $F = [T_{18}, T_{20}, T_{24}]$. Only three tasks complete their execution and the total value obtained by D^{over} is 29. A clairvoyant scheduler can achieve a value of 34 by scheduling T_{17}, T_{20} and T_{34}. Also notice that the system is already overloaded at time 1 but the first time an overload is "detected" by D^{over} is at time 4.

3.2.1 Proof Strategy

Our goal is to show that D^{over} has a competitive multiplier of $(1 + \sqrt{k})^2$ for every collection of tasks with importance ratio of k. We will start by proving some lemmas about the behavior of D^{over}. Then we will try to estimate the best possible behavior of a clairvoyant algorithm by comparison to D^{over}. Our basic strategy is to bound from below what D^{over} achieves during each interval. This will lead to a global lower bound over the entire execution. Then we bound from above what a clairvoyant scheduler can achieve during the entire execution.

t	re-lea-sed	pre-empted (LST)	com-ple-ted	sch-edu-led	availtime	Q_priv-ileged	rec-ent-val	Q_wait ing	comment
0					∞	[]	0	[]	
0	T_{20}			T_{20}	$laxity(T_{20})$	[]	0	[]	new interval
1	T_{34}				14	[]	0	$[T_{34}]$	T_{34}'s LST is 8
1	T_{24}				14	[]	0	$[T_{24}, T_{34}]$	T_{24}'s LST is 4
2	T_{18}	T_{20} (16)		T_{18}	$min(14-5, 13)$	$[T_{20}]$	6	$[T_{24}, T_{34}]$	
3	T_{17}	T_{18} (14)		T_{17}	$min(9-2, 12)$	$[T_{18}, T_{20}]$	$5+6$	$[T_{24}, T_{34}]$	
4					$min(9-2, 12)$	$[T_{18}, T_{20}]$	11	$[T_{34}]$	T_{24}'s LST, it is abandoned
4	T_5	T_{17} (16)		T_5	$min(7-1, 0)$	$[T_{17}, T_{18}, T_{20}]$	$2+11$	$[T_{34}]$	T_5 has no no laxity
5			T_5	T_{17}	$7-(5-4)=6$	$[T_{18}, T_{20}]$	$5+6$	$[T_{34}]$	
6			T_{17}	T_{18}	$9-(6-3)=6$	$[T_{20}]$	6	$[T_{34}]$	
8		T_{18} (15)		T_{34}	0	[]	0	$[T_{18}, T_{20}]$	T_{34}'s LST
15					0	[]	0	$[T_{18}]$	T_{20}'s LST
16					0	[]	0	[]	T_{18}'s LST
34			T_{34}		0	[]		[]	interval closed

Table 3.2: D^{over}'s scheduling for example 3.2.2.

3.2.2 Some Lemmas about D^{over}'s Scheduling

In this section we present some technical lemmas about the behavior of D^{over}. These lemmas will be used in the next section when comparing D^{over}'s performance with that of a clairvoyant scheduler. These lemmas concern the relationship between the interval length and the value achieved by D^{over} in that interval (lemma 3.2.3). As well as the relationship between the computation time and value of tasks abandoned in an interval with respect to the value achieved in the interval (lemma 3.2.4 and 3.2.5). Recall that BUSY is the union of all intervals (definition 3.2.1).

Lemma 3.2.2

1. *For any task T_i in S^0, $\Delta_i = [r_i, d_i] \subseteq BUSY$*
2. *For any task T_i in F. Suppose T_i was abandoned at time t_{aban}, then $[r_i, t_{aban}] \subseteq BUSY$*

PROOF.
A processor is idle under D^{over} scheduling only if there is no *ready* task.

- A task T_i of S^0 does not complete before its deadline hence it is a *ready* task during all its executable period. This implies that there is no idle time during the executable period of T_i.

- Similarly a task of F is a ready task from its release time to the point at which it is abandoned. Therefore there is no idle time between its release point and its abandonment point.

□

Lemma 3.2.3 *For any interval $I = [t_{begin}, t_{close}]$, the length of I, $t_{close} - t_{begin}$ will satisfy*

$$\begin{aligned} t_{close} - t_{begin} &\leq \mathsf{ordervalue}(I) + (1 + \frac{1}{\sqrt{k}}) \cdot \mathsf{lstvalue}(I) \\ &= \mathsf{achievedvalue}(I) + \frac{1}{\sqrt{k}} \cdot \mathsf{lstvalue}(I) \end{aligned}$$

Recall that $\mathsf{ordervalue}(I)$ *and* $\mathsf{lstvalue}(I)$ *are the values obtained by D^{over} from the order-scheduled and the lst-scheduled tasks respectively during I.*

PROOF.

An interval $I = [t_{begin}, t_{close}]$ has the following two sub-portions the second of which may be empty:

1. $[t_{begin}, t_{first\_lst}]$
   From the beginning of I to the point in time t_{first_lst} in which the first lst-scheduled task is scheduled. During this period all tasks are order-scheduled and some may complete their execution.
   If no task is lst-scheduled in I then define t_{first_lst} to be t_{close}. In this case the second sub-portion is empty.
2. $[t_{first\_lst}, t_{close}]$
   During this period all tasks are scheduled and preempted by latest-start-time interrupt. Only the last task to be scheduled completes.

If there are no lst-scheduled tasks in I then all tasks that executed from t_{begin} to t_{close} completed successfully. The value achieved is $\mathsf{ordervalue}(I)$ and is at least as big as the duration of execution[5]. Hence the lemma is proved in this case.

Otherwise suppose that $T_1, T_2, \cdots, T_m$ $(m \geq 1)$ are the tasks that were lst-scheduled in I. Hence T_1 was scheduled at t_{first_lst} later it was preempted (and abandoned) by T_2 and so forth. Eventually T_m preempts T_{m-1} and completes at t_{close} its value v_m is $\mathsf{lstvalue}(I)$.

[5]Recall that a value density is always equal or greater than 1, by assumption 2.0.2 above.

Denote by l_i the length of the execution of T_i during the process above. T_m preempted T_{m-1} hence $v_m > (1+\sqrt{k})v_{m-1}$. Which yields[6]

$$l_{m-1} < v_{m-1} < \frac{v_m}{(1+\sqrt{k})} = \frac{\mathsf{lstvalue}(I)}{(1+\sqrt{k})}$$

Going backward along the chain of preemptions we get:

$$l_i < v_i < \frac{v_{i-1}}{(1+\sqrt{k})} < \frac{\mathsf{lstvalue}(I)}{(1+\sqrt{k})^{m-i}} \quad \textit{for all } 1 \le i \le m-1 \tag{3.1}$$

T_1 preempted the last order-scheduled task hence (see statement 80 of D^{over})

$$v_1 > (1+\sqrt{k})\{\mathsf{recentval}(t_{first\_lst}) + \textit{value}(\textit{current task at time } t_{first\_lst})\} \tag{3.2}$$

Also

$$\begin{aligned} t_{first\_lst} - t_{begin} \quad \le \quad & \mathsf{ordervalue}(I) + \mathsf{recentval}(t_{first\_lst}) + \\ & \textit{value}(\textit{current task at time } t_{first\_lst}) \end{aligned} \tag{3.3}$$

This holds because the processor is not idle between t_{begin} and t_{first_lst} (as part of $BUSY$) and the right hand side above represents the sum of the values of all the tasks that were scheduled between t_{begin} and t_{first_lst}. This sum must be greater than or equal to their period of execution by the normalized importance assumption (assumption 2.0.2). Inequalities 3.1 3.2 and 3.3 imply

$$t_{first\_lst} - t_{begin} < \mathsf{ordervalue}(I) + \frac{v_1}{(1+\sqrt{k})} < \mathsf{ordervalue}(I) + \frac{\mathsf{lstvalue}(I)}{(1+\sqrt{k})^m}$$

We have produced the following bound on the distance between t_{begin} and t_{close}:

$$\begin{aligned} t_{close} - t_{begin} \quad &= \quad (t_{first\_lst} - t_{begin}) + (t_{close} - t_{first\_lst}) \\ &= \quad (t_{first\_lst} - t_{begin}) + (l_1 + l_2 + \cdots + l_m) \\ &\le \quad \mathsf{ordervalue}(I) + \\ & \qquad \mathsf{lstvalue}(I) \cdot \left(1 + \frac{1}{(1+\sqrt{k})} + \frac{1}{(1+\sqrt{k})^2} + \cdots + \frac{1}{(1+\sqrt{k})^m}\right) \\ &\le \quad \mathsf{ordervalue}(I) + \mathsf{lstvalue}(I) \cdot \sum_{i=0}^{\infty} \frac{1}{(1+\sqrt{k})^i} \\ &= \quad \mathsf{ordervalue}(I) + \mathsf{lstvalue}(I) \cdot (1 + \frac{1}{\sqrt{k}}) \\ &= \quad \mathsf{achievedvalue}(I) + \frac{1}{\sqrt{k}} \cdot \mathsf{lstvalue}(I) \end{aligned}$$

[6]Note that always $l_i \le v_i$. However, for a task that was abandoned a strict inequality $l_i < v_i$ holds.

The last equality follows from the fact that $\mathsf{achievedvalue}(I) = \mathsf{ordervalue}(I) + \mathsf{lstvalue}(I)$ by definition. □

Lemma 3.2.4 *Suppose T_i was abandoned during the interval I. Then*

$$v_i \leq (1+\sqrt{k}) \cdot \mathsf{achievedvalue}(I)$$

Recall that $\mathsf{achievedvalue}(I)$ *is the total value obtained during I.*

PROOF.

Let $I = [t_{begin}, t_{close}]$ be an interval. Define the *Prospective Value* map of I PV_I as follows:

$$PV_I(t) = \mathsf{ordervalue}(t) + \mathsf{recentval}(t) + value(current\ task\ at\ time\ t)$$
$$where\ t_{begin} \leq t \leq t_{close}$$

Claim For every interval $I = [t_{begin}, t_{close}]$

1. PV_I is monotone non-decreasing.
2. PV_I reaches at the end of the interval the total value obtained in I i.e

$$PV_I(t_{close}) = \mathsf{achievedvalue}(I)$$

Note: PV is not a function because it might have several values for one time instance since D^{over} can make several scheduling decisions at one time instance (assumption 2.0.2). However as a map with the ordered sequence of scheduling decisions as its domain PV_I is a function.

Proof of claim.

There are two cases. The first applies when there are no lst-scheduled tasks in I the other applies when such tasks exist.

Case 1: Suppose that there are no lst-scheduled tasks in I. Then every task that was scheduled does complete. Let $S(t)$ be the set of tasks that were scheduled (not necessary completed) up to t. One can verify by induction that

$$PV_I(t) = \sum_{T_i \in S(t)} v_i$$

The reason is that no scheduled task is abandoned hence at each moment a task is either the current task or in $\mathsf{Q\_privileged}$ or had completed. At the closing of I all tasks have completed. Hence

$$PV_I(t_{close}) = \sum_{T_i \in S(t_{close})} v_i = \mathsf{achievedvalue}(I)$$

PV_I is monotone (when there are no lst-scheduled tasks) because $S(t)$ is a monotone increasing set of tasks.

Case 2: Suppose there were lst-scheduled tasks. Assume that the first lst-scheduled task T_1 was scheduled at time t_{first_lst}. Let t be a time instance just before the scheduling of T_1 then by definition:

$$PV_I(t) = \mathsf{ordervalue}(t) + \mathsf{recentval}(t) + value(current\ task\ at\ time\ t)$$

T_1 is scheduled only if

$$v_1 > (1+\sqrt{k}) \cdot (\mathsf{recentval}(t) + value(current\ task\ at\ time\ t))$$

When T_1 is scheduled **recentval** is set to zero hence we can conclude that

$$\begin{aligned} PV_I(t_{first\_lst}) &= \mathsf{ordervalue}(t_{first\_lst}) + \mathsf{recentval}(t_{first\_lst}) + value(T_1) \\ &= \mathsf{ordervalue}(t) + 0 + value(T_1) \\ &> \mathsf{ordervalue}(t) + (\mathsf{recentval}(t) + value(current\ task\ at\ time\ t)) \\ &= PV_I(t) \end{aligned}$$

Thus PV_I is monotone from t_{begin} to t_{first_lst} (as in the case when there are no lst-scheduled tasks). It is left to show that PV_I continues to be monotone. After t_{first_lst} PV_I equals to

$$\mathsf{ordervalue}(I) + value(current\ task\ at\ time\ t)$$

because **recentval** remains equal to zero. This is a monotone increasing value since $\mathsf{ordervalue}(I)$ is fixed and a task T will preempt the current task only if it has a larger value than the current task's value. In particular if T_i is the last task to be scheduled in I then

$$\begin{aligned} PV_I &= \mathsf{ordervalue}(I) + v_i \\ &= \mathsf{ordervalue}(I) + \mathsf{lstvalue}(I) = \mathsf{achievedvalue}(I) \end{aligned}$$

So the claim is proved.

End of proof of claim

We return to the proof of lemma 3.2.4. There is only one way a task T_i can be abandoned at time t:

- T_i reaches its LST at t. A latest-start-time interrupt is generated. However T_i has insufficient value to preempt the task executing at time t.

Hence if T_i was abandoned then

$$\begin{aligned} v_i &< (1+\sqrt{k})\cdot\{\mathsf{recentval}(t) + value(current\ \ task\ at\ time\ t)\} \\ &\le (1+\sqrt{k})\cdot PV_I(t) && \text{, by definition of } PV \\ &\le (1+\sqrt{k})\cdot \mathsf{achievedvalue}(I) && \text{, by the claim} \end{aligned}$$

□

Lemma 3.2.5 *Suppose T_i was abandoned at time t in $I = [t_{begin}, t_{close}]$. Then,*

$$c_i \ge d_i - t_{close}$$

PROOF.
A task T_i can be abandoned at time$t$ only when:

- It reaches its LST at t. A latest-start-time interrupt is generated. However the current task is not preempted.

