The Power of Migration in Multi-Processor Scheduling of Real-Time Systems.

Gilad Koren* Netanya College and Bar-Ilan University

Amihood Amir† Georgia Tech and Bar-Ilan University

Emanuel Dar‡ Bar-Ilan University

September 23, 1999

Abstract

In this paper we study the performance of off-line multiprocessor real-time schedules that allows task migration compared to those that forbid migration. We consider an **off-line** scheduling problem in which a given collection of tasks each with release time, computation time and **deadline** are to be run on a multi-processor system. A preemptive schedule allows the execution of a task to be temporarily suspended and resumed at a later time. A **migrative** schedule allows the task to resume on any processor whereas the **non-migrative** schedule allows the task to resume only on the processor it was initially started. A schedule *value* is the summation of all the values of all the tasks that were completed by their deadlines. In this paper we assume that a task value is proportional to its computation time.

We present lower and upper bound results. For a system with n processors, We construct a non-migrative schedule that is guaranteed to achieve at least $1 - \left(1 - \frac{1}{2n}\right)^n$ of the **optimal** migrative schedule value.

In addition, there are task sets, for which even an optimal non-migrative schedule achieves at most $n \,/\, (2n-1)$ of the optimal migrative value. Asymptotically (as $n \to \infty$) our upper bound approaches 1/2 and the lower bound approaches $1 - \frac{1}{\sqrt{e}} \sim 0.3935$.

1 Introduction

Real-time systems can be most generally described as systems that need to complete certain tasks within a specified time limit. Traditionally, these are systems that handle critical processes such as power stations, nuclear reactors, or space vehicles. In many such applications, failure of the task to complete in time may have disastrous results. Such systems are called *Hard Real-Time Systems.*

*School of Mathematics and Computer Science, Netanya Academic College, 42365 Netanya, Israel and Department of Mathematics and Computer Science, Bar-Ilan University, 52900 Ramat Gan, ISRAEL; koren@cs.biu.ac.il;

†Department of Mathematics and Computer Science, Bar-Ilan University, 52900 Ramat Gan, ISRAEL; (972-3)531-8770; amir@cs.biu.ac.il; Partially support by NSF grant CCR-96-10170, the Israel Ministry of Science and Arts grants 6297 and 8560 and a 1997 Bar-Ilan University Internal Research Grant.

‡Department of Mathematics and Computer Science, Bar-Ilan University, 52900 Ramat Gan, ISRAEL; dar@cs.biu.ac.il;

A less stringent type of real-time system is one where every task still has to be completed by a given deadline (it is then said to *succeed*) otherwise it *fails*. However, such a failure merely means that a certain stimulus has not been responded to, it does not have catastrophic connotations. An example may be a case where the stimuli arrive at a quick rate. Failure to respond to a number of them is still tolerable, as long as a certain ratio succeeds. In this paper we are concerned with the latter type of real-time systems, the *Soft Real-Time Systems*. Even within the realm of soft real-time systems, several models appear in the literature. We concern ourselves with the *Firm Real-Time Systems*. In this model each task has a **value** which is obtained by the system only if the task completes by its deadline.

A *scheduling algorithm* for a real-time system is presented with tasks, each of which has a *release time* from which it can start working, a *deadline* by which it must complete working if it is to be successful, and a *computation time*, the time it actually needs in order to run in its entirety. Each task has a *value* obtained if completed successfully. The special case where the value of a task is proportional to its running time is called the *Uniform Value Density* case [3, 10].

The aim of the system is to maximize the sum of the values obtained from all successfully completed tasks. It is the duty of the scheduler to schedule the tasks that maximize this total value.

A number of issues are raised at this point. A scheduler that knows in advance all the tasks and their parameters is called an *off-line* or *clairvoyant* scheduler. If it is only presented with a task upon its release then the scheduler is called an *on-line* scheduler. The model is said to allow *preemption* if the scheduler may interrupt a working task. Various different types of preemption exist. A preempted task may proceed later from the point it was preempted, forced to start from the beginning, or not be allowed to run again. In this paper we are concerned with real-time systems that allow preemption. A preempted task may later resume its execution where it left off.

A significant amount of work has appeared about the various possibilities [5, 4]. In a single processor environment, it has been shown that *underloaded* systems, i.e. where there exists a (possibly clairvoyant) schedule that allows all tasks to successfully complete, there exist on-line schedulers with 100% guarantee [6]. In the *overloaded case* such an optimal schedule is impossible [11]. Scheduling algorithms such as D^{over} give a competitive guarantee for overloaded firm real time systems [10].

In this paper we are concerned with *multiprocessor* real-time systems with preemption. Mok and Dertouzos [7] showed that even in an underloaded system, no on-line algorithm can guarantee 100% success. A task is said to *migrate* if it continues running on a different processor from the one it was preempted in. Koren and Shasha [9], give an algorithm ($MOCA$) with a competitive guarantee for a multiprocessor real-time system with no *migration*. $MOCA$ does not use migration but its guarantee holds even when migration is allowed. There are a number of differences between the case of $MOCA$ and our problem. (1) $MOCA$ is an on-line algorithm, while we study the off-line case. (2) $MOCA$ does not offer any guarantees in the *uniform value density case* – where the value of a task is proportional to its running time. This is precisely the case we handle.

Bar-Noy et. al. [1] study the the case of uniform value density but in their model tasks have no slack time (must be scheduled upon release, if at all). They give an upper bound of 0.66 on the best possible competitive factor. Bar-Noy, Mansour and Schieber [2] devised on-line scheduling algorithms for this case with competitive factor of $\frac{1}{2} - \frac{1}{4n}$ for even number of processors n, and $\frac{1}{2}$ for odd number.

To our knowledge, *there is no on-line scheduler* for a real-time multiprocessor system *with migration* that has any non-trivial *competitiveness guarantees* for the uniform value density case of tasks with slack-time.

We would like to investigate the power of migration. There are both practical and theoretical reasons for such an investigation. From a practical perspective, systems without migration are less centralized and less complicated. However, migration gives more power in processor utilization, in versatility and in fault handling. It may also be easier to prove bounds on systems with migration.