T_i reached its LST hence its remaining computation time is $d_i - t$. Also $t \le t_{close}$ by assumption. Hence the (initial) computation time of T_i is at least $d_i - t_{close}$. □

3.2.3 How Well Can a Clairvoyant Scheduler Do?

Given a collection of tasks Γ our goal is to bound the maximum value that a clairvoyant algorithm can obtain from scheduling Γ. We do it by observing the way D^{over} schedules Γ. From D^{over}'s scheduling we get the partitioning of the tasks to S^0, S^p and F we also take notice of the time periods in which the processor was not idle in this scheduling. As defined earlier the union of these periods is called$BUSY$.

In order to bound the value that can be achieved from scheduling Γ we will offer the clairvoyant algorithm two gifts that can only improve the value it can obtain. We will show an upper bound on the value the clairvoyant algorithm can get with these gifts hence bounding the value it can achieve from the original collection.

- As a first gift we will give the clairvoyant algorithm the sum of the values of all tasks in S^p at no cost to it (i.e. it will devote no time to these tasks). Then we will see what the clairvoyant algorithm can achieve on $F \cup S^0$.
- As a second gift suppose that in addition to the value achieved from scheduling the tasks $F \cup S^0$ the clairvoyant scheduler can get an additional value called *granted value*. The amount of granted value depends on the schedule chosen by the the clairvoyant scheduler: A value density of k will be granted for every period of $BUSY$ that is not used for executing a task.

The clairvoyant scheduler must consider that scheduling a task might reduce the granted value (since time in $BUSY$ is used). Of course when this reduction is bigger than the value of a task then the task should not be scheduled. Suppose the clairvoyant algorithm had chosen a scheduling for $F \cup S^0$. We can assume that no task was scheduled entirely during $BUSY$ because the granted value lost would be greater or equal to the value gained from scheduling the task. We will show that tasks of S^0 can execute only during $BUSY$ hence this leaves only tasks of F that were scheduled partially[7] outside $BUSY$. Executing T results in a gain of $value(T)$ but entails a loss of the granted value for the time that T executed in $BUSY$.

The clairvoyant scheduler has now two options. It can schedule no task during the entire $BUSY$ period and get only (the whole) granted value or it can use some of $BUSY$ in order to schedule some of F tasks. We will show that the maximal possible gain from choosing the second option over the first is bounded by $(1+\sqrt{k}) \cdot \mathsf{achievedvalue}$. Putting this altogether will give the desired result (theorem 3.2.12).

Example 3.2.3 To see the possibilities opened to the clairvoyant algorithm by introducing the granted value consider the following example: The length of $BUSY$ is 5 and the importance ratio k is 4. F contains only one task T with computation length 3 and value density 2. Without scheduling T the value obtained by the clairvoyant algorithm

[7]When the computation time of a task is known precisely when it is released, a task $T \in F$ cannot be scheduled completely outside $BUSY$ (see lemma 3.2.2). However, if the computation time of a task is not exactly known (appendix C), then a failed task T may be scheduled completely outside $BUSY$.

only from the granted value is $5 \times 4 = 20$. If T could have been scheduled without using any of $BUSY$ time then its value will be *added* to give $20 + 2 \times 3 = 26$. However if the clairvoyant algorithm must use 2 units of $BUSY$'s time in order to schedule T then the total value will be only $(5-2) \times 4 + 6 = 18$ hence it is better not to schedule T in this case. As a matter of fact whenever T has to use more than 1.5 units of $BUSY$'s time it should not be scheduled. $\square$

In the scenario above the clairvoyant scheduler can achieve (using the gifts) the maximal value of the sum in equation 3.4 below ranging over all possible schedulings[8] of F.

$$\begin{array}{c}\textit{value obtained from those}\\ \textit{tasks of } F \textit{ that were scheduled}\end{array} + k \cdot \begin{array}{c}\textit{length of time in } BUSY \textit{ not utilized to}\\ \textit{schedule the tasks of } F\end{array} \tag{3.4}$$

Denote by $C(\cdot)$ the value that a clairvoyant algorithm can achieve from a collection of tasks. We would like to show that $C(F \cup S^0)$ cannot be greater then this maximal value. This will then give us an upper bound on what a clairvoyant algorithm can achieve.

Lemma 3.2.6

$$C(F \cup S^0) \quad \leq \quad \max_{\substack{\textit{possible}\\ \textit{scheduling}\\ \textit{of } F}} \left\{ \begin{array}{c}\textit{value obtained by}\\ \textit{scheduling tasks of}\\ F\end{array} + k \cdot \begin{array}{c}\textit{length of time in } BUSY \textit{ not}\\ \textit{utilized by tasks of } F\end{array} \right\}$$

PROOF.

$$C(F \cup S^0) = \max \left\{ \begin{array}{c}\textit{value obtained from}\\ \textit{scheduling tasks of } F\end{array} + \begin{array}{c}\textit{value obtained from scheduling tasks of } S^0\\ \textit{during the time not used by tasks of } F\end{array} \right\}$$

[8] Suppose a *clairvoyant scheduler* has to schedule a collection of tasks A. We can assume that it schedules a task only if that task eventually completes. Hence the work of a clairvoyant scheduler is first to choose the set of tasks $A' \subseteq A$ that will be scheduled and then to work out the details of the processor allocation among the tasks of A'. We will call all possible selections of A' and processor allocation *a scheduling* of A.

S^0 tasks can be scheduled only during $BUSY$ (lemma 3.2.2) hence

$$\begin{array}{c} \textit{value obtained from} \\ \textit{scheduling tasks of } F \end{array} + \begin{array}{c} \textit{value obtained from scheduling tasks of } S^0 \\ \textit{during the time not used by tasks of } F \end{array}$$

$$\leq \begin{array}{c} \textit{value obtained by} \\ \textit{scheduling } F \end{array} + k \cdot \begin{array}{c} \textit{length of time in } BUSY \textit{ not utilized} \\ \textit{by tasks of } F \end{array}$$

The lemma is proved. □

With the above gifts the clairvoyant scheduler can maximize the sum in 3.4 above and hence obtain a value of at least $C(F \cup S^0)$.

Suppose a task $T_f \in F$ is scheduled to completion by the clairvoyant algorithm. If T_f executes entirely during $BUSY$ then the left hand factor of the sum is increased only by v_i which is smaller than or equal to $k \cdot c_i$ while the right hand factor is decreased by $k \cdot c_i$ giving zero or negative net change. Thus we assume that T_f executes (at least partially) outside $BUSY$.

Lemma 3.2.7 *Suppose T_f is abandoned (by D^{over}) at time t_{aban} and that $I = [t_{begin}, t_{close}]$ is the interval in which T_f is abandoned. Then, if T_f is to be executed (by the clairvoyant algorithm) anywhere outside $BUSY$ it must be after t_{close}.*

PROOF.
$\Delta_f = [r_f, t_{aban}] \cup [t_{aban}, d_f]$. The first portion of Δ_f is contained in $BUSY$ (lemma 3.2.2). $[t_{aban}, t_{close}] \subseteq I \subseteq BUSY$ hence ifT_i is to be scheduled anywhere outside $BUSY$ it must be after t_{close} [9]. □

Now we are ready to give an upper bound on how much additional value can the clairvoyant algorithm achieve by scheduling tasks of F compared with collecting only the granted value without scheduling any tasks. We make strong use of the fact that when a task T is abandoned during I T's value cannot be too large with respect to achievedvalue(I).

Lemma 3.2.8 *With the above gifts, the total net gain obtained by the clairvoyant algorithm from scheduling the tasks abandoned during I is not greater than*

$$(1 + \sqrt{k}) \cdot \mathsf{achievedvalue}(I)$$

[9]Note that parts of $[t_{close}, d_f]$ might be included in $BUSY$ as a new interval may be opened before d_f

PROOF.
Assume that a clairvoyant scheduler selected a scheduling for the tasks of F considering the value that can be gained from leaving $BUSY$ periods idle. We can assume that a clairvoyant algorithm executes a task only if this task eventually completes. If the clairvoyant algorithm does not schedule any of the tasks abandoned during I the lemma is proved. Hence assume that of all the tasks abandoned in $I = [t_{begin}, t_{close}]$ the clairvoyant scheduler schedules $T_1, T_2, \cdots T_m$ (in order of completion). These tasks execute for $l_1, l_2, \cdots l_m$ time *after* t_{close} (hence maybe outside $BUSY$). We know that all the l_i's are greater than zero (otherwise there is no net gain).

Lemma 3.2.4 ensures that the biggest possible value of a task to be abandoned during I is $(1+\sqrt{k}) \cdot \mathsf{achievedvalue}(I)$. If such a task has value density k its execution time is $\frac{(1+\sqrt{k}) \cdot \mathsf{achievedvalue}(I)}{k}$. Denote by L the maximal value of this execution time and the length of l_1

$$L = \max\{\frac{(1+\sqrt{k}) \cdot \mathsf{achievedvalue}(I)}{k}, l_1\} \tag{3.5}$$

Let j be the index less than to m such that

$$\sum_{i \le j} l_i \le L < \sum_{i \le j} l_i + l_{j+1}$$

If no such j exists define j to be m.

First assume that we have an equality $\sum_{i \le j} l_i = L$. The $\sum_{i \le j} l_i < L$ case is a little more complicated and will be treated later.

We will show that the net gain from scheduling tasks within a period of L after the end of the interval cannot be greater than $(1+\sqrt{k}) \cdot \mathsf{achievedvalue}(I)$.

- Suppose that in equation 3.5 the maximum is the first term. Then the total net gain from $T_1, T_2, \cdots T_j$ is not greater than

$$k \cdot \sum_{i \le j} l_i = k \cdot L = (1+\sqrt{k}) \cdot \mathsf{achievedvalue}(I) \tag{3.6}$$

- If on the other hand the second term is maximal in equation 3.5 then the value obtained by scheduling T_1 is at most $(1+\sqrt{k}) \cdot \mathsf{achievedvalue}(I)$ (lemma 3.2.4).

Now we will show that the net gain from scheduling tasks "after" L is never positive.

Every task T_i that executed at a time of at least L after the end of the interval where $j < i \leq m$ has an execution timec_i of at least $d_i - t_{close}$ (see lemma 3.2.5).

$$\begin{aligned} c_i &\geq d_i - t_{close} \\ &\geq \text{“the point at which } T_i \text{ completes (according to the clairvoyant)”} - t_{close} \\ &\geq (t_{close} + \sum_{g \leq i} l_g) - t_{close} = \sum_{g \leq i} l_g \\ &\geq l_i + \sum_{g \leq j} l_g = l_i + L \end{aligned}$$

For $i > j$ T_i was scheduled by the clairvoyant scheduler but used only l_i time after t_{close}. Hence T_i executed at least L time before t_{close} that is to say in $BUSY$ by lemma 3.2.7. The “loss” from scheduling T_i during $BUSY$ is at least $k \cdot L$. The value obtained by scheduling T_i is at most $(1 + \sqrt{k}) \cdot \mathsf{achievedvalue}(I)$ (lemma 3.2.4). Hence the net gain is less than or equal to

$$\begin{aligned} & (1 + \sqrt{k}) \cdot \mathsf{achievedvalue}(I) - k \cdot L \\ \leq\ & (1 + \sqrt{k}) \cdot \mathsf{achievedvalue}(I) - (1 + \sqrt{k}) \cdot \mathsf{achievedvalue}(I) = 0 \end{aligned}$$

We conclude that the clairvoyant algorithm is better off not scheduling any task T_i, $j < i \leq m$. Hence the lemma is proved for the case that $\sum_{i \leq j} l_i = L$.

What if L does not equal any of the partial sums? That is if$\sum_{i \leq j} l_i < L < \sum_{i \leq j+1} l_i$. We will augment the total value given to the clairvoyant by some non-negative amount. Then we will show that even with this addition the net gain achieved by the clairvoyant algorithm is bounded by $(1 + \sqrt{k}) \cdot \mathsf{achievedvalue}(I)$ hence proving the lemma.

First we will take the value density of T_j to be k. This move can only increase the overall value achieved by the clairvoyant algorithm. We will also “transfer” some execution time (and hence also value) from T_{j+1} to T_j. We will transfer exactly $L - \sum_{i \leq j} l_i$ execution time. There will be a non-negative net increase of $(k - \mathit{value\ density}(T_{j+1})) \cdot (L - \sum_{i \leq j} l_i)$ in the overall achieved value of the clairvoyant algorithm and we are back in the case of $L = \sum_{i \leq j} l_i$. The total net gain from $T_1, \cdots, T_j$ is bounded by $(1 + \sqrt{k}) \cdot \mathsf{achievedvalue}(I)$ while the net gain from all other tasks is zero or negative. $\square$

Our strategy thus far has entailed partitioning the problem into what the clairvoyant can obtain with respect to a given interval. We now compute an upper bound for what the clairvoyant algorithm can obtain over all intervals. Note that this may overestimate what the clairvoyant algorithm obtains because the time periods that the clairvoyant algorithm uses on the tasks of two neighboring intervals may overlap.

Corollary 3.2.9 *With the above gifts, the total net gain (over the entire execution) obtained by the clairvoyant algorithm from scheduling the tasks of F is not greater than*

$$(1+\sqrt{k})\cdot \mathsf{achievedvalue}$$

PROOF.
Lemma 3.2.8 measured the maximum net gain per interval. By construction each task is accounted for in exactly one interval. Therefore summing over all intervals we conclude that the total net gain during the entire execution is less than or equals to
$(1+\sqrt{k})\cdot \mathsf{achievedvalue}$. $\square$

The previous corollary bounds the value the clairvoyant algorithm could obtain beyond the granted value which equals $k\cdot BUSY$. Now we will estimate the granted value (by bounding the length of $BUSY$) to get an upper bound on $C(S^0\cup F)$.