This paper is a first step in the direction of understanding the power of migration in real-time systems. Specifically, we would like to answer the following question: *In the Uniform Value Density case, does a clairvoyant scheduler without migration have the same power as a clairvoyant scheduler with migration?* If we can pinpoint the *performance ratio* (as defined below) between off-line schedulers with migration and without, we will have gone a step forward in obtaining on-line schedulers. We will be able to limit ourselves to either the case of migration or non-migration. Any competitive on-line scheduling algorithm that will be obtained, will guaranteed a competitive on-line scheduling algorithm for both cases.

Definition 1.1 *A set of tasks S, in a system with n processors, has a* performance ratio *of* $\omega_n(S) = \omega_n$ *if*

$$\omega_n = \frac{\{value\ obtained\ by\ an\ optimal\ \mathbf{non}{-}migrative\ schedule\ for\ S\}}{\{value\ obtained\ by\ an\ optimal\ migrative\ schedule\ for\ S\}}$$

An off-line non-migrative scheduling algorithm $\mathcal{A}$ in a system with n processors, has a performance ratio *of* $\omega_n(\mathcal{A}) = \omega_n$ *if for all S*

$$\omega_n \leq \frac{\{value\ obtained\ when\ \mathcal{A}\ schedules\ S\}}{\{value\ obtained\ by\ an\ optimal\ migrative\ schedule\ for\ S\}}$$

1.1 Main results

The contribution of this paper is two-fold. In the "grand view" of computer science we add our voice to the impressive body of work dealing with theoretical complexity analysis of operating systems. The operating systems area is at the heart of computer science and understanding the power and limitations of its various concepts is of fundamental importance to our field. It is not surprising that much work has historically been done in this area, particularly in job scheduling (see e.g. [12, 8])

However, the area of real-time systems, albeit important in practice, has been relatively neglected in the theory literature. Real-time systems grapple with different issues thus the scheduling literature does not generally provide answers to such problems. This paper is a modest addition to a growing number of works that deal with theoretical issues in real-time systems.

The more specific contribution of this paper is in providing both an upper and lower bound on the performance ratio of off-line schedulers without migration vs. with migration in a multiprocessor real-time system with n processors.

- We show an example where an off-line scheduler without migration cannot schedule more than $\frac{n}{2n-1}$ the value of the optimal schedule with migration.
- We describe a scheduler that without using migration obtains a value of at least $1-\left(1-\frac{1}{2n}\right)^n$ the optimal value using migration.

Note that asymptotically (as $n \to \infty$) our upper bound approaches $1/2$ and the lower bound approaches $1-\frac{1}{\sqrt{e}} \sim 0.3935$.

This paper is constructed as follows: We start with some notations and definitions in section 2. In section 3 we show the upper bound on the performance ratio. Section 4 is an overview of the scheduler's main ideas and proof of the lower bound on the performance ratio. Section 5 presents the algorithm that, inputing the system tasks' parameters, constructs a non-migrative schedule with the necessary guarantees for section 4. We conclude with some open problems.

Note on Terminology: We have defined an *off-line* scheduling problem. Therefore all tasks and their parameters are known from the start although the tasks are not ready to run until some specified time in the future which we denote here as *release time*. In describing our schedulers, the reader will notice occasional use of on-line-like language: "*the task just released*" , "*the new task*" etc. The reader should understand that these are just form of expressions that we consider natural to describing a possible future situation considered by the scheduler. In reality the entire computation is done *off-line*.

2 Preliminaries: Notation and an Example

We view time as the set of real non negative numbers R^+, however the results are equally valid for Q^+ or N^+.

Definition 2.1 *Let T be a task. We denote its parameters by $T\left(r\left(T\right), c\left(T\right), d\left(T\right)\right)$ where $r\left(T\right)$ is the* release time, *$c\left(T\right)$ is the* computation time, *and $d\left(T\right)$ is the* deadline.

A legal schedule *of a task set S on n processors is a schedule where each task T is scheduled for execution only between $r(T)$ and $d(T)$ for a total length that does not exceed $c(T)$. In addition, a task may execute on at most one processor at any given moment. Note that a schedule is legal even if some tasks fail. In a* legal non-migrative schedule *there is the additional constraint that a task may not run on more than one processor.*

Example 2.2 *Performance Ratio*

Consider a system with two processors ($n = 2$) and the following tasks set S of size three:

$$S = \left\{T_1\left(0, \frac{2}{3}, 1\right), \ T_2\left(0, \frac{2}{3}, 1\right), \ T_2\left(0, \frac{2}{3}, 1\right)\right\}$$

An optimal non-migrative schedule will schedule $\{T_1, T_2\}$ to obtain a value of $c(T_1) + c(T_2) = \frac{2}{3} + \frac{2}{3} = 1\frac{1}{3}$ (see fig. 1). An optimal migrative schedule will schedule all three tasks obtaining a value of $c(T_1) + c(T_2) + c(T_3) = \frac{2}{3} + \frac{2}{3} + \frac{2}{3} = 2$ (see fig. 2). Hence, $\omega_2\left(S\right) = \frac{1\frac{1}{3}}{2} = \frac{2}{3}$.

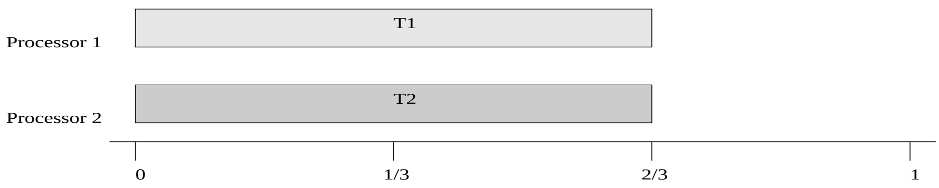

Figure 1: A non-migrative schedule

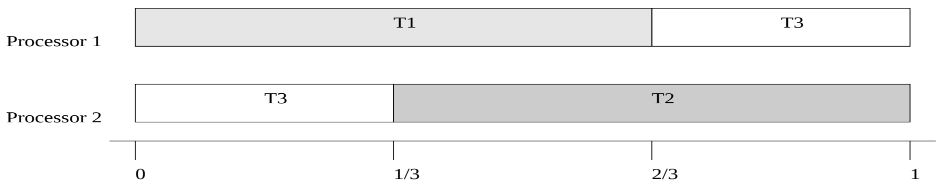

Figure 2: A migrative schedule

3 The Upper Bound

We begin by demonstrating an upper bound on the performance ratio. The example below shows that for every number of processors n there are task sets that can be scheduled must fail when migration is not allowed.

Theorem 3.1 *For any number of processors n, $n \geq 2$, there exist a task set, S^n, such that* $\omega_n\left(S^n\right) = \frac{n}{2n-1}$.

PROOF.

Let S^n be a set of $2n-1$ identical tasks: $T_i\left(0, \frac{n}{2n-1}, 1\right)$ (all with release time 0, deadline at 1 and computation time of $\frac{n}{2n-1}$).

An optimal off-line migrative schedule can successfully complete all those tasks (see figure 3), therefore obtains a value of

$$\sum_{i=1}^{2n-1} T_i = (2n-1)\frac{n}{2n-1} = n$$

An optimal non-migrative schedule, completes at most one task on each processor, hence a total of n tasks obtaining a value of

$$n\frac{n}{2n-1} = \frac{n^2}{2n-1}$$

The performance ratio is therefore:

$$\omega_n\left(S^n\right) = \frac{\frac{n^2}{2n-1}}{n} = \frac{n}{2n-1}$$

$\square$

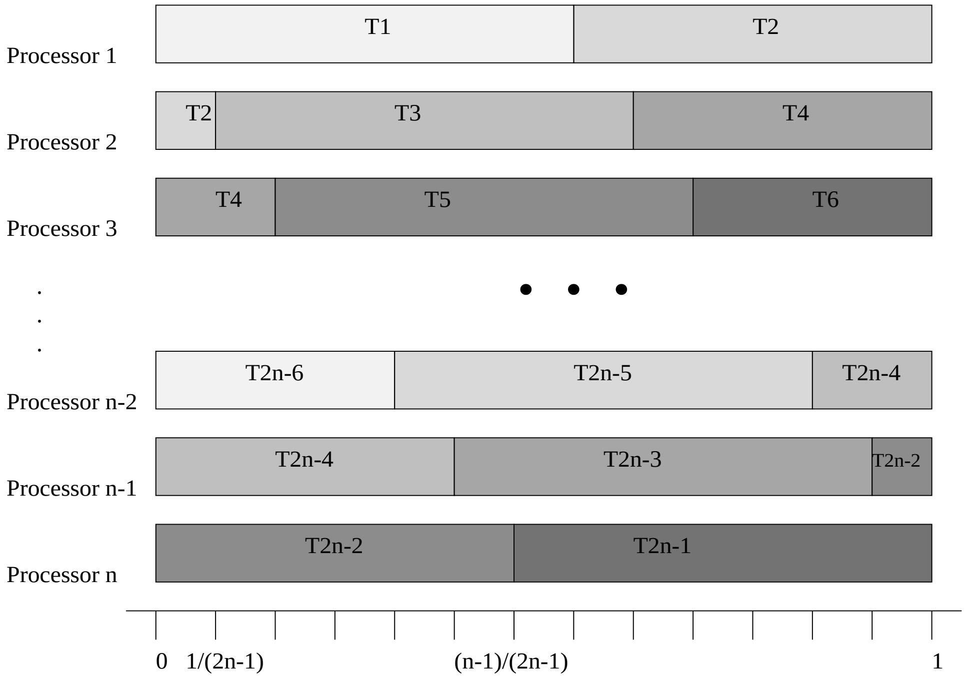

Figure 3: Upper Bound Example

Corollary 3.2 *There is no off-line non-migrative scheduling algorithm for n processors with performance ratio above*

$$\frac{\frac{n^2}{2n-1}}{n} = \frac{n}{2n-1}$$

4 The Lower Bound

To prove the lower bound we will present the NO-MIGRATION algorithm – a scheduling algorithm that does not use migration.

Denote $1 - \left(1 - \frac{1}{2n}\right)^n$ by $\rho(n)$. The scheduler NO-MIGRATION rdeceives as its input a task set and constructs a schedule that achieves at least $\rho(n)$ the value of the optimal migrative schedule. We will assume the existence of such a migrative schedule in order to prove that NO-MIGRATION indeed satisfies the ratio $\rho(n)$. However, a construction of such optimal scheduler is not needed and it is not used by NO-MIGRATION.

Before describing algorithm NO-MIGRATION we need one more definition.

Definition 4.1 *Given a schedule σ on n processors, define $Proj\,(\sigma) \subseteq R^+$ the* **projection** *of σ as follows: $t \in Proj\,(\sigma)$ iff there exist a task T, that according to σ executes at time t on any of the processors.*

The idea of NO-MIGRATION is the following. Start with the set of tasks S and construct a subset R_1 that is schedulable on a single processor such that the sum of the computation times of the tasks in R_1 is at least half the projection of an optimal migrative schedule of S. Delete the tasks of R_1 from S and repeat the same process, constructing a set R_2 with the same properties. Do this n times, producing a non-migrative schedule on n processors.

For clarity of exposition and proof, we assume the existence of two building blocks, EDF* and EDF-LST that construct the sets R_i, $i = 1, \dots, n$ with the desired properties. The details of these algorithms are described in the next section.

4.1 The No-Migration Algorithm

Algorithm Outline

INPUT: A set of tasks S.

1. $TASK\_SET \leftarrow S$
2. for $i = 1$ to n do
   (a) Construct a subset $R_i \subseteq TASK\_SET$ (using the EDF-LST algorithm and the construction in section 5.3).
   /* We will see in section 5 that R_i satisfies:
   - R_i is schedulable on one processor. and,
   - The sum of of R_i tasks' computation times is at least half of P_i, the projection of an optimal migrative schedule of $TASK\_SET$.

   */
   (b) $TASK\_SET \leftarrow TASK\_SET \setminus R_i$.
3. End For

End Algorithm

For each $i = 1, \dots, n$, the tasks in R_i can run on a single processor. Thus the tasks in $\cup_{i=1}^{n} R_i$ constitute a non-migrative schedule on n processors. For each $i = 1, \dots, n$, the computation time of all the tasks in R_i is at least half the projection of an optimal schedule on the tasks in $TASK\_SET$ during the ith iteration. The following theorem proves that the total computation time of the tasks in $\cup_{i=1}^{n} R_i$ is at least $\rho(n)$ of the total computation time in an optimal migrative schedule.

Notation: Denote by V the *value* of a migrative schedule of task set S on n processors. Denote by P_i, $i = 1, \dots, n$ the *projection* of an optimal migrative schedule on n processors of $TASK\_SET$ in iteration i of the No-Migration algorithm.