Lemma 3.2.10

$$\begin{aligned} C(F\cup S^0) &\le k\cdot(\mathsf{achievedvalue}+\frac{1}{\sqrt{k}}\cdot\mathsf{zerolaxval})+(1+\sqrt{k})\cdot\mathsf{achievedvalue} \\ &= (k+1+\sqrt{k})\cdot\mathsf{achievedvalue}+\sqrt{k}\cdot\mathsf{zerolaxval}\end{aligned}$$

PROOF.
Lemma 3.2.6 shows that $C(S^0\cup F)$ is bounded by the maximum ranging over all possible schedulings of the tasks of F of the following sum:

$$\begin{aligned}&(\textit{value obtained by scheduling } F)+\\ &k\cdot(\textit{length of time in } BUSY \textit{ not utilized by } F \textit{ tasks}).\end{aligned}$$

Corollary 3.2.9 above shows that this sum is less than or equal to

$$(1+\sqrt{k})\cdot\mathsf{achievedvalue}+k\cdot BUSY$$

Lemma 3.2.3 summed over all intervals yields:

$$BUSY\le\mathsf{achievedvalue}+\frac{1}{\sqrt{k}}\cdot\mathsf{lstvalue}$$

$\mathsf{lstvalue}(I) \leq \mathsf{zerolaxval}(I)$ always holds because every task that is lst-scheduled must have completed at its deadline. This implies that

$$BUSY \leq \mathsf{achievedvalue} + \frac{1}{\sqrt{k}} \cdot \mathsf{zerolaxval}$$

Hence

$$\begin{aligned} C(S^0 \cup F) &\leq k \cdot (\mathsf{achievedvalue} + \frac{1}{\sqrt{k}} \cdot \mathsf{zerolaxval}) + (1+\sqrt{k}) \cdot \mathsf{achievedvalue} \\ &= (k+1+\sqrt{k}) \cdot \mathsf{achievedvalue} + \sqrt{k} \cdot \mathsf{zerolaxval} \end{aligned}$$

Which proves the lemma. □

We gave the clairvoyant algorithm the value of all tasks in S^p. We also got a bound on $C(S^0 \cup F)$. The following lemma shows that the sum of these two values bounds the value the clairvoyant can get from the entire collection.

Lemma 3.2.11

$$C(F \cup S^0 \cup S^p) \leq C(F \cup S^0) + C(S^p) = C(F \cup S^0) + \sum_{T_i \in S^p} v_i$$

PROOF.
The first inequality is due to the fact that $C(\cdot)$ is a sub-linear function. The reason is that executing tasks of S^p might interfere with tasks of $F \cup S^0$ and vice versa. Therefore the value of the union may be less than the sum of the individual sets. D^{over} schedules to completion all the tasks of S^p hence $C(S^p)$ equals the sum of the values of all these tasks. This yields the desired result. □

Given a collection of tasks Γ lemmas 3.2.10 and 3.2.11 give an upper bound on the value the clairvoyant algorithm can obtain from Γ in terms of the value obtained by D^{over} ($\mathsf{achievedvalue}$ $\mathsf{zerolaxval}$ and $\mathsf{poslaxval}$). The next theorem puts these results together.

Theorem 3.2.12 *D^{over} has a competitive multiplier of of $(1+\sqrt{k})^2$. That is, D^{over} obtains at least $\frac{1}{(1+\sqrt{k})^2}$ times the value of a clairvoyant algorithm given any task collection Γ.*

PROOF.
In the notation of the lemmas above we got from lemma 3.2.10 that

$$C(S^0 \cup F) \leq (k+1+\sqrt{k}) \cdot \mathsf{achievedvalue} + \sqrt{k} \cdot \mathsf{zerolaxval}$$

We will bound $\sqrt{k} \cdot \mathsf{zerolaxval}$ in the above equation.

$$\begin{aligned} \sqrt{k} \cdot \mathsf{achievedvalue} &= \sqrt{k} \cdot \mathsf{zerolaxval} + \sqrt{k} \cdot \mathsf{poslaxval} \geq \sqrt{k} \cdot \mathsf{zerolaxval} + \mathsf{poslaxval} \\ &\Rightarrow \sqrt{k} \cdot \mathsf{zerolaxval} \leq \sqrt{k} \cdot \mathsf{achievedvalue} - \mathsf{poslaxval} \end{aligned}$$

Hence replacing $(\sqrt{k} \cdot \mathsf{zerolaxval})$ by $(\sqrt{k} \cdot \mathsf{achievedvalue} - \mathsf{poslaxval})$ yields:

$$\begin{aligned} C(S^0 \cup F) &\leq (k + 1 + \sqrt{k}) \cdot \mathsf{achievedvalue} + \sqrt{k} \cdot \mathsf{achievedvalue} - \mathsf{poslaxval} \\ &= (1 + \sqrt{k})^2 \cdot \mathsf{achievedvalue} - \mathsf{poslaxval} \end{aligned}$$

Using lemma 3.2.11 we get:

$$\begin{aligned} C(F \cup S^0 \cup S^p) &\leq C(F \cup S^0) + C(S^p) \\ &= C(F \cup S^0) + \mathsf{poslaxval} \\ &\leq ((1 + \sqrt{k})^2 \cdot \mathsf{achievedvalue} - \mathsf{poslaxval}) + \mathsf{poslaxval} \\ &= (1 + \sqrt{k})^2 \cdot \mathsf{achievedvalue} \end{aligned}$$

□

3.2.4 The Running Complexity of $\mathbf{D}^{over}$

In the previous section we analyzed the performance of D^{over} in the sense of what value it will achieve from scheduling tasks to completion. In this section we study the cost of executing the scheduling algorithm itself.

Theorem 3.2.13 *If n bounds the number of unscheduled tasks in the system at any instant then each task incurs an $O(\log n)$* **amortized** *cost.*

PROOF.

D^{over} requires three data structures called Q_privileged Q_waiting and Qlst all of them priority queues implemented as balanced search trees e.g. 2-3 trees. They support Insert Delete Min and Dequeue operations each taking $O(\log n)$ time for a queue with n tasks. The structures share their leaf nodes which represent tasks.

D^{over} consists of a main loop with three "interrupt handlers" within it. The total number of operations is dominated by the number of times each of these handler clauses is executed and the number of data structure operations in each clause.

Suppose a history of m tasks is given. First let us estimate the number of times each handler clause can be executed. A task during its lifetime causes exactly one task release event and at most one task completion event as well as at most one latest-start-time interrupt event. Hence while scheduling m tasks the total number of events is bounded by $3m$.

Now we will bound the number of queue operations in each handler clause.

- In the handler for the task release event (statement 54) there is a constant number of queue operations. Hence this contributes a total of $O(m)$ queue operations during the entire history.
- In the handler for the task completion event (statement 9) there is a constant number of queue operations. Hence this contributes a total of $O(m)$ queue operations during the entire history.
- In the handler for latest-start-time interrupt event (see statement 76) the number of queue operations is proportional to the number of tasks in Q_privileged plus a constant (because the privileged tasks are all inserted into Q_waiting statement 83). How many tasks can be in Q_privileged throughout the history? A task can enter Q_privileged only as a result of task release event (statement 64) there are at most m such events. Hence the total number of tasks in Q_privileged is at most m which means that the total number of queue operations is $O(m)$ during the entire history.

We conclude that the total number of operations for the entire history is $O(m \log n)$ and the theorem is proved. $\square$

3.3 Underloaded Periods: Conflicting Tasks

Intuitively D^{over} is an optimal scheduler during underloaded periods because it mimics the *earliest-deadline-first* algorithm during those periods. It gives its non-trivial competitive guarantee during overloaded periods.

To make these statements precise we must define what underloaded and overloaded mean. Informally underload means a situation in which all tasks can be scheduled to completion by their deadlines. Such tasks are designated as *conflict-free*. The following algorithm (figure 3.5) gives a precise definition of conflict-free and their antithesis —

```
Function Remove_Conflicts ( $\Gamma$ ) ;

if num_of_tasks($\Gamma$) == 1 then
    return($\Gamma$);
endif;

collection_num_of_tasks :=2;
repeat
    (* Finds a collection of tasks that their combined computation time is
    longer than their combined executable periods *)

    select a collection of tasks $S = T_{i_1}, T_{i_2}, \cdots, T_{i_{collection\_num\_of\_tasks}}$ of size
    collection_num_of_tasks such that
    $r = Min_{T_i \in S}\{r_i\}$ and $d = Max_{T_i \in S}\{d_i\}$ and
    $c_{i_1} + c_{i_2} + \cdots + c_{i_{collection\_num\_of\_tasks}} > (d - r)$;
    if (such a collection is found) then
        mark all the tasks in $S$ as conflicting tasks;
        create a task $T$ with release time $r$ and deadline $d$
        and with no slack time;
        (* T is an aggregated task *)
        return( remove_conflicts( $\Gamma - S + \{T\}$));
        (* Start again with the new collection of tasks. The new collection has
        a smaller number of tasks. When the recursive calls reach the bottom
        of the recursion (that is when $\Gamma$ has no conflicting tasks) the result is
        propagated upwards (tail recursion). *)
    else
        collection_num_of_tasks := collection_num_of_tasks + 1;
    endif;
until collection_num_of_tasks > num_of_tasks($\Gamma$);
return($\Gamma$) (*In case that no conflict was found *)
```

Figure 3.5: The Remove Conflicts algorithm.

conflicting tasks[10].

D^{over} schedules to completion all conflict-free tasks (thus all tasks in an underloaded system) and also obtains at least $\frac{1}{(1+\sqrt{k})^2}$ times the value a clairvoyant algorithm can get from the conflicting tasks. The proof rests on the proof of the competitive guarantee given in this chapter and can be found in appendix 3.3.

[10]Note that the purpose of this algorithm is to define conflicting and conflict_free tasks. No scheduler needs ever to execute it.

Chapter 4

Multiprocessor Environments

4.1 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 $1, k^{\frac{1}{n}}, k^{\frac{2}{n}}, \cdots, k^{\frac{n}{n}} = k$ call them levels $0, 1, \cdots, n$. 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 $i+1$'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*[1] 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*[2]. Figure 4.1 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 next complete set is to be released (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 there will be no new task releases. 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

[1]See definition 4.1.1.

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

Figure 4.1: 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 processors. Finally each lower value density task T is so much longer than any higher value density task T' so that the value of T is much greater than the value of T'.

number of stages (see proof of theorem 4.1.3).

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 with lower value density tasks or with higher value density tasks that have very short duration (hence have little value). After the completion of these short high value density tasks 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. In the case that 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 4.1.1 EFFECTIVE VALUE DENSITY

The effective value density obtained by a scheduling algorithm $\mathcal{A}$ at 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 T such that T was 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 4.1.1 *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 bigger than the effective value density obtained by the on-line scheduler for the same period.*

PROOF.

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 $nk^{\frac{i_0}{n}}$.

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 no new tasks are released 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*[3] (recall that each level can have up to n representatives). This level 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 less) tasks hence this process must terminate producing a "promoted" schedule with a level that has the the single representative property.

Now we are ready to state and prove the version of lemma 4.1.1 for the general case (i.e. when more than one level is missing).

Lemma 4.1.2 *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 bigger than the effective value density obtained by the on-line scheduler for the same period.*

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

[3]If only one level is missing from the schedule then the single representative property is satisfied for that level.

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 $\mathcal{S}$ [4]. 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 4.1.1 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.

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

Theorem 4.1.3 *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.

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 appears in lemma 4.1.2 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 4.1.2 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

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

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 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. □

Corollary 4.1.4 *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 4.1.2 For $n = 1$ the lower bound is k which is not as good as the already known tight lower bound of $(1 + \sqrt{k})^2$ (chapter 3). For $k = 1$ a different treatment is needed.

In the next section we introduce our competitive scheduling algorithm for multiprocessor environments.

4.2 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 break the processors into *bands* (of 2 processors each) and one *central pool*. The main idea of the algorithm is 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 [7]. 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*)[5] 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 $n > \omega \geq 0$).

[5] Recall the definition of LST (section 3.1): LST = (deadline - remaining computation time). If a task is not scheduled at its LST, it will not complete.

We consider ψ intervals[6] (*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. In addition the remaining 2ν 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 (possibly after cascading).
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 *MOCA Algorithm*. In this variant the central pool is also broken into bands of two processors each[7]. 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 includes all the ready tasks that are not privileged.
- When a new task T is released it is assigned to a band as follows:

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

[7]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.

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 *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[8].

- 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[9].
  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.

     If T has a later deadline than T^* then abandon T^* and schedule T in its place; otherwise abandon$T$ [10].
- If at task completion event SP of a band becomes idle then the two processors should switch roles; the *safe-processor* becomes the *risky-processor* and vice versa[11]. This does not require task migration.

Figure 4.2 is a schematic description of the *MOCA Algorithm.* 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[12]. However an algorithm that uses the "pure" bands structure

[8] 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.

[9] Heuristics can be used to choose the processor in case more than one RPis idle.

[10] 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.

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

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

Figure 4.2: 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 4.1: The tasks for example 4.2.1.