Lemma 4.1

For all $1 \leq i \leq n$,

$$\sum_{j=1}^{i} R_i \geq \frac{1}{2n} \sum_{j=1}^{i} \left(\frac{2n-1}{2n} \right)^{j-1} V$$

PROOF.

We prove this claim by induction.

For $i = 1$, $R_1 \geq \frac{1}{2n}\left(\frac{2n-1}{2n}\right)^0 V = \frac{1}{2n}V$. This is true since, by definition of the projection of a schedule, the length of the first projection, P_1, is not smaller than the busy time of any single processor. Hence $nP_1 \geq V$ and R_1's value is at least half of P_1's length . Therefore,

$$R_1 \geq \frac{P_1}{2} \geq \frac{1}{2n}V$$

Assume the correctness of the claim for i, we have to prove it for $i+1$:

In step $i+1$, the total length of tasks that still remain to be scheduled is $V - \sum_{j=1}^{i} R_j$. The length of the $i+1$st projection, P_{i+1}, is greater than the busy time of any single processor. Hence, $nP_{i+1} \geq (V - \sum_{j=1}^{i} R_j)$. The value of R_{i+1} is at least half of P_{i+1}'s length . Therefore,

$$R_{i+1} \geq \frac{P_{i+1}}{2} \geq \frac{1}{2n}\left(V - \sum_{j=1}^{i} R_j\right) \tag{1}$$

And,

$$\begin{aligned}
\sum_{j=1}^{i+1} R_j &= \sum_{j=1}^{i} R_j + R_{i+1} \\
&\geq \sum_{j=1}^{i} R_j + \frac{V - \sum_{j=1}^{i} R_j}{2n} && \text{(by inequality 1 above)} \\
&= \frac{2n\left(\sum_{j=1}^{i} R_j\right) + V - \sum_{j=1}^{i} R_j}{2n} \\
&= \frac{V + (2n-1)\sum_{j=1}^{i} R_j}{2n} \\
&= \frac{V}{2n} + \frac{(2n-1)}{2n}\sum_{j=1}^{i}\frac{1}{2n}\left(\frac{2n-1}{2n}\right)^{j-1} V && \text{(by the induction hypothesis)} \\
&= \frac{V}{2n} + \sum_{j=2}^{i+1}\frac{1}{2n}\left(\frac{2n-1}{2n}\right)^{j-1} V \\
&= \sum_{j=1}^{i+1}\frac{1}{2n}\left(\frac{2n-1}{2n}\right)^{j-1} V
\end{aligned}$$

which concludes the induction. $\square$

Theorem 4.2 *Given a set of tasks S there exists a non-migrative schedule for n processors obtaining a value of at least,*

$$\rho(n)V$$

where V denotes the value of an optimal non-migrative schedule. Furthermore, such a schedule is produced by No-Migration.

PROOF.

The schedule created by No-Migration assigns the tasks of R_i to processor i. Since each of the R_i's are schedulable on one processor we obtain a legal schedule on n processors that use no migration. Its total value is $\sum_{i=1}^{n} R_i$ which according to lemma 4.1 satisfies:

$$\begin{aligned}
\sum_{i=1}^{n} R_i &\geq \sum_{i=1}^{n} \frac{1}{2n}\left(\frac{2n-1}{2n}\right)^{j-1} V \\
&= \frac{1}{2n}\left(\frac{\left(1-\frac{1}{2n}\right)^n - 1}{\left(1-\frac{1}{2n}\right) - 1}\right) V \\
&= \left(1-\left(1-\frac{1}{2n}\right)^n\right) V
\end{aligned}$$

□

The following lemma present the asymptotic behavior (as the number of processors, n grows) of the above guarantee.

Lemma 4.3 *$\rho(n)$ is a monotonic decreasing function and*

$$\text{for all n} \quad \rho(n) \geq \lim_{n\to\infty} \rho(n) = 1 - \frac{1}{\sqrt{e}}$$

PROOF.

$\rho(n)$ is a monotonic decreasing function therefore its infimum is reached at the limit.

$$\lim \rho(n) = \lim 1 - \left(1 - \frac{1}{2n}\right)^n = 1 - \lim\left(1 + \frac{-\frac{1}{2}}{n}\right)^n = 1 - e^{-\frac{1}{2}} = 1 - \frac{1}{\sqrt{e}}$$

(We have made use of the equality: $\lim_{n\to\infty}\left(1+\frac{x}{n}\right)^n = e^x$.) □

We conclude that for each n the lower bound is greater then $1 - \frac{1}{\sqrt{e}} \sim 0.3935$.

Recall that the upper bound we constructed is $\frac{n}{2n-1}$ which converges to 0.5 as $n \to \infty$. In fact, both upper and lower bound sequences converge quite rapidly, as can be observed in table 4.1 and the graph in figure 4.

n	2	3	4	7	10	40	70	100	400	700	1000
lower	.4375	.4213	.4138	.4047	.4013	.3954	.3946	.3942	.3937	.3936	.3935
upper	.6667	.6000	.5714	.5385	.5263	.5063	.5036	.5042	.5006	.5004	.5003
ratio between bounds	.656	.702	.724	.752	.762	.781	.784	.782	.786	.787	.787