(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 *MOCA Algorithm*. An intuitive analogy 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 4.2.1 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 from 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 4.1. Figure 4.3 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

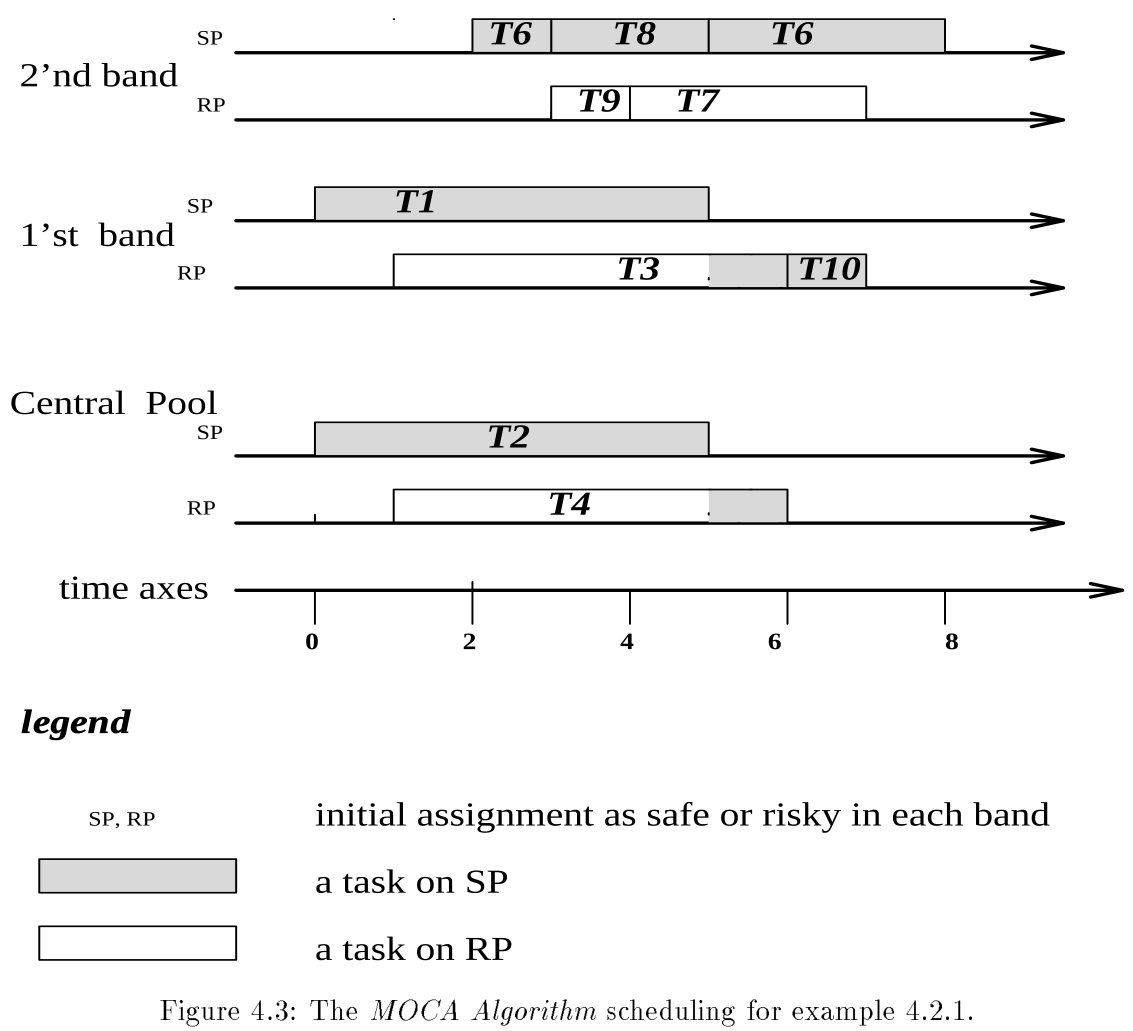

Figure 4.3: The *MOCA Algorithm* scheduling for example 4.2.1.

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

4.3 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 4.3.3. Before we start we must 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 4.3.1 *The Lost Value Lemma*

If for some constant c and every set of tasks Γ*,*

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

Then,

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

PROOF.

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

□

Definition 4.3.1

- 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. productive).
- 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 [13] .

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

[13] 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 $2nk$.

$C(F) \leq \alpha C(\Gamma)$ (for some constant α). Using the Lost Value Lemma (4.3.1) we will get a competitive multiplier of $\alpha + 1$.

First we note that only tasks of level i or higher can be scheduled on band i. Suppose a task T was abandoned by the *MOCA Algorithm* we will show that this implies that *all* the bands corresponding to the value density of T or lower were productive during the entire executable period[14] of T. If a band was productive during the executable period of T then the *MOCA Algorithm* gains a value of at least the band's value density times the length of the period. In this way we get a lower bound on the value gained by the *MOCA Algorithm* (i.e. the "enough value" mentioned above).

The following technical lemma is used in item 2 below.

Lemma 4.3.2 *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). □

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 $T(r, c, d)$[15]. 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)

[14]Recall the definition of *executable period* (2.0.1).

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

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

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 $2nk^{\frac{high(t)}{\psi}}$.

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[16]:

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

This leads to the following theorem.

Theorem 4.3.3 *For a system with $2n$ processors and maximal value density of $k > 1$ the MOCA Algorithm has a competitive multiplier of* at most

$$1 + 2n \min_{(0\le\psi\le n;\, n=\omega+\psi)} \left\{ \max_{1\le i\le\psi} \frac{k^{\frac{i}{\psi}}}{\omega + \frac{(k^{\frac{i}{\psi}}-1)}{(k^{\frac{1}{\psi}}-1)}} \right\} \tag{4.1}$$

[16]For the case $k=1$, the *uniform value density case*, see remark 4.3.2 below.

PROOF.
The discussion above demonstrated that

$$\frac{C(F)}{V(\Gamma)} \leq \max_{1\leq i\leq\psi} \frac{2nk^{\frac{i}{\psi}}}{\omega + \frac{(k^{\frac{i}{\psi}}-1)}{(k^{\frac{1}{\psi}}-1)}} = 2n \max_{1\leq i\leq\psi} \frac{k^{\frac{i}{\psi}}}{\omega + \frac{(k^{\frac{i}{\psi}}-1)}{(k^{\frac{1}{\psi}}-1)}} \tag{4.2}$$

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. □

Remark 4.3.2

- 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 (in appendix D).
- 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.

4.3.1 Setting ψ

We will estimate the complex expression for the upper bound on the competitive multiplier given by theorem 4.3.3 by setting[17] [18] $\psi = n\frac{\ln k}{\ln k+1}$. (hence $\omega = n\frac{1}{\ln k+1} = \frac{\psi}{\ln k}$) The bound in (4.2) 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 $(k^{\frac{1}{\psi}}-1)$. The maximum[19] denoted by the equation above is attained at $i = \psi$ and the upper bound (equation 4.2) is:

$$2n(k^{\frac{1}{\psi}}-1)\frac{k}{\frac{\psi}{\ln k}(k^{\frac{1}{\psi}}-1)+(k-1)} < 2n(k^{\frac{1}{\psi}}-1)\frac{k}{1+(k-1)} = 2n(k^{\frac{1}{\psi}}-1) \tag{4.3}$$

We have just proved the following lemma:

Lemma 4.3.4 *The MOCA Algorithm has a competitive multiplier of at most*

$$1 + 2n(k^{\frac{1}{\psi}}-1) \tag{4.4}$$

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 (4.4) above tends to

$$1 + 2\lim_{\psi\to\infty}\psi\frac{\ln k+1}{\ln k}(k^{\frac{1}{\psi}}-1) = 1 + 2\frac{\ln k+1}{\ln k}\ln k = 2\ln k + 3 \tag{4.5}$$

[17] ln is the natural logarithm

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

[19] Define $f_a(x)$ to be $\frac{x}{a+(x-1)}$. When $a > 1$, this function is monotone increasing with x $(x \ge 0)$. 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.

Corollary 4.3.5 *For given n and k, the ratio between the lower bound and the algorithmic guarantee is at most*

$$\frac{1 + 2n(k^{\frac{1}{\psi}} - 1)}{\frac{k}{k-1}2n(k^{\frac{1}{2n}} - 1)}, \quad \textit{where } \psi = n\frac{\ln k}{\ln k + 1} \tag{4.6}$$

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

$$\frac{2\ln k + 3}{\frac{k}{k-1}\ln k} = \frac{k-1}{k}\left(2 + \frac{3}{\ln k}\right) \tag{4.7}$$

Which gives the desired result. □

Figure 4.4 gives a graphical representation of the above result.

Remark 4.3.3 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 $n\frac{\ln k}{\ln k+1}$

4.3.2 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 *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[20]. 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 [39] to distribute the functionality of the centralized scheduler among the processors. This is an interesting and important extension to the work presented here.

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

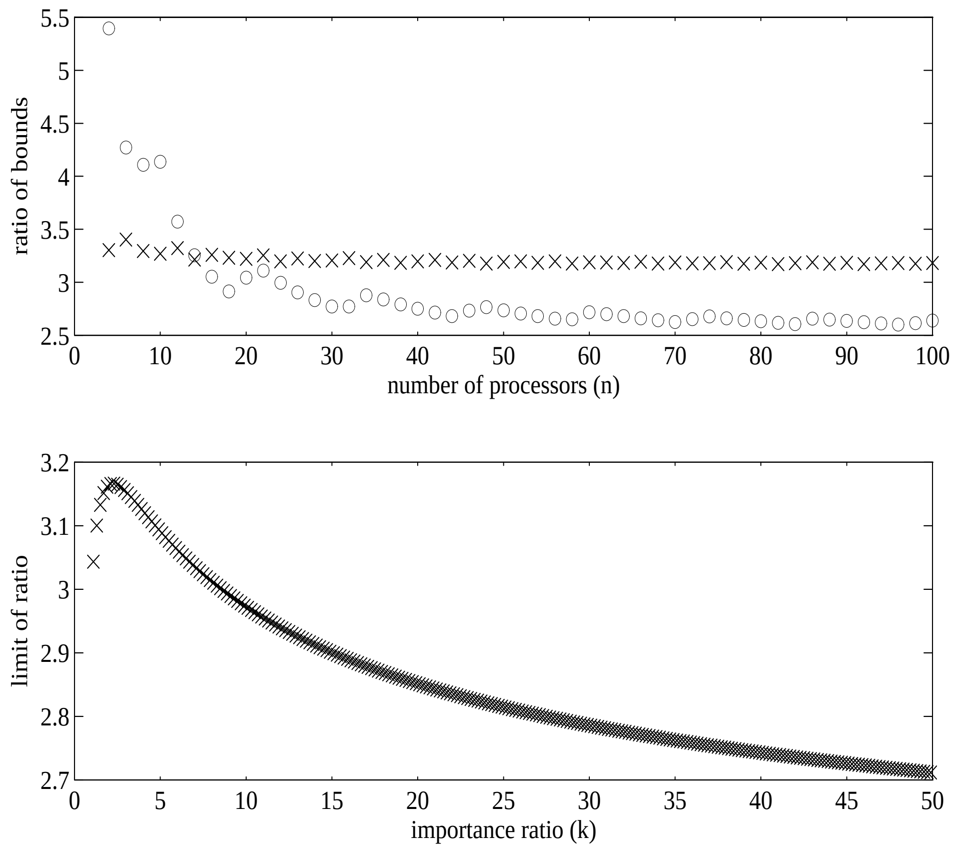

Figure 4.4: The ratio between the value guaranteed to be obtained by the algorithm and the lower bound for varying number of processors and importance ratios.
The upper graph shows the ratio (equation 4.6 above) for $k = 2$ (the 'x's) and $k = 256$ (the 'o's) for varying number of processors.
The lower graph shows the *limit* as n tends to infinity of the ratio between the algorithmic guarantee of the *MOCA Algorithm* and the lower bound (equation 4.7 above) for varying importance ratios.

4.3.3 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 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 $O(nlogM)$.

Chapter 5

Conclusions

number of processors		importance ratio	bounds		comments
			lower bound	algorithmic bound	
1	:	any $k \geq 1$	$(1+\sqrt{k})^2$ [34]	tight $\ddagger$	[1]
2	: :	1 (uniform)	2	tight	no slack[2] [33]
2	: :	1 (uniform)	2	3 $\ddagger$	with slack no migration[3]
2	: :	1 (uniform)	2	tight $\ddagger$	with slack and migration [4]
$n \geq 2$	: :	$k > 1$	$\frac{k}{(k-1)} n(k^{\frac{1}{n}} - 1)$ $\ddagger$	$1 + n(k^{\frac{1}{\psi}} - 1)$ $\ddagger$ where $\psi = \frac{n}{2} \frac{\ln k}{\ln k+1}$	[5]
$n >> 2$	:	$k > 1$	$\frac{k}{(k-1)} \ln k$ $\ddagger$	$2 \ln k + 3$ $\ddagger$	[6]

Table 5.1: State of the art of competitive real time scheduling.

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 $\ddagger$ are part of this dissertation.

This work has presented an optimal on-line scheduling algorithm for uniprocessor overloaded systems. It is optimal in the sense that it gives the best competitive factor possible relative to a clairvoyant scheduler. It also gives 100% of the value of a clairvoyant scheduler for underloaded systems. In fact the performance guarantee of D^{over} is even stronger: D^{over} schedules to completion all tasks in underloaded periods and achieves at least $\frac{1}{(1+\sqrt{k})^2}$ of the value a clairvoyant algorithm can get during overloaded periods[7]. The model accounts for different value densities and generalizes to soft deadlines.

[1]tight bound achieved by D^{over}.

[2]tasks have *no* slack time and may not migrate between processors.

[3]tasks may have slack time but may not migrate between processors.

[4]tasks may have slack time and may migrate between processors.

[5]bounds are tight within constant coefficient for many cases. The exact algorithmic guarantee can be seen in theorem 4.3.3.

[6]asymptotic behavior.

[7]The definitions of underloaded and overloaded periods and the proof for the above claim can be found in section 3.3.

For multiprocessor environments a gap remains between the guarantees achieved by the *MOCA Algorithm* and the lower bounds we have proved. The algorithmic guarantee is within a small multiplicative factor from the lower bound for large enough n and k (figure 4.4 and figure 5.1 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. [2].

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 cannot 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 *MOCA 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[8] 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 $m \geq 2$).
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} 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. The ratio between the algorithmic guarantee and the lower bound is less then 3.2 for all $k > 1$ and a large enough n.

[8]Heuristic improvements can be obtained for example scheduling tasks on the Risky Processor before tasks arrive at their latest start times.