Table 4.1: Lower and upper bounds for different processor numbers

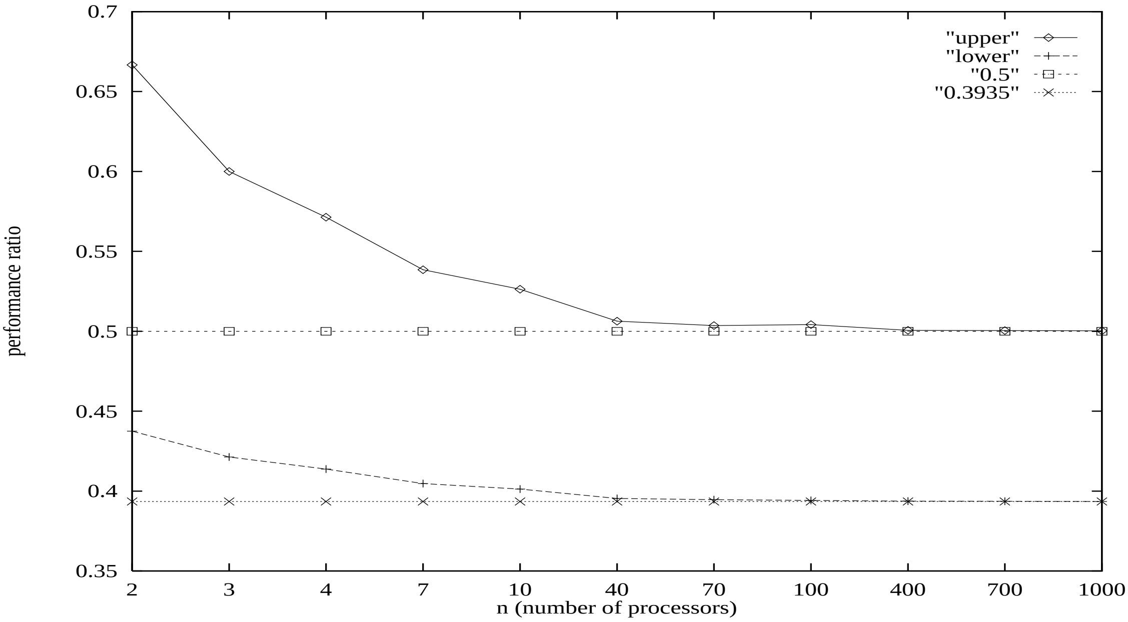

Figure 4: Graph of convergence speed

Theorem 4.4 *For any number of processors $n \geq 1$. Given a set of tasks S there exists a non-migrative schedule for n processors obtaining a value of at least,*

$$\left(1 - \frac{1}{\sqrt{e}}\right) V \sim 0.3935\ V$$

Where V denotes the value of an optimal non-migrative schedule for n processors. Furthermore, such a schedule is produced by No-Migration.

PROOF.

We use theorem 4.2 and lemma 4.3 above. $\square$

5 Building Uni-Processor Schedulable Subsets (the R_is)

In this section we show how to build the R_i subsets, i.e. subsets of the scheduled tasks whose value is at least half of the projection of any n processor schedule. We describe two uni-processor schedulers – EDF* and EDF-LST.

Theorem 5.1 *Given $Sched_n(S)$ a schedule of S on n processors, there exist a $R' \subseteq S$ schedulable (on a* **single** *processor) with total value of at least*

$$\frac{1}{2} Proj\left(Sched_n(S)\right)$$

The theorem will be proven in three steps as follows:

1. First, we introduce a preemptive uni-processor scheduling algorithm - EDF*. Note that this is essentially an on-line algorithm, but we use it here as part of an off-line scheduler.

2. EDF-LST is a uni-processor scheduling algorithm that uses EDF* as a subroutine. It schedules S on one processor (of course some tasks may not successfully complete).

   Using the schedule created above by EDF-LST we partition the projection of S into disjoint intervals. For each interval we have a (possibly overloaded) task subset scheduled in this interval.

3. For each such tasks subset, we construct a sub-subset with the property that the sum of its tasks' values is at least half the length of the corresponding interval and it is schedulable on a uni-processor system. The collection of all these sub-subsets is schedulable on a single processor and achieves value not less than half of the projection of $Sched_n(S)$.

5.1 The EDF* Algorithm

We start by introducing EDF* - a uni-processor preemptive scheduling algorithm. EDF* is used as subroutine in EDF-LST.

The EDF* algorithm works as follows: EDF* maintains a set of **accepted** tasks, and schedules these tasks according to the EDF (*earliest deadline first*) strategy [6]. Initially the set of accepted tasks is empty. Upon their release, "new" tasks may be added to the accepted set, provided that this set remains underloaded (that is, schedulable by EDF [6]).

Algorithm Outline

INPUT: A set of tasks S.

1. At the release time of some task T, if T has the minimal deadline among all currently accepted tasks we check whether the acceptance of T would not cause any of currently accepted tasks to miss its deadline. (A similar strategy is employed by D^{over}[10].) If there is no such danger then the new task T is accepted, the currently executing task is preempted and T starts to execute. Else T is not accepted, and the current task continues executing.

2. Schedule the accepted tasks according to EDF.

3. If (at task completion) **accepted** becomes empty, then the ready task with an earliest deadline becomes "accepted" and scheduled for execution.

End Algorithm.

Example 5.1 *Run Through* EDF*

	r	c	d
T_1	0	3	6
T_2	0	2	8
T_3	1	2	5
T_2	2	2	4

Table 5.1: Set of tasks for example 5.1

We will illustrate the scheduling of EDF* by the set of tasks presented in table 5.1. Initially, **accepted** is empty. At time 0, the first two tasks are released. Since T_1 has earlier deadline, **accepted** becomes $\{T_1\}$ and T_1 is scheduled. At time 1, T_3 is released, its deadline is earlier then T_1's deadline and both tasks can be scheduled to completion, hence T_3 preempts T_1 and it is added to **accepted**. At time 2, T_4 is released, it has a deadline which is smaller than all the tasks in **accepted** but can not be scheduled without preventing any of the other tasks in **accepted** from completion. Hence, T_3 continues until its completion at time 3, then T_2 resumes and completes at time 5, leaving **accepted** empty. At time 5, T_2 is added to the empty **accepted** set, it is scheduled until it completes at time 7. Note that T_4 was not scheduled at all.

5.2 The EDF-LST algorithm

In cases of underloaded task sets, EDF* would have been enough for our needs. In fact an underloaded case would mean that *all* tasks can be scheduled on *one* processor. Consequently, we can certainly expect that a task set schedulable on n processors will be un-schedulable (overloaded) when scheduled on one processor. For this reason, we introduce a scheduling algorithm called **EDF-LST** that is an "extension" of EDF* for overloaded cases.

5.2.1 LST: Latest Start Time

We define **LST** – **Latest Start Time**, an emergency point that may occur during task's execution: if a task reaches its **LST** it means that it has no more slack time. Hence, it must be scheduled immediately and executed continuously until its completion on its deadline otherwise it will miss its deadline. Formally,

Definition 5.2

At any given moment t for a task T:

$$\mathbf{LST}\,(t,T) = deadline\,(T) - remaining\_computation\_time\,(T)$$

LST-exception: when (and if) a task that is not currently executing reaches an LST point it raises the LST-exception.

The EDF-LST scheduling algorithm schedules according to the EDF* algorithm as long as no LST-exception is raised. At some point an LST-exception may be raised, either by previously preempted tasks, some task that EDF* did not "accept" or at the release time of tasks with no slack time. As soon as an LST-exception is raised EDF-LST might diverge from EDF* by scheduling tasks at their LST points. These tasks are called *LST tasks* while the tasks scheduled according to EDF* are called *EDF tasks*.

When there are no more LST tasks to schedule, EDF-LST returns to schedule according to EDF* and so forth until there are no more tasks to schedule. The precise conditions for LST tasks or EDF tasks will be seen in the next subsection.

This schedule creates alternating sets of EDF tasks, and LST tasks covering some contiguous time interval bordered by *idle* periods. A time period is said to be *idle* if at all its points all tasks either completed, missed their deadlines or were not released yet.

We will prove that we can select from the tasks in a time interval bordered by idle periods, an underloaded (i.e., schedulable) subset of tasks whose cumulative value is at least half the length of that time interval. Combining this for all intervals, we build the desired schedule.

5.2.2 EDF-LST

The pseudo code of the algorithm is built from one main function and two exception handlers (the subroutine that implements EDF* is not shown here).

At each moment, the algorithm can be in one of two phases: **edf_phase** or **lst_phase**. In the first, tasks are scheduled according to EDF*. In the later, scheduling decisions are made in the LST-exception handler as a result of raising such an exception.

The algorithm have cycles. Each cycle starts with an edf_phase followed by an optional lst_phase. While in edf_phase the algorithm maintains a collection of "accepted" tasks (exactly in the same manner as it is done by EDF*). If a task T raises an LST exception it will be scheduled only if its deadline is far enough [1] is in the future. Otherwise, T is aborted. Scheduling T means switching to lst_phase. T task will schedule to completion unless another lst exception will be raised by another task with an even "further enough" deadline [2] and so on.

This chain of exceptions/abortions ends only if some task is left to complete. This marks the end of a cycle, in which case the algorithm switches back to edf_phase (the "accepted" set is the ready task with earliest deadline).

Why do we use the above criteria for "far enough" deadline? The main idea is to identify in each cycle a collection of "executing units" (the set of all the accepted tasks is one unit (the first) and the tasks scheduled by their lst-exception each one as a subsequent unit by its own) that covers an interval.

Two adjacent units may interfere with each other (in the sense that scheduling one may prevent

[1] The meaning of far enough here is as follows, we consider the time interval in which the currently accepted tasks will execute to completion (if not interrupted). We look at all the tasks that will raise an lst exceptions during that interval and their deadlines are beyond the end of that interval. T's deadline must be not smaller than all these deadlines.

[2] The meaning of further enough here is as follows, we consider the time interval in which the currently scheduled task will execute to completion (if not interrupted). We look at all the tasks that will raise an lst exceptions during that interval and their deadlines are beyond the end of that interval. T's deadline must be not smaller than all these deadlines.

from scheduling the other) but removing any one unit will enable its (one or) two adjacent units to complete. This mean that eliminating the collection of all odd numbered units enable the collection of all even numbered units to complete and vis versa. These two collections covers the entire interval hence at bigger of the two covers at least half of the interval which is what we looked for.

The EDF-LST Algorithm

INPUT: A set of tasks S.

In the following algorithm, **accepted** denotes the set of accepted tasks as defined in the definition of EDF*.

- **main body** /* main body of the algorithm */

  1. if S is empty, end of execution. /* strange, no tasks were presented */
  2. /* otherwise, there are tasks to execute */
     /* in the beginning, starting in edf_phase */
     $phase \leftarrow edf\_phase$
  3. /* main loop of the algorithm */
     loop (forever) {
     - as long as **phase** equals **edf_phase**, schedule according to EDF* .
       /* tasks scheduled in this phase are called EDF tasks */
     - otherwise, **phase** equals **lst_pase** in which case scheduling decisions are made in the LST_exception handler (see below).

  }

- **End of main body**

- **LST_Exception handler**
  /* a task T raises an LST-exception, say at time t */

  - **if** ($phase = edf\_phase$) then
    /* scheduling is currently done according to EDF* */
    1. Let $endof$ (**accepted**) be the time that tasks of **accepted** are expected to complete if scheduled without any interruption.
       Consider the set of all tasks that will raise an LST-exception in the time interval $[t, endof$ (**accepted**)] provided the current tasks of **accepted** would continue without interruption, and whose deadline is later than endof(**accepted**) .
       ¿From this set of tasks, choose the task with the latest deadline. Denote its deadline by d_{max}.
    2. If $d(T) \geq d_{max}$ then
       /* $d(T)$ is indeed far enough */
       /* switch from edf_phase to lst_phase */
       /* end of scheduling according to EDF* start to schedule according to LST exceptions */
       $phase \leftarrow lst\_phase$
       • $i \leftarrow 1$ /* index of LST_i */

- Label T as $\mathbf{LST_1}$
- /* Let $raised\,(\mathbf{LST_1}) = t$
(the time that task $\mathbf{LST_1}$ raised its LST-exception) */
- /* Let $endof\,(\mathbf{LST_1}) = d(LST_1)$ (the time that LST_1 is expected to complete if scheduled without any interruption) */
- Start to execute $\mathbf{LST_1}$

else
/* $d(T)$ is not far enough */
- **abort** T (continue in **edf_phase**, scheduling according to EDF*)

– **else**
/* some T raised an **LST**-exception, while some LST_i is executing */

1. Consider the set of all tasks that will raise an LST-exception in the time interval $[raised\,(\mathbf{LST_i})\,, endof\,(\mathbf{LST_i})]$, provided that LST_i would continue without interruption, (T will be aborted), and whose deadline is later than endof($\mathbf{LST_i}$) .
From this set of tasks, choose the task with the latest deadline. Denote its deadline by d_{max}.
2. If $d(T) \geq d_{max}$ then
/* $d(T)$ is far enough */
/* switch from the current LST task to T */
   - $i \leftarrow i + 1$
   - **Label** T as $\mathbf{LST_{i+1}}$
   - /* Let $raised\,(\mathbf{LST_{i+1}}) = t$
   (the time that task $\mathbf{LST_{i+1}}$ raised its LST-exception) */
   - /* Let $endof\,(\mathbf{LST_{i+1}}) = d(LST_{i+1})$
   (the time that LST_{i+1} is expected to complete if scheduled without any interruption) */
   - start to execute $\mathbf{LST_{i+1}}$

   else
   /* $d(T)$ is not far enough */
   - **abort** T (continue with $\mathbf{LST_i}$)

– **End if**

End LST_Exception handler

- **Task_Completion handler**
/* The currently executing task T completes, say at time t */
if $(phase = edf\_phase)$ then
/* T was scheduled by EDF* */
task completion is handled according to EDF* (schedule the next accepted task or if empty the next ready task with earliest deadline)
else
/* T is LST_j for some j */
/* switch phases , return to EDF* scheduling */
$phase \leftarrow edf\_phase$
start scheduling according to EDF*
(**accepted** is set to the next ready task with earliest deadline)

End Task Completion handler

End Algorithm.