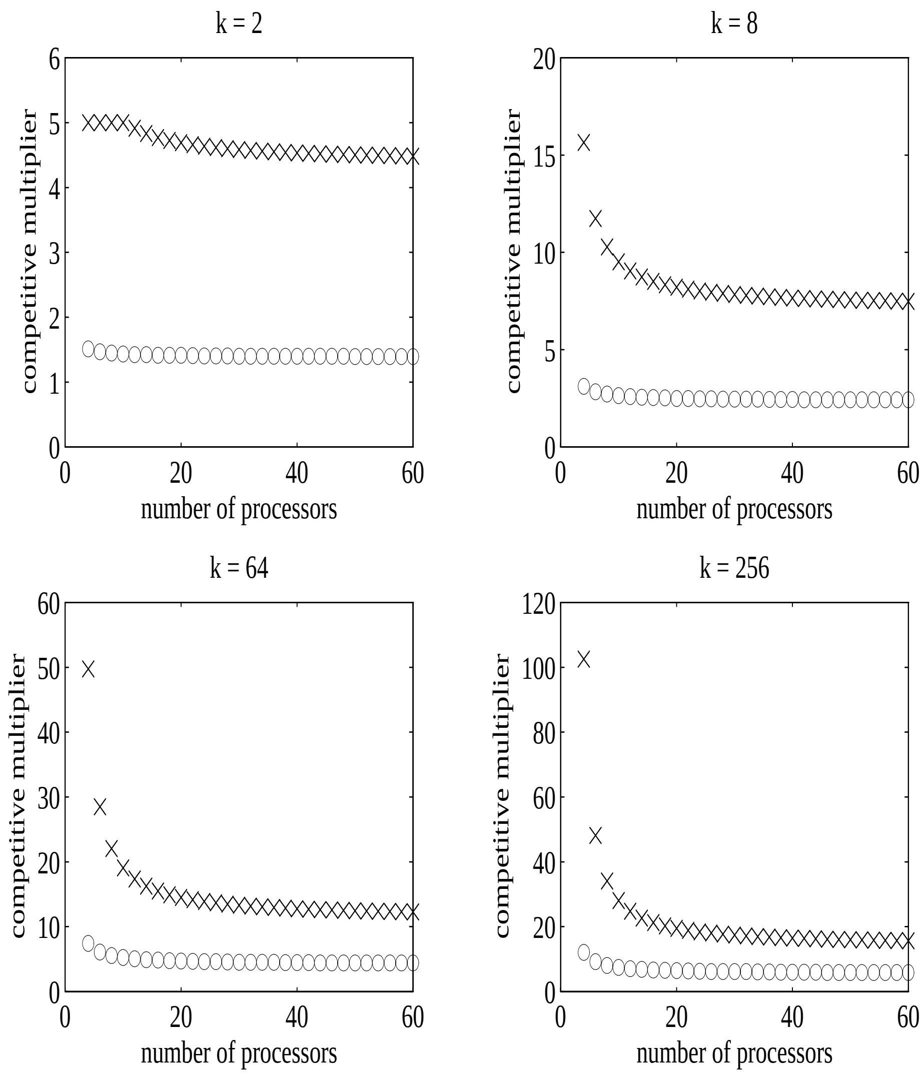

Figure 5.1: Comparison of competitive multipliers of lower bounds (o's) and algorithmic bounds (x's) for fix k's and varying number of processors.

This shows that the current algorithms are tight for large numbers of processors but that work remains to be done for small numbers of processors (see figures 4.4 and 5.1).

This work leaves many problems open. Here is a small sample.

- We assume that k is given and known in advance. It is interesting to know if this assumption can be relaxed. Recently in an unpublished result Schieber [32] devised a variant of D^{over} that gives the optimal guarantee even when k is not known in advance. Can the same be done for multiprocessor environments?
- What guarantees can be given when tasks are not independent e.g. for systems with locks or precedence constraints? What if some characteristics of the task set are known a priori (e.g. periodic tasks)?
- What guarantees can be given when not all tasks can be scheduled on all processors? What if not all processors have the same speed?
- What guarantees can be given when a penalty (i.e. a negative value) is incurred for every task that does not complete.
- In practice real-time systems have some periodic critical tasks and other less critical tasks which may be aperiodic. A typical solution (as taken in the Spring Kernel for example [37]) is to devote certain intervals to the critical tasks and to allow the less critical tasks to run during the rest of the time. D^{over} gives its usual guarantee with respect to the less critical tasks in this situation (the accounting is a little more difficult since useful time has "holes" in it corresponding to subintervals allocated to critical tasks). A much more subtle question is what is a good competitive algorithm that can take advantage of the cases when a given critical task executes in less time than is allocated for it. We suspect the competitive guarantee may be worse since the clairvoyant algorithm might then execute a task that D^{over} had prematurely abandoned.
- 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 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 *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.

Bibliography

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

[2] A. Bar-Noy Y. Mansour and B. Schieber. private communication. I.B.M T.J. Watson Research Center Yorktown Heights NY 1992.

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

[4] 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.

[5] 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.

[6] H. Chetto M. Silly and T. Bouchentouf. Dynamic scheduling of real-time tasks under precedence constraints. *The Journal of Real-Time Systems* 1:181–194 1990.

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

[8] 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.

[9] S. K. Dhall and C. L. Liu. On a real-time scheduling problem. *Operation Research* 26(1):127–140 1978.

[10] 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.

[11] 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.

[12] R. K. J. Henn. Feasible processor allocation in hard-real-time environment. *The Journal of Real-Time Systems* 1:77–93 1989.

[13] K. S. Hong and J. Y.-T. Leung. On-line scheduling of real-time tasks. *IEEE Transactions on Computers* 41(10):1321–1331 1992.

[14] K. Jeffay D. F. Stanat and C. U. Martel. On non-preemptive scheduling of periodic and sporadic tasks. In *Proceedings of the 12th Real-Time Systems Symposium* pages 129–139 San Antonio Texas Dec. 1991. IEEE.

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

[16] G. Koren B. Mishra A. Raghunathan and D. Shasha. On-line schedulers for overload real-time systems. Computer Science Department Technical Report No. 558 Courant Institute NYU New York NY May 1991.

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

[18] G. Koren and D. Shasha. MOCA : A multiprocessor on-line competitive algorithm for real-time system scheduling. *Theoretical Computer Science.* to appear in Special Issue on Dependable Parallel Computing.

[19] 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.

[20] 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.

[21] G. Koren and D. Shasha. D-over: An optimal on-line scheduling algorithm for overloaded real-time systems. Computer Science Department Technical Report No. 594 Courant Institute NYU New York NY Jan. 1992. Also technical report no. 138 INRIA Rocquencourt France Feb. 1992.

[22] G. Koren D. Shasha and S.-C. Huang. MOCA : A multiprocessor on-line competitive algorithm for real-time system scheduling. In *Proceedings of the 14th Real-Time Systems Symposium* Raleigh-Durham NC Dec. 1993. IEEE. to appear.

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

[24] J. P. Lehoczky and S. R. Ramos-Thuel. An optimal algorithm for scheduling soft-aperiodic tasks in fixed-priority preemptive systems. In *Proceedings of the 13th Real-Time Systems Symposium* pages 110–123 Phoenix Arizona Dec. 1992. IEEE.

[25] J. Y.-T. Leung. A new algorithm for scheduling periodic real-time tasks. *Algorithmica* 3:209–219 1989.

[26] J. Y.-T. Leung and M. L. Merrill. A note on preemptive scheduling of periodic. real-time tasks. *Information Processing Letters* 11(3):115–118 1980.

[27] J. Y.-T. Leung and J. Whitehead. On the complexity of fixed-priority scheduling of periodic real-time tasks. *Performance Evaluation* 2:237–250 1982.

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

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

[30] 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.

[31] 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.

[32] B. Schieber. private communication. I.B.M T.J. Watson Research Center Yorktown Heights NY 1992.

[33] L. Sha J. P. Lehoczky and R. Rajkumar. Solutions for some practical problems in prioritized preemptive scheduling. In *Proceedings of the 7th Real-Time Systems Symposium* pages 181–191 New Orleans LA Dec. 1986. IEEE.

[34] L. Sha R. Rajkumar and J. P. Lehoczky. Priority inheritance protocols. *IEEE Transactions on Computers* 39(9):1175–1185 Sept. 1990.

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

[36] B. Sprunt L. Sha and J. P. Lehoczky. Aperiodic scheduling for hard real-time system. *The Journal of Real-Time Systems* 1:27–60 1989.

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

[38] 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.

[39] 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.

Appendix A

Underloaded Periods: Conflicting Tasks

What if the collection of tasks to be scheduled is underloaded that is to say that all tasks can be scheduled to completion? We would like the on-line scheduler to be optimal in this case.

D^{over} is optimal for underloaded systems. In fact it has an even stronger performance guarantee: We devise a procedure ($Remove\_Conflicts$ see figure 3.5) to partition the tasks into two classes. The *conflict-free* tasks are those that can be scheduled to completion without preventing any other task from completing (in a sense to be made precise in the algorithm below). A task is *conflicting* otherwise.

A.1 The Remove_Conflict Procedure

We will show that D^{over} schedule to completion all conflict-free tasks (in particular all tasks in an underloaded system) and also obtains at least $\frac{1}{(1+\sqrt{k})^2}$ the value a clairvoyant algorithm can get from the conflicting tasks.

The definitions of underloaded and overload systems in section 2 are natural and widely accepted. However even when a system is overloaded it is possible that some periods are "underloaded" i.e. it is possible that some tasks will be scheduled to completion by <u>all</u> clairvoyant algorithms since they do not prevent any other task from completion. One can define the periods occupied by the aggregated tasks (definition A.1.2) as overloaded *intervals*. We prefer this definition to the one we used earlier [4 16] because it does not depend on the behavior of D [1].

Example A.1.1 To see how remove_conflicts works[2] consider the following example. Suppose we are given the collection of tasks depicted in table A.1.

In the beginning remove_conflicts is invoked with the above collection. The algorithm seeks a conflicting collection S starting with collections of size two. $S = \{T_1, T_2\}$ is such a collection since the computation time of these tasks (combined) is 8 but their combined

[1] Also in [4 16] a task is "overloaded" if and only if its deadline is in an overloaded interval. This is not reasonable because even tasks that have enough slack time to complete "safely" before the overloaded interval starts will be considered as "overloaded".

[2] Another version of this algorithm is an iterative algorithm that at each iteration selects non-deterministically a minimal set of conflicting tasks and replace them by an aggregated task. A collection is minimal in the sense that removing any one task will make the remaining tasks schedulable. Our algorithm always selects a minimal collection with the smallest possible number of tasks. Note that the purpose of this algorithm is to define conflicting and conflict_free tasks. No scheduler needs ever to execute it.

Task	*Release-Time*	*Computation-Time*	*Deadline*
T_1	0	4	6
T_2	2	4	6
T_3	0	2	8
T_4	6	2	8
T_5	0	1	9

Table A.1: Tasks for example A.1.1.

execution periods has only a length of 6. Hence these tasks are conflicting tasks and an aggregated task T_a is created with release time 0 computation time 6 and deadline 6.

The aggregated task replaces T_1 and T_2 and **remove_conflicts** is invoked with the new collection. This time there is no conflicting collection of size 2 but there is one of size 3 namely $\{T_a, T_3, T_4\}$. This is true since the combined computation time is 10 while the length of the combined execution periods is only 8. These tasks are replaced by a new aggregated task T_b which is created with release time 0 and computation time 8.

The new aggregated task replaces T_a T_3 and T_4. **remove_conflicts** is invoked again but this time there are no conflicts. The process terminates. Table A.2 summarizes the results.

Task	*Release-Time*	*Computation-Time*	*Deadline*	*Final Status*
T_1				conflicted
T_2				conflicted
T_3				conflicted
T_a	0	6	6	aggregated task
T_4				conflicted
T_b	0	8	8	aggregated task
T_5				conflict-free

Table A.2: Aggregated tasks for example A.1.1.

Definition A.1.2

- Conflicting and Conflict-Free Tasks; Aggregated Tasks: We are given a set Γ of *original* tasks. A task T is said to be *conflicting* if it was "marked" as such by the initial or any recursive call of **remove_conflicts** (statement 12). Conflicting tasks are merged into *aggregated* tasks. A task (original or aggregated) that is not conflicting is said to be a *conflict-free*. When all the tasks (original or aggregated) of a collection are conflict-free the collection is *conflict-free* and otherwise *conflicting*.

A.1.1 The Performance Guarantee of $\mathbf{D}^{over}$

In the following assume that a collection Γ is given.

Lemma A.1.1 *Γ is overloaded if and only if it is conflicting.*

PROOF.

Assume Γ is conflicting we will show that Γ is overloaded. Let T be the *first* aggregated task to be created by remove_conflicts when invoked with Γ as its input. T is an aggregate of original tasks. This means that the sum of the computation times needed for these tasks is greater than the time between their earliest release and latest deadline (see statement 10 of remove_conflicts). Hence these tasks cannot be all scheduled. We conclude that Γ is overloaded.

Assume Γ is overloaded we will show that Γ is conflicting. Let τ be a minimal set of tasks in Γ that cannot be scheduled. τ is minimal in the sense that removing any one task will make the rest of the tasks in τ schedulable[3]. Let r be the earliest release time and d the latest deadline among all tasks in τ. Let τ be scheduled by D.

Claim *When D schedules τ, there is no idle time between r and d.*

Proof of claim.

Suppose the system is idle time at time t then at that time there is no ready task. This means that τ can be partitioned into two non-empty sets (one with all tasks with deadline before t and the other with deadline after t). At least one of these sets cannot be scheduled[4] contradicting the minimality of τ.

End of proof of claim

Since the claim shows that there is no idle time and that D could not schedule all the tasks even while executing continuously we conclude that the sum of computation times needed for the tasks of τ is greater than the time that can be possibly allotted to them.

Remove_conflicts must have found a conflict in Γ. To see this notice that as long as no conflict is found the value of the variable collection_num_of_tasks is advanced and is bound

[3]Such a minimal set must exists since the entire set of tasks Γ cannot be scheduled but every singleton set of tasks can be scheduled.

[4]Recall that D is an optimal scheduler for underloaded systems [28 7].

to reach the value of num_of_tasks(τ). At that point all the tasks of τ are still present (i.e. were not merged into an aggregated task) and satisfy the condition of statement 10.

Hence Γ is conflicting. □

Lemma A.1.2 *When scheduling Γ, D—the earliest-deadline-first algorithm— will schedule to completion all conflict-free tasks.*

PROOF.

Let C be the time that can be "occupied" by the aggregated tasks that is the pointwise union of all their executable periods.

$$C = \cup_{T_i \text{ is an aggregated task}}[r_i, d_i] \tag{A.1}$$

One can verify (see statement 13 of remove_conflicts) that

$$C = \cup_{T_i \text{ is an (original) conflicting task}}[r_i, d_i] \tag{A.2}$$

Remove_conflicts(Γ) contains all the conflict-free tasks. It is conflict-free otherwise remove_conflicts would not have halted. Hence by lemma A.1.1 all the tasks in remove_conflicts(Γ) can be scheduled by D. The aggregated tasks of remove_conflicts(Γ) cannot be scheduled *outside* C. Moreover all of C must be occupied by aggregated tasks since they have no slack time. Hence the conflict-free tasks are scheduled by D using only time that lies outside C.

We showed that D schedules all the conflict-free tasks when the collection to be scheduled is remove_conflicts(Γ) but does this hold when D schedules the original set of tasks Γ? The answer is *yes*. When scheduling Γ equation A.2 above shows that all the time outside C is available to the conflict-free tasks hence by the previous paragraph all conflict-free tasks complete their execution when Γ is scheduled by D. □

Corollary A.1.3 *D can schedule all the conflicting-free tasks using only time outside C.* □

Lemma A.1.4 *Suppose T is not the current executing task and is not in* Q_privileged*. If T has an earlier deadline than all the tasks in* Q_privileged *and the current executing task (if any), then T, the current executing task, and all the tasks in* Q_privileged *can be scheduled by D.*

if and only if

$$\textsf{availtime} \geq \textit{remaining computation time}(T)$$

PROOF.
The proof is by induction on the scheduling decisions of D^{over}. The induction is done separately on each interval. □

Definition A.1.3

- REAL *LST* EVENT : According to D^{over} scheduling when a task completion event occurs the next task to be scheduled is the ready task T with the earliest-deadline. It is possible that the slack-time of T reached zero exactly when a task completion event occurred thus creating an *LST* event for T. We will call this *LST* event a *false* event since T would have been scheduled even without the interrupt. All other *LST* events will be called *real*.

In all of the following we ignore the false events. Only *real LST* events are considered.

Lemma A.1.5 *1. Let C be the time that can be occupied by the aggregated tasks,*

$$C \quad = \quad \cup_{T_i \text{ is an aggregated task}}[r_i, d_i]$$

then, outside C, D^{over} schedules according to earliest-deadline-first (D).

2. Under D^{over} scheduling a conflict-free task will never generate a (real) latest-start-time interrupt.

3. Let A be an aggregate task in remove_conflicts*(Γ) with parameters (r_a, d_a), then D^{over} will complete on or before r_a all conflict-free tasks with deadline on or before d_a.*

PROOF.
Recall that the aggregated tasks in remove_conflicts(Γ) are those tasks that were created "from" conflicting tasks. List all the aggregated tasks according to deadline order

$$T_{a_1}, T_{a_2}, T_{a_3}, \cdots$$

By the construction of these tasks we know that

$$r_{a_1} < d_{a_1} < r_{a_2} < d_{a_2} < r_{a_3} \cdots$$

(Actually from**remove_conflicts** one can infer only that $d_{a_1} \leq r_{a_2}$ but if it happens that $d_{a_1} = r_{a_2}$ we can for the purpose of the the following proof merge T_{a_1} and T_{a_2} into one aggregated task with parameters r_{a_1} and d_{a_2})

D^{over} departs from the earliest-deadline-first scheduling policy only when one of the following events occurs:

- The current task is *lst-scheduled* i.e. it was scheduled as the result of a latest-start-time interrupt.
- At a task release event or at a task completion event the task with the earliest deadline among all ready tasks is not scheduled because **availtime** is too small (see statement 61 and 72 of D^{over}).

D^{over} starts to schedule according to earliest-deadline-first. Before r_{a_1} there is no conflict hence by lemma A.1.1 there is no overload. This means that neither of the above conditions occurs (lemma A.1.4). Hence before the first aggregated task (up tor_{a_1}) D^{over} schedules in the same way as D. Also from corollary A.1.3 we conclude that all conflict-free tasks with deadline on or before d_{a_1} completed on or before r_{a_1}.

Between the first and second aggregated task i.e. between d_{a_1} and r_{a_2} there cannot be any ready conflicting tasks because all conflicting tasks have their deadlines before d_{a_1} or release time after r_{a_2}. So during this time only conflict-free tasks are scheduled. Moreover they will be scheduled according to earliest-deadline-first. We will show this by showing that neither of the two cases above can hold. A conflict-free task would not create a real *LST* event (corollary A.1.3 [5]). Also a task with the earliest-deadline will be immediately scheduled. This holds because if it is delayed then D encounters an overloaded situation while executing the conflict-free tasks outside C. This contradicts corollary A.1.3.

We conclude that up to r_{a_2} D^{over} acts like D and all the conflict-free tasks with deadline before d_{a_2} complete before r_{a_2}. The induction can proceed through the entire list of aggregated task and the lemma is proved. □

Corollary A.1.6 *D^{over} will schedule to completion all conflict-free tasks.*

[5]As a matter of fact the conflict-free tasks might have even used some of the time of C (when scheduled by D^{over}).

Lemma A.1.7 *Let A be an aggregate task in* remove_conflicts(Γ) *with parameters (r_a, d_a), then during (r_a, d_a) a conflict-free task will be scheduled by D^{over} only if there are no ready conflicting tasks.*

PROOF.

Lemma A.1.5 states that a conflict-free task with deadline on or before d_a would complete before r_a. So if any conflict-free task T with release time r and deadline d is to be scheduled during A it must satisfy $d > d_a$.

Suppose at time $t \in (r_a, d_a)$ there is a ready conflicting task T_i. Then $d_i < d_a$ must hold because T_i must be a part of the aggregated task A [6] [7].

Hence at time t all ready conflicting tasks have deadlines before the deadline of any conflict-free task. A conflict-free task can be scheduled in these circumstances only by a latest-start-time interrupt. This cannot occur because a conflict-free task will not generate a (real) latest-start-time interrupt (lemma A.1.5) $\square$

Theorem A.1.8 *D^{over} schedules to completion all conflict-free tasks and obtains at least $\frac{1}{(1+\sqrt{k})^2}$ the value a clairvoyant algorithm gets from all other (i.e., conflicting) tasks.*

PROOF.

The first part of this lemma is merely a repetition of corollary A.1.6. From lemma A.1.7 we conclude that D^{over} schedules the conflicting tasks regardless the presence of the conflict-free tasks. Suppose the clairvoyant algorithm has to schedule only the conflicted-tasks. It can schedule this tasks only during C. But we have just shown that D^{over} schedules the conflicted tasks as if the conflict-free tasks do not exist. Since D^{over} has a competitive multiplier of $(1+\sqrt{k})^2$ it is guaranteed to achieve at least $\frac{1}{(1+\sqrt{k})^2}$ of what a clairvoyant algorithm can achieve from all conflicting tasks. $\square$

[6] We say that a task T is a part of an aggregated task A if it is one of the tasks that were "merged" to create A.

[7] T_i is a conflicting task hence it is a part of some aggregated task B if this task is not A then the two aggregated tasks should be merged contradicting the fact that A is a task in remove_conflicts(Γ) hence $A = B$.

Appendix B

D^{over}: Gradual Descent

In the previous sections we assumed *firm* deadlines. That is a task has zero value if it misses its deadline. We would like to generalize to *soft* deadlines which means that a task may have some value even after its deadline.

We assume here a soft deadline scheme called *gradual descent* and show that a suitable variant of D^{over} is $(1+\sqrt{k})^2$ competitive in this case. D^{over} is also $(1+\sqrt{k})^2$ competitive in some possible generalizations of this scheme. We discuss these generalizations at the end of this appendix.

B.1 Exponential Gradual Descent

Consider the following *"exponential"* value assignment for gradual descent. If a task T_i with computation time c_i and value v_i does not complete by its deadline d_i (we call this deadline the *zeroth* deadline and denote it by d_i^0) then a value of $\frac{v_i}{2}$ can be obtained if it completes by $d_i + \frac{c_i}{2}$. This "deadline" is denoted by d_i^1. In general a value of $\frac{v_i}{2^y}$ is obtained the task completes by its y'th deadline $d_i^y = d_i + \frac{c_i}{2} + \frac{c_i}{4} + \cdots + \frac{c_i}{2^y}$. We keep the list of deadlines finite by postulating that a task's value density cannot go below 1. This means that the index of the last deadline after which the tasks has zero value is $\lfloor log_2(value\ density(T_i))\rfloor - 1 = \lfloor log_2(\frac{v_i}{c_i})\rfloor - 1$.

For notational convenience any task T_i will have *associated descending* tasks denoted by

$$T_i^0, T_i^1, T_i^2, \cdots, T_i^{\lfloor log_2(imp(T_i))\rfloor - 1}$$

where the release times and the computation times of all these tasks are equal to the release time and the computation time of T_i. T_i^y has a <u>firm</u> deadline at d_i^y and a value of $\frac{v_i}{2^y}$. Only one of the tasks associated with T_i can possibly complete. That is if we say that an algorithm executes T_i^y we mean that T_i completes by deadline d_i^y but after deadline d_i^{y-1}.

B.2 A Variant of D^{over} for Gradual Descent

We modify the latest-start-time interrupt handler of D^{over} in such a way that when T_i^0 is to be abandoned because it reached its *LST* but does not have enough value to be scheduled (see statement 89 of D^{over}) T_i^0 is indeed removed from all the data structures but in addition a task release for T_i^1 is simulated. T_i^1's remaining computation time is set to the remaining computation time of T_i^0. In the same way ifT_i^1 is to be abandoned then a third task is "released". This process continues as long as the value density does not go below 1.

B.3 Analysis of D^{over} in the Gradual Descent Model

The analysis is similar to one in section 3.2. We will discuss the differences only. Suppose that a collection of tasks Γ with importance ratio k is given and that D^{over} schedules this collection. We partition the collection of tasks according to the question of which associated tasks (if any) completed.

- Let S^p denote the set of tasks that completed successfully and that ended some positive time before their <u>zeroth</u> deadline.
- Let S^0 denote the set of tasks that completed successfully but ended exactly at their <u>zeroth</u> deadline.
- For $1 \leq y \leq \lfloor log_2 k \rfloor - 1$ letS^y denote the set of tasks that completed successfully after their $(y-1)$'th deadline but not after their y'th deadline (i.e. the$y$'th associated task completed).
- Let $FAIL$ denote the set of tasks that never completed.

We will start by modifying the technical lemmas of subsection 3.2.2 to the new setting.

B.3.1 Lemmas about D^{over}'s Scheduling

For notational convenience we define a minus one deadline d_i^{-1} which equals to the zeroth deadline d_i^0.

$-$In this setting lemma 3.2.2 reads

Lemma B.3.1

1. *For any task T_i in S^y (with $y \geq 0$). Suppose T_i^y completed at time $t_{complete} \leq d_i^y$, then*

$$[r_i, d_i^{y-1}] \subseteq [r_i, t_{complete}] \subseteq BUSY$$

2. For any task T_i in $FAIL$. Suppose T_i was abandoned at time t_{aban}, then

$$[r_i, t_{aban}] \subseteq BUSY$$

PROOF.
The proof is similar to that of lemma 3.2.2. □

−Lemma 3.2.3 holds without change. Note that we continue to make the normalized importance assumption because we never allow the value density to fall below 1.
−Lemma 3.2.4 holds without change.
−Lemma 3.2.5 reads:

Lemma B.3.2 *Suppose T_i^y was abandoned at time t in $I = [t_{begin}, t_{close}]$. Then,*

$$c_i \geq d_i^y - t_{close}$$

PROOF.
The proof is the same as the proof of lemma 3.2.5. □

B.3.2 How Well Can a Clairvoyant Scheduler Do?

As in subsection 3.2.3 given a collection of tasks Γ our goal is to bound the maximum value that a clairvoyant algorithm can obtain from scheduling Γ. We observe the scheduling of Γ by D^{over} which gives rise to the definition of S^p theS^y's and $FAIL$. As before $BUSY$ is defined to be the union of the periods in which the processor was not idle (under D^{over}'s scheduling).

The clairvoyant algorithm is offered the same two gifts as before. The first is the sum of the values of all tasks in S^p at no cost to it. The second gift is the *granted value.* That is in addition to the value obtained from scheduling

$$LATE = (S^0 \cup S^1 \cup \cdots S^{\lfloor log_2 k \rfloor - 1} \cup FAIL)$$

a value density of k will be granted for every period of $BUSY$ that is not used for executing a task of $LATE$. By a similar argument to lemma 3.2.6 we can see that[1]

$$C(LATE) \leq \max_{\substack{\textit{possible}\\ \textit{scheduling}\\ \textit{of } LATE}} \left\{ \begin{array}{c} \textit{value obtained by} \\ \textit{scheduling tasks of} \\ LATE \end{array} + k \cdot \begin{array}{c} \textit{length of time in } BUSY \textit{ not} \\ \textit{utilized by tasks of } LATE \end{array} \right\}$$

In lemma 3.2.8 we bounded the net gain that the clairvoyant algorithm could get from scheduling tasks of F [2]. This was done by examining each interval separately. If $T \in F$ is scheduled then its value is *accounted* for in the interval in which T was abandoned by D^{over}. Here the method of relating the value of a task $T \in LATE$ to the interval in which it is *accounted* for is more complicated. Suppose the clairvoyant algorithm chose to execute the z'th task of T_i to completion. D^{over} could have chosen to complete any of the associated tasks of T_i ($T_i \in S^y$ for some y) or none ($T_i \in FAIL$). In the first case we account for T_i^z in the interval in which D^{over} completed T_i; in the second case in the interval during which T_i^z was abandoned.

Assume that a clairvoyant scheduler selected an optimal scheduling for the tasks of $LATE$ considering the value that can be gained from leaving $BUSY$ periods idle. The execution of a task can give a positive net gain only if the task executed (at least partially) outside $BUSY$. The following lemma shows that such execution may take place only after t_{close}.

Lemma B.3.3 *Suppose the associated task T_i^z of $T_i \in LATE$ is scheduled to completion by the clairvoyant algorithm. Suppose that T_i is accounted for in $I = [t_{begin}, t_{close}]$. Then, if T_i is to be executed (by the clairvoyant algorithm) anywhere outside $BUSY$ it must be after t_{close}.*

PROOF.

There are two cases:

[1]Recall that $C(FAIL)$ denotes the value that a clairvoyant algorithm can achieve from scheduling $LATE$.

[2]Note that in section 3.2 the clairvoyant scheduler could not make any net gain from tasks of S^0 that completed in I because they can be executed only during $BUSY$. This is not the case here because if T_i^y completed in I the clairvoyant algorithm could choose a different completion point for T_i^y or even to abandon it in favor of another associated task T_i^z with $z \neq y$.

- D^{over} never completed T_i ($T_i \in FAIL$). In this case let t be the time when D^{over} abandoned T_i^z.
  T_i^z can be executed only during $\Delta_{T_i^z}$ which is is $[r_i, t] \cup [t, d_i^z]$. The first portion of $\Delta_{T_i^z}$ is contained in $BUSY$ (lemma B.3.1). The second portion is contained in I. Hence $[r_i, t_{close}] \subseteq BUSY$.
- D^{over} completed T_i^y for some y. Let t be the completion time of T_i^y.
  A similar argument as above for $\Delta_{T_i^y} = [r_i, t] \cup [t, d_i^y]$ shows that $[r_i, t_{close}] \subseteq BUSY$.