Note that the collection of tasks scheduled by EDF-LST may be an overloaded task set, that is, some of tasks do not execute to completion. This can happen if tasks that where scheduled by EDF* are preempted and never resumed, or tasks that raise LST-exception are not selected to execute or, even if selected, may be aborted when the next LST task is scheduled.

See fig. 5 for a schematic example of an EDF-LST schedule.

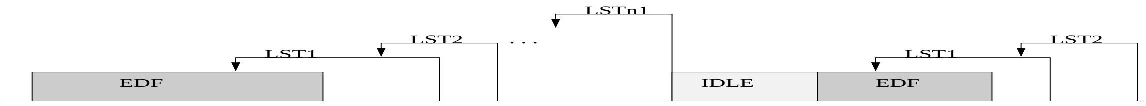

Figure 5: A schematic of an EDF-LST schedule

In the next section we show how to select an underloaded subset of tasks with "big enough" value, i.e., the sum of its tasks' values is no less than the projection of an optimal migrative schedule.

5.3 Selecting an Underloaded Subset from the EDF-LST Schedule

This section completes the last necessary step to prove our lower bound. We study the schedule created by EDF-LST. First we suitably partition the tasks in this schedule. Then, we use this partition to construct an underloaded subset with "big enough" value.

5.3.1 Partitioning EDF-LST's Schedule

The schedule of EDF-LST naturally splits the tasks of S into two collections. The first is composed of tasks that *appear* in the schedule. These are tasks that were scheduled for execution for at least one time interval and eventually either completed or aborted. The second includes the rest of the tasks – those that were never scheduled for execution. These are tasks that were not scheduled by EDF* and when their LST was raised they were aborted.

The result of the EDF-LST algorithm is a schedule of tasks that have the following structure:

$$EDF^1,\, LST^1, EDF^2,\, LST^2, ..., EDF^i,\, LST^i, ...$$

Where EDF^i is a set of tasks scheduled according to EDF* and LST^i is a set of tasks scheduled following an LST-exception. The individual tasks of EDF^i and LST^i are denoted by: $EDF^i_1, EDF^i_2, EDF^i_3$, ..., and $LST^i_1, LST^i_2, LST^i_3$, ..., . respectively.

We will now partition this collection into two sets, each of which schedulable on a single processor, based upon the above structure.

For every i define the following subsets of S:

- α_i : Tasks that **last appear**[3] as one of LST^i tasks but with *odd* indexes. I.e. $LST^i_{\mathbf{1}}$, $LST^i_{\mathbf{3}}, LST^i_{\mathbf{5}}$, ...

[3] "*Last appear*" means the last time it appeared, i.e., the last time it was executed. Note that a task may be preempted and rescheduled many times. However, since each task may "last appear" at most once we obtain a partitioning into **disjoint** task.

- β_i : Tasks that **last appear** as part of EDF^i and *did not complete* and tasks that **last appear** as one of LST^i tasks but with *even* indexes. I.e. $LST^i_{\mathbf{2}}$, $LST^i_{\mathbf{4}}, LST^i_{\mathbf{6}}$, ...

- γ : Tasks that **last appear** as any of EDF^is and *completed* successfully.

5.3.2 Building an Underloaded Task Set

Lemma 5.2 *Each of the sets: γ, α_i, β_i (for all i) is an underloaded task set (i.e., can be scheduled successfully on one processor).*

PROOF.

γ is obviously underloaded since all its tasks were successfully completed by EDF-LST.

We will show that $\forall i$, α_i and β_i are underloaded. If α_i is empty so is β_i (because empty α_i implies that LST^i is empty, if this is the case then all tasks of EDF^i were scheduled to completion. Hence β_i is empty) and we are done. Otherwise, suppose we eliminate the tasks of α_i from the schedule created by EDF-LST. We will show how all the tasks of β_i can now successfully complete by using only the time intervals thus vacated.

• *Eliminating LST^i_1 will enable all the tasks of EDF^i to complete.*

This can be seen by the following argument:

1. If there is no LST^i_2 than LST^i_1 completes running by its deadline, according to the EDF-LST schedule. EDF-LST schedules LST^i_1 only if its deadline occurs after $endof(EDF^i)$.

2. If LST^i_2 exists then the LST-exception of LST^i_2 occurred after $endof(EDF^i)$ (otherwise, LST^i_1 would not have been scheduled).

Concluding that in both cases the time freed by LST^i_1 is enough for all EDF^i tasks.

• *Eliminating, LST^i_3 will enable LST^i_2 to complete.*

The reasoning is similar. If there is no LST^i_4 than LST^i_3 completed by EDF-LST. EDF-LST schedules LST^i_3 only if its deadline is later than $d(LST^i_2)$.

If there exists LST^i_4 then the LST of LST^i_4 occurred after $d(LST^i_2)$ (otherwise, LST^i_3 would not have been scheduled).

Concluding that in both cases the time freed by LST^i_3 is enough for LST^i_2.

• *In general, eliminating α_i enables all the tasks of β_i to complete.*

This can be easily verified by induction in a similar manner to the above two cases.

Conclusion: β_i is underloaded.

A similar argument, though somewhat simpler since the tasks of EDF^i do not matter here, shows that eliminating β_i enables all the tasks of α_i to complete.

Conclusion: α_i is underloaded.
□

Notation 5.3

- Let σ be any schedule (migrative or non-migrative) of S on n processors. Denote the projection of σ by $PROJ$.
- Denote the union of the non-idle periods of EDF-LST by $BUSY$.