Hence in both cases ifT_i^z is to be executed outside $BUSY$ it must be after t_{close}. □

−Lemma 3.2.8 has to be replaced by the following

Lemma B.3.4 *With the above gifts, the total net gain obtained by the clairvoyant algorithm from scheduling the (associated) tasks* accounted for *in I is not greater than*

$$(1 + \sqrt{k}) \cdot \mathsf{achievedvalue}(I)$$

PROOF.
Let $T_1, T_2, \cdots T_m$ be those tasks that are accounted for in $I = [t_{begin}, t_{close}]$ and that the clairvoyant algorithm scheduled after t_{close} (in order of completion). These tasks execute for $l_1, l_2, \cdots l_m$ time *after* t_{close} (hence maybe outside $BUSY$ by the above lemma).
Denote by L the following value

$$L = \max\{\frac{(1 + \sqrt{k}) \cdot \mathsf{achievedvalue}(I)}{k}, l_1\} \tag{B.1}$$

Let j be the index less than or equal to m such that

$$\sum_{i \le j} l_i \le L < l_{j+1} + \sum_{i \le j} l_i$$

If no such j exists define j to be m.

First assume that we have an equality $\sum_{i \le j} l_i = L$.

The proof now has two parts.

◊*Part 1 :*

We will show that the net gain from scheduling tasks within a period of L after the end of the interval cannot be greater than $(1 + \sqrt{k}) \cdot \mathsf{achievedvalue}(I)$.

- Suppose that in B.1 the maximum is the first term. Then the total net gain from $T_1, T_2, \cdots T_j$ is not greater than

$$k \cdot \sum_{i \le j} l_i = k \cdot L = (1 + \sqrt{k}) \cdot \mathsf{achievedvalue}(I) \tag{B.2}$$

- Suppose the second term is maximum in B.1 and that the z'th associated task of T_1 was scheduled by the clairvoyant algorithm. If T_1^z was abandoned in I (by D^{over}) then lemma 3.2.4 ensures that its value is bounded by $(1+\sqrt{k}) \cdot \mathsf{achievedvalue}(I)$. The other possibility is that D^{over} completed T_1^y in I. If $z \ge y$ then $value(T_1^z) \le value(T_1^y)$ but $value(T_1^y)$ is a component of $\mathsf{achievedvalue}(I)$ so must be less or equal to it. $z < y$ implies that T_1 executed to completion (by the clairvoyant algorithm) before t_{close} since $d_i^z < d_i^y \le t_{close}$ — a contradiction.
  Hence in any case the value obtained by scheduling T_1 is at most $(1 + \sqrt{k}) \cdot \mathsf{achievedvalue}(I)$.

$\diamondsuit$*Part 2 :*

Now we will show that the net gain from scheduling a task T_i $(j < i \le m)$ L time after the end of I is never positive. Here we have to distinguish between two cases depending on whether D^{over} completed or abandoned T_i in I.

- D^{over} completed T_i
  Suppose that D^{over} completed T_i^y at $t_{complete} \in I$ and that the clairvoyant algorithm chose to schedule T_i^z.
  There are two possible cases:
  - $z < y$:
    Lemma B.3.1 shows that

$$[r_i, d_i^z] \subseteq [r_i, t_{complete}] \subseteq BUSY$$

    This means that T_i^z executes during $BUSY$ a contradiction.
  - $z \ge y$:
    The gradual descending scheme ensures that

$$\begin{aligned} d_i^z &= d_i^0 + \frac{c_i}{2} + \frac{c_i}{4} + \cdots + \frac{c_i}{2^z} \\ &= d_i^0 + (c_i - \frac{c_i}{2^z}) \end{aligned}$$

From lemma B.3.1 we see that

$$d_i^0 \leq d_i^{y-1} \leq t_{complete} \leq t_{close} \in BUSY$$

Hence we conclude that

$$d_i^z \leq t_{close} + (c_i - \frac{c_i}{2^z})$$

T_i^z must complete at or before d_i^z implying that the clairvoyant algorithm schedules T_i^z for at least $\frac{c_i}{2^z}$ time before t_{close} hence in $BUSY$. The loss from the execution during $BUSY$ is at least $\frac{c_i}{2^z} \times k$ while the value of T_i^z is at most $\frac{c_i \times k}{2^z}$. Hence the net gain is not positive.

- $T_i \in FAIL$

  Suppose that the y'th associated task of T_i was scheduled by the clairvoyant algorithm and that T_i^y was abandoned by D^{over} in $I = [t_{begin}, t_{close}]$. T_i^y has an execution time of at least $d_i^y - t_{close}$ by lemma B.3.2.

$$\begin{aligned} & d_i^y - t_{close} \\ \geq\ & \text{“the point at which } T_i^y \text{ completes (according to the clairvoyant)”} - t_{close} \\ \geq\ & (t_{close} + \sum_{g \leq i} l_g) - t_{close} \\ \geq\ & l_i + \sum_{g \leq j} l_g = l_i + L \end{aligned}$$

  T_i^y was scheduled by the clairvoyant scheduler but used only l_i time after t_{close}. Hence T_i executed at least L time before t_{close} that is to say in $BUSY$ (lemma B.3.3). The "loss" from scheduling T_i during $BUSY$ is at least $k \cdot L$. The value obtained by scheduling T_i is at most $(1+\sqrt{k}) \cdot \mathsf{achievedvalue}(I)$ (lemma 3.2.4). Hence the net gain is less than or equal to

$$\begin{aligned} & (1+\sqrt{k}) \cdot \mathsf{achievedvalue}(I) - k \cdot L \\ \leq\ & (1+\sqrt{k}) \cdot \mathsf{achievedvalue}(I) - (1+\sqrt{k}) \cdot \mathsf{achievedvalue}(I) \\ =\ & 0 \end{aligned}$$

What if L does not equal any of the partial sums? That is if$\sum_{i \leq j} l_i < L < \sum_{i \leq j+1} l_i$. As in the proof of lemma 3.2.8 we augment the total value given to the clairvoyant by some non-negative amount. Even with this addition the net gain achieved by the clairvoyant algorithm is bounded by $(1+\sqrt{k})\cdot \mathsf{achievedvalue}(I)$ hence proving the lemma. □

−Corollary 3.2.9 holds with $LATE$ replacing F.

Before we continue we must clarify the meaning of $\mathsf{poslaxval}$ and $\mathsf{zerolaxval}$ in this setting. $\mathsf{poslaxval}$ denotes the value obtained by tasks that completed before their <u>zeroth</u> deadline (tasks in S^p). $\mathsf{zerolaxval}$ denotes the total value obtained by tasks that completed at or after that deadline (i.e. tasks in$S^0 \cup S^1 \cup \cdots S^{\lfloor log_2 k \rfloor -1}$).

−Lemma 3.2.10 holds without change given these new definitions of $\mathsf{poslaxval}$ and $\mathsf{zerolaxval}$.

−Lemma 3.2.11 holds with $LATE$ replacing $F \cup S^0$.

Theorem B.3.5 *In the exponential gradual descent model, D^{over} has a competitive multiplier of* $(1+\sqrt{k})^2$.

PROOF.

Proof as in theorem 3.2.12. □

B.4 Inherent Bounds

The inherent bound given by Baruah et. al. [4 3] cannot be directly applied here. Hence it is not clear whether D^{over} is optimal in this setting. It might very well be that the introduction of descending value schemes helps the on-line scheduler more then it helps the clairvoyant one. Thus the question of finding the inherent bounds in this case is open.

B.5 Performance Guarantee for Underloaded Periods

In the gradual descent model we define an underloaded collection of tasks as a collection such that all its tasks can be scheduled by the zeroth deadline (i.e. with their full value). It is clear that D^{over} will get 100% of the value for such a collection since it will execute according to earliest deadline first scheduling.

B.6 Other Gradual Descent Schemes

In this section we presented a specific scheme of gradual descent. In fact the current argument can provide the same result for more general schemes of descending value.

All schemes must have the following properties:

- The value density of a task must not go below 1 (used in lemma 3.2.3).
- For every possible associated task T_i^z of T_i

  $$d_i^z < d_i^0 + c_i$$

  and

  $$(d_i^z - d_i^0) \times k \geq \text{“the value of } T_i^z\text{”}$$

  (used in part 2 of lemma B.3.4)

Within these constraints many schemes are possible. Some tasks can have firm deadlines; others obtain values that decrease monotonically as the distance from the deadline increases. The base of the exponent (Base 2 used here is an arbitrary choice) can be different for different tasks.

Appendix C

D^{over}: Exact Computation Time Is Not Known

Suppose the on-line scheduling algorithm is not given the exact computation time of a task upon its release. However for every task T_i an upper bound on its possible computation time denoted by $c_{i,max}$ is given. Also the actual computation time of T_i denoted by c_i satisfies:

$$(1-\epsilon)\cdot c_{i,max} \le c_i \le c_{i,max}$$

Where $0 \le \epsilon < 1$ is a given *error margin* which is common to all the tasks. We make the following additional assumptions:

Assumption C.0.1

- THE ACTUAL COMPUTATION TIME IS ENVIRONMENT-INVARIANT: The actual computation time of a task does not depend on the point in time in which it was scheduled the number of times it was preempted and rescheduled etc.
- THE ACTUAL COMPUTATION TIME IS NOT KNOWN BEFORE THE COMPLETION POINT: An on-line scheduler cannot know the exact computation time of a task until it completes.

Some terms has to be redefined in the new set up:

Definition C.0.2

- UNDERLOADED COLLECTION OF TASKS: A collection of tasks is *underloaded* (in this setting) if the <u>actual</u> computation times enable execution of all the tasks to completion.
- IMPORTANCE RATIO: The *importance ratio* k of a collection with an error margin of ϵ is defined to be the ratio of the largest *possible* value density to the smallest *possible* value density.

$$k = \frac{\max_i \frac{v_i}{(1-\epsilon)\cdot c_{i,max}}}{\min_i \frac{v_i}{c_{i,max}}} = \frac{1}{(1-\epsilon)} \cdot \frac{\max_i \frac{v_i}{c_{i,max}}}{\min_i \frac{v_i}{c_{i,max}}} \tag{C.1}$$

Here the *normalized importance* assumption (assumption 2.0.2) means that

$$\min_i \frac{v_i}{c_{i,max}} \ge 1$$

C.1 An Inherent Bound On The Competitive Multiplier

The inherent bound proof of Baruah et. al. [4 3] can be applied here as well. In the notation of those references all the *major tasks* execute at their longest possible computation time with an actual value density 1 while all the *associated tasks* execute at their shortest possible computation time and value density k. This argument shows that no on-line scheduler can achieve a competitive multiplier smaller than $(1 + \sqrt{k})^2$.

C.2 Underloaded Systems

Example C.2.1 Suppose we are given the following collection of two tasks:

Task	*Release-Time*	*Max. Computation-Time*	*Value*	*Deadline*
T_1	0	1	1	1
T_2	0	200	200	200

For an error margin $\epsilon < \frac{1}{201}$ this collection will always constitute an *overloaded* system. However if $\epsilon \geq \frac{1}{201}$ then depending on the actual computation times the system may be either underloaded or overloaded. □

Theorem C.2.1 *An on-line scheduler that guarantees 100% of the value for an underloaded system is not competitive.*

PROOF.

Suppose an on-line scheduler S guarantees 100% of the value for underloaded systems. Suppose the tasks of example C.2.1 with error margin of $\epsilon = \frac{1}{200}$ are scheduled by S. Consider the following possible cases:

1. The actual executing time of T_1 is the maximum possible — 1 while that of T_2 is the minimum possible — 199. In this case the system is underloaded and S should be able to schedule both tasks to completion. That is schedule T_1 between 0 and 1 and T_2 from 1 to 200.
2. The actual executing time of both T_1 and T_2 are the maximum possible. In this case the system is overloaded and only one of the tasks can possibly complete. However S cannot distinguish between case 1 and case 2 (not before time 200). Hence S will

schedule T_1 between 0 and 1 and T_2 will reach its deadline without completing its execution.

In the second case S obtains a value of 1 out of the possible value of 200. Hence S has a competitive multiplier of at least 200. Of course 200 is an arbitrary number and can be as large as wanted which gives the desired result. $\square$

C.3 Overloaded Systems

Theorem C.2.1 shows that we cannot guarantee both a finite competitive multiplier and a 100% the value for an underloaded system.

The *earliest-deadline-first* algorithm is an optimal on-line scheduler for *underloaded* systems. We will show that a version of D^{over} can achieve a competitive multiplier of $(1+\sqrt{k})^2 + (\epsilon \cdot k)(1+\sqrt{k}) + 1$.
We utilize the following *version* of D^{over}:

- k is taken to be as in equation C.1.
- D^{over} assumes that the computation time of a task to be the maximum possible — $c_{\cdot,max}$. This affects the values of *remaining_computation_time* availtime *laxity* and the *LST* point of a task (statements 20 23 24 36 57 61 66 67 and 77).