For all i define,

- max_i: the set with higher total value among α_i and β_i (if equal value, choose arbitrarily).
- min_i: the set with lower total value among α_i and β_i.

For ease of notation we will henceforth denote the *values* of α_i, β_i and max_i also by α_i, β_i and max_i, resp. Similarly we will use $PROJ$ and $BUSY$ to also denote their total lengths.

Lemma 5.3 $\gamma \bigcup \{\cup_i max_i\}$ *is an underloaded task set.*

PROOF.

From the proof of lemma 5.2 above, we see that expanding the tasks of max_i to the time vacated by eliminating min_i (and only this time) suffices for the completion of all the tasks of max_i. Hence, eliminating $\cup_i min_i$ will enables $\cup_i max_i$ to complete without disturbing any of the γ tasks. $\square$

Lemma 5.4 $\gamma + 2\sum_i max_i \geq BUSY$

PROOF.

Only tasks of γ, α_i and β_i appears in the schedule of EDF-LST. Hence,

$$\gamma \bigcup \cup_i \{\alpha_i \cup \beta_i\} \geq BUSY$$

Since $2max_i \geq (\alpha_i + \beta_i)$ we get the desired result. $\square$

Lemma 5.5 $BUSY + \gamma \geq PROJ$

(recall that $BUSY$ refers to a uni-processor schedule created by EDF-LST *while $PROJ$ refers to an n processors schedule)*

PROOF.

It is obvious that $BUSY + (PROJ \setminus BUSY) \geq PROJ$. Consider a point t in $PROJ \setminus BUSY$ (i.e., an idle point). Let T be a task that executes at time $t \leq d(T)$ (according to σ).

If T was aborted by EDF-LST then EDF-LST has no idle time between $r(T)$ and $d(T)$, otherwise T would occupy the idle periods, making them busy.

Hence, T completed by EDF-LST. Moreover, it completed before t (otherwise t won't be an idle point) meaning it completed as an EDF task[4], that is $T \in \gamma$.

[4]Recall that an LST task completes exactly on its deadline.

We conclude, that for all the tasks T_i which executed (even partially) at idle periods of EDF-LST are tasks of γ i.e, $PROJ - BUSY \leq \gamma$ which proves the lemma. $\Box$

Corollary 5.6 $\gamma + \sum_i max_i \geq \frac{PROJ}{2}$

PROOF.

Lemma 5.4 shows that,

$$\gamma + 2\sum_i max_i + \gamma \geq BUSY + \gamma$$

Using lemma 5.5 we get,

$$2\left(\gamma + \sum_i max_i\right) \geq PROJ$$

$\Box$

This concludes the proof of theorem 5.1 and assures us that there is indeed a non-migrative schedule on n processors whose value is no less than $1 - (1 - \frac{1}{2n})^n$ of the optimal migrative schedule.

5.4 A Run Through No-Migration Algorithm

Let us run through No-Migration executed set of tasks presented in on the performance ratio example (example 2.2). This set of tasks has three identical tasks, T_1, T_2, and T_3. They all have the same parameters: start time 0, deadline 1 and computation time $\frac{2}{3}$.

In the first iteration, we start by utilizing EDF-LST which actually start by utilizing EDF*. Since all tasks have the same start time, EDF* chooses any one to start with, say T_1. Therefore, $EDF_1^1 = T_1$. At time $t = \frac{1}{3}$, there is an LST-exception for both T_2 and T_3. Suppose the exception of T_2 is treated first. Since its deadline is greater than the time that EDF_1^1 is expected to complete, we preempt T_1 and schedule T_2. Hence, $LST_1^1 = T_2$. Immediately, EDF-LST treats the exception generated by T_3 leading to its abortion. At time 2/3, T_1 raises an LST-exception which leads to its abortion.

In this example, α_1 is $\{T_2\}$, β_1 is $\{T_1\}$ and γ is empty (so far) Note that T_3 is missing from all these sets. In this case, α and β are of equal size so R_1 can be either of them, suppose the later is chosen.

Now, to the second iteration in No-Migration, T_1 is removed from the initial task collection and proceed as above with the remaining tasks (T_2 and T_3). Suppose T_2 is scheduled initially, then at time 1/3 it will be preempted by an exception of T_3 which will complete. Hence, α_2 is $\{T_2\}$ and β_2 is $\{T_3\}$ (γ is empty) . Since, these are equal R_2 could be any of them, suppose the later is chosen. Which concludes the second and last iteration of No-Migration.

Thus, the schedule generated by No-Migration is: T_1 gets scheduled on the first processor and T_3 gets scheduled on the second processor. T_3 is not scheduled at all.

In this example the No-Migration algorithm, in fact, chose an optimal non-migrative schedule.

6 Conclusion and Open Problems

We gave an upper and lower bound on the performance ratio between a non-migrative and migrative clairvoyant schedulers in real-time systems. While the bounds are close there is still a gap between them. We believe that the upper bound is the real bound.

An important open problem is finding a competitive on-line scheduler for n processors in a real-time system with the uniform density value measure. The results in this paper guarantee that whether such a competitive on-line scheduler is migrative or non-migrative, it nevertheless implies the existence of *both* a migrative and non-migrative competitive scheduler.

References

[1] A. Bar-Noy, R. Canetti, S. Kutten, Y. Mansour, and B. Schieber. Bandwidth allocation with preemption. In *Proc. 27th ACM Symp. on Theory of Computing*, pages 616–625, 1995.

[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 real-time task scheduling. *The Journal of Real Time Systems*, 4(2):124–144, 1992.

[4] A. Burns. Scheduling hard real-time systems: a review. *Software Engineering Journal*, 6(3):116–128, 1991.

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

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

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

[8] E. G. Coffman Jr. and M. R. Garey. Preemptive vs. nonpreemptive two processor scheduling. *Journal of the ACM*, 40(5):991–1018, 1993.

[9] G. Koren and D. Shasha. Moca: A multiprocessor on-line competitive algorithm for real-time system scheduling. *Theoretical Computer Science*, 128(1&2):75–97, 1994.

[10] G. Koren and D. Shasha. D-over: An optimal on-line scheduling algorithm for over-loaded real-time systems. *SIAM J. Computing*, 24(2):318–339, 1995.

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

[12] J. D. Ullman. Complexity of sequencing problems. In Jr. E. G. Coffman, editor, *Computer and Job Scheduling Theory.* Wiley, 1976.