Theorem C.3.1 *D^{over} has a competitive factor of* $\frac{1}{(1+\sqrt{k})^2+(\epsilon \cdot k)(1+\sqrt{k})+1}$
PROOF.
The proof will be an adaptation of the analysis for the case of exact knowledge of computation time in section 3.2. The following is a list of modification that are needed in that analysis:

1. Lemma 3.2.5 should read :

$$c_{i,max} \geq d_i - t_{close}$$

   Hence

$$c_i \geq c_{i,max} \cdot (1-\epsilon) \geq d_i - t_{close} - \epsilon \cdot c_{i,max}$$

2. In this set up lemma 3.2.8 should be replaced by:

Lemma C.3.2 *The total net gain from scheduling the tasks abandoned during I is not greater than*

$$(1+\sqrt{k})(1+\epsilon\cdot k)\cdot \mathsf{achievedvalue}$$

The proof is essentially the same as in the proof of lemma 3.2.8 but here the value of L is taken to be[1]:

$$L = \max\{(1+\sqrt{k})\cdot(\frac{1}{k}+\epsilon)\cdot \mathsf{achievedvalue}(I), l_1\}$$

- The total net gain from those tasks of F $T_1, T_2, \cdots T_j$ whose total computation time after t_{close} equals L is not greater than

$$k\cdot L = (1+\sqrt{k})(1+\epsilon\cdot k)\cdot \mathsf{achievedvalue}(I)$$

- Every other task T_i where $j < i \le m$ has an execution time of at least

$$d_i - t_{close} - \epsilon\cdot c_{i,max} \ge L + l_i - \epsilon\cdot c_{i,max}$$

  T_i was scheduled by the clairvoyant scheduler but used only l_i time after t_{close}. Hence T_i executed at least $L - \epsilon\cdot c_{i,max}$ time before t_{close} that is to say in $BUSY$.

$$\begin{array}{lcl} & L-\epsilon\cdot c_{i,max} & \\ \ge & L-\epsilon\cdot v_i & \text{, by assumption 2.0.2 } c_{i,max}\le v_i \\ \ge & L-\epsilon\cdot(1+\sqrt{k})\cdot \mathsf{achievedvalue}(I) & \text{, by lemma 3.2.4} \\ \ge & \frac{(1+\sqrt{k})\cdot \mathsf{achievedvalue}(I)}{k} & \end{array}$$

  The "loss" from scheduling T_i during $BUSY$ is at least $k\cdot\frac{(1+\sqrt{k})\cdot \mathsf{achievedvalue}(I)}{k}$. The value obtained by scheduling T_i is at most $(1+\sqrt{k})\cdot \mathsf{achievedvalue}(I)$ (lemma 3.2.4). Hence the net gain is less than or equal to zero.

[1]Instead of $L = \max\{\frac{(1+\sqrt{k})\cdot \mathsf{achievedvalue}(I)}{k}, l_1\}$ in section 3.2

3. Lemma 3.2.10 should state that

$$\begin{aligned}
C(S^0 \cup F) &\leq (1+\sqrt{k})(1+\epsilon \cdot k) \cdot \mathsf{achievedvalue} + k \cdot BUSY \\
&\leq (1+\sqrt{k})(1+\epsilon \cdot k) \cdot \mathsf{achievedvalue} \\
&\quad + k \cdot (\mathsf{achievedvalue} + \frac{1}{\sqrt{k}} \cdot \mathsf{lstvalue}) \\
&= (1+\sqrt{k}+k+(\epsilon \cdot k)(1+\sqrt{k})) \cdot \mathsf{achievedvalue} + \sqrt{k} \cdot \mathsf{lstvalue} \\
&\leq ((1+\sqrt{k})^2 + (\epsilon \cdot k)(1+\sqrt{k})) \cdot \mathsf{achievedvalue}
\end{aligned}$$

The first inequality follows from the fact that lemma 3.2.3 holds without change. The last inequality is due to the fact that lstvalue is always less or equal to achievedvalue.

Finally we can prove the theorem:

$$\begin{aligned}
C(\Gamma) &= C(F \cup S^0 \cup S^p) \leq C(F \cup S^0) + C(S^p) \\
&\leq C(F \cup S^0) + \mathsf{poslaxval} \\
&\leq ((1+\sqrt{k})^2 + (\epsilon \cdot k)(1+\sqrt{k})) \cdot \mathsf{achievedvalue} + \mathsf{poslaxval} \\
&\leq ((1+\sqrt{k})^2 + (\epsilon \cdot k)(1+\sqrt{k}) + 1) \cdot \mathsf{achievedvalue}
\end{aligned}$$

The last inequality is due to the fact that poslaxval is always less or equal to achievedvalue.
□

Appendix D

Two Processor Systems

This appendix deals with systems having only two processors. We also assume *uniform value density* (i.e. $n = 2, k = 1$). 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 cannot execute on any other processor. An on-line scheduler can do better when migration is possible.

D.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 (figures D.1- D.2) depict the code of the *Safe-Risky* algorithm:

Remark D.1.1 All privileged tasks should be able to complete using *only a single* processor at a time. This mean that the privileged tasks constitute a *uniprocessor* underloaded system. A new task is accepted to *Q_privileged* only if it does not create an overload when added to the tasks already in *Q_privileged*.

Note an important difference between the above algorithm (and for that matter also D^{over} and the version of the *MOCA 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 schedulability test*). This stands in contrast to the *global schedulability*

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

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.                               *)

Q_privileged := $\phi$;
Q_waiting    := $\phi$;

loop :
 task release : (* $T$ is released *)
    if (SP is idle ) then
        schedule $T$ on SP;
    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
              preempt current task;
              add the current task to Q_privileged;
              schedule $T$ on SP;
        else    (*SP is not idle and $T$ cannot be scheduled on SP *)
            add $T$ to Q_waiting;
    endif
end (*task release *)
```

Figure D.1: The *Safe-Risky* algorithm- a scheduler for two processors systems.

```
40  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)                                                                    *)
41   42 if (RP is idle) then
43          schedule $T$ on RP;
        else    (* RP is not idle and $T$ can not be scheduled on SP *)
44          let $T_{RP}$ be the current task on RP;
45          if ($value(T) > value(T_{RP})$) then
46            abandon $T_{RP}$;
47            schedule $T$ on RP;
48          else
49            abandon $T$;
50          endif (*comparing values *)
51      endif (*RP is not idle *)
52  end (*LST *)

53   task completion :
    (* on SP; There is no special event when a task completes on RP *)
54
55      let $T$ be the task with earliest deadline in Q_waiting;
56      if ($T$ has earlier deadline than the current task on SP
            and can be scheduled **with all** other privileged tasks) then
57            remove $T$ from Q_waiting;
58            add $T$ to Q_privileged;
59          (* which amounts to scheduling $T$ on SP *)
60      endif (*$T$ can be added to Q_privileged *)
61      if (all the privileged tasks can complete **with** the
            task currently executing RP) then
62            preempt the current task on RP;
63            add it to Q_privileged;
64            (* this task will be scheduled on SP by the following piece of code. This
              is the only place where migration is used *)
65      endif
66      schedule on SP the task with earliest deadline in Q_privileged
        (if any);
67  end (*task completion *)
68 end {loop}
```

Figure D.2: The *Safe-Risky* algorithm- the Latest Start Time and the Task Completion routines.

task used by the *MOCA Algorithm.* That is: a new task enters *Q_privileged* *if and only if* it does not create overload irrespectively of its deadline.

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

In this section we show that the *Safe-Risky* algorithm has a competitive multiplier of 2. Our approach is to partition the execution into disjoint intervals the *Safe-Risky* algorithmgets 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 *Safe-Risky* algorithmuses at least one processor productively during all that period. The clairvoyant scheduler could gain a factor of two by utilizing both processors.

Definition D.2.1

- EARLINESS, LATEST_AFFECTED, COVER_END: 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 the *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

$$\mathit{cover\_end}(T) \overset{\text{def}}{=} \max\{\mathit{deadline}(T), \mathit{latest\_affected}(T)\}$$

- 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 D.2.1 *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 D.2.2

- C(I), V(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.

Definition D.2.3

- CONFLICTED TASK, MAX_COVER: We say that T is a *conflicted task if it conflicts with the tasks of Q_privileged (when considered for execution at the task release routine, task completion routine or the LST routine, see statements 32, 56 and 41) and hence was diverted to RP (and later was either scheduled to execution or was abandoned).*

  Define max_cover as,

$$max\_cover = max\_cover(I) \stackrel{\text{def}}{=} \max_{\substack{T \text{ is a} \\ \text{conflicted task}}} cover\_end(T)$$

  If there is no conflicted task this max equals 0.

We are going to show that for every task that was abandoned[1] SP was productive during its entire executable period. This is true for all the tasks that were abandoned during some

[1]Every task that was abandoned is a conflicted-task.

interval. Hence the *Safe-Risky* algorithm gets a value of at least the union of all these periods that is *max_cover*.

Lemma D.2.2 *For every task T, if T is a conflicted task (i.e, was diverted to RP by the Safe-Risky algorithm) then*[2]*, the total value obtained by the Safe-Risky algorithm is at least cover_end(T).*

PROOF.

Recall that $\mathit{cover\_end}(T) = \max\{\mathit{deadline}(T), \mathit{latest\_affected}(T)\}$

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

  T is diverted to RP at its latest start time which is at $\mathit{deadline}(T) - \mathit{computation\_time}(T)$. This means that the interval length is at least $\mathit{deadline}(T) - \mathit{computation\_time}(T)$ (recall that the interval starts at time 0). Lemma D.2.1 says that the value obtained by the *Safe-Risky* algorithm on SP is at least $\mathit{deadline}(T) - \mathit{computation\_time}(T)$.

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

$$\begin{aligned} V(I) &\geq (\mathit{deadline}(T) - \mathit{computation\_time}(T)) + \mathit{value}(T) \\ &= (\mathit{deadline}(T) - \mathit{computation\_time}(T)) + \mathit{computation\_time}(T) \\ &= \mathit{deadline}(T) \end{aligned}$$

- Now we are going to show that $V(I) \geq \mathit{latest\_affected}(T)$.

  By definition *latest_affected*(T) is the deadline of some privileged task $T_{affected}$. Let t_{actual} be the time when $T_{affected}$ completed (according to the *Safe-Risky* algorithm). t_{actual} cannot be any earlier than the estimated completion time of $T_{affected}$ computed when T was considered for execution (the actual time can be in fact later than the estimated time if some tasks were later added to *Q_privileged*). The length of the interval is at least t_{actual}. Lemma D.2.1 shows that the value obtained on SP is at least this length.

  As mentioned before the *Safe-Risky* algorithm an additional value of at least $\mathit{value}(T)$. Hence

$$V(I) \geq t_{actual} + \mathit{computation\_time}(T)$$

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

$$\begin{aligned} &\geq \textit{deadline}(T_{affected}) \\ &= \textit{latest\_affected}(T) \end{aligned}$$

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

□

Corollary D.2.3 *The value obtained by the Safe-Risky algorithm during an interval is at least*

$$\max\{\textit{the interval length}, \textit{max\_cover}\}$$

□

Definition D.2.4

- 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*$(I) = \phi$ (the empty set).

We will show that during the first portion of an interval (from 0 to *max_cover*) the *Safe-Risky* algorithm gets a value of at least one half the value obtainable from the tasks of *covered*. If there are other tasks (i.e. tasks of*uncovered*) then they are all going to complete successfully without competing with *covered* tasks (either before or after *max_cover*).

Lemma D.2.4 *For an atomic_interval I,*

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

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·*max_cover*. Corollary D.2.3 ensure that the *Safe-Risky* algorithm gets a value of *max_cover* at least.

□

Now we are ready to extend the above lemma to a general interval. We are going to use induction. The result is already given for *atomic_interval*. We will show that a general interval can be broken into *atomic_intervals* and "uncovered" periods between them. The *covered* tasks "cover" the *atomic_intervals*. That is they generate value of at least the combined length of these intervals. The tasks of *uncovered* execute without interfering with the *covered* task and generate value of at least the length of the uncovered periods.

Lemma D.2.5 *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.*
- *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 the tasks of 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* cannot 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[3]. But if this is the case all the tasks of *Q_privileged* could be scheduled on one processor along with T_{rp}. Hence T_{rp} would have migrated to SP a contradiction to the fact that T_{rp} is still executing on RP. Note that the ability to migrate (in statement 61) is used to get the contradiction (T_{rp} could have and was supposed to migrate but it didn't). This is the only place in the analysis where migration is used.

- The execution of a task of *covered* on SP can be delayed or interrupted (i.e pre-empted) 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* cannot 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*.

□

Lemma D.2.6 *For any interval I,*

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

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

Basis: If the interval is *atomic_interval* then lemma D.2.4 ensures that the induction hypothesis holds.

[3]It cannot 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}.

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[4]

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

Lemma D.2.5 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. □

We are ready to state the main theorem of this section.

Theorem D.2.7 *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. □

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

D.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[5]. We call this variant the *Safe-Risky-(fixed)*.

D.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 (see figure D.3).

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

    let $T$ be the task with earliest deadline in $Q\_waiting$;
    if ( $T$ has earlier deadline than the current task on SP
        and can be scheduled **with all** other privileged tasks) then
          remove $T$ from $Q\_waiting$;
          add $T$ to $Q\_privileged$;
          (* which amounts to scheduling $T$ on SP *)
    endif (*$T$ can be added to $Q\_privileged$ *)
    schedule on SP the privileged task with earliest deadline (if any);
    if (SP is idle) then
        switch roles between the SP and RP processors;
    endif (*SP is idle *)
end (*task completion *)
```

Figure D.3: The *Safe-Risky-(fixed)* algorithm.

[5]When tasks have no slack time an optimal algorithm with competitive multiplier of 2 is known [38,3].

D.3.2 Analysis

We define *latest_affected* and all the other definitions as before. Observe that lemmas D.2.1 and D.2.2 hold for the new algorithm as well.
Recall, that F denotes the set of tasks abandoned by the on-line algorithm

Lemma D.3.1 *For every interval, I,*

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

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 [6]. The best the clairvoyant can do, with tasks of F, is to schedule both processors continuously to get a value of $2d$. Lemma D.2.2 shows that the *Safe-Risky-(fixed)* gets a value of at least d. $\square$

Theorem D.3.2 *For a system of uniform value density and 2 processors, the Safe-Risky-(fixed) achieves a competitive multiplier of* 3 *(when migration of tasks is not permitted).*
PROOF.
This follows from lemma D.3.1 and lemma D.2.5 (applied to the no migration case) and the Lost Value Lemma (lemma 4.3.1). $\square$

D.4 Scheduling Overhead

In this section we study the cost of executing the scheduling algorithms themselves. What is the cost of testing whether a newly arriving task[7] can be added to *Q_privileged* without causing overload? This test can be done in a constant number of operations (as in D^{over}).

Q_waiting is a 2-3 tree organized according to Latest Start Time. Hence, inserting and removing a task from this queue costs $O(logM)$ operations. Where, M is a bound on the total number of ready tasks at any given moment in *Q_waiting*.

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

[6]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.
[7]With an earlier deadline then all the tasks in *Q_privileged*.