To appear in *Proceedings of the IEEE Real-Time Systems Symposium*, December 3–6, 2001, London, England

Static-priority scheduling on multiprocessors*

Björn Andersson† Sanjoy Baruah‡ Jan Jonsson†

Abstract

The preemptive scheduling of systems of periodic tasks on a platform comprised of several identical processors is considered. A scheduling algorithm is proposed for static-priority scheduling of such systems; this algorithm is a simple extension of the uniprocessor rate-monotonic scheduling algorithm. It is proven that this algorithm successfully schedules any periodic task system with a worst-case utilization no more than a third the capacity of the multiprocessor platform. It is also shown that no static-priority multiprocessor scheduling algorithm (partitioned or global) can guarantee schedulability for a periodic task set with a utilization higher than one half the capacity of the multiprocessor platform.

1 Introduction

Over the years, the preemptive periodic task model [16, 15, 4] has proven remarkably useful for the modelling of recurring tasks that occur in hard-real-time computer application systems. Accordingly, much effort has been devoted to the development of a comprehensive theory dealing with the scheduling of systems comprised of such independent periodic real-time tasks. Particularly in the uniprocessor context — in environments in which all hard-real-time jobs generated by all the periodic tasks that comprise the hard-real-time application system must execute on a single shared processor — there now exists a wide body of results (necessary and sufficient feasibility tests, optimal scheduling algorithms, efficient implementations of these algorithms, etc.) for systems modelled as a collection of independent preemptive periodic real-time tasks. Some of these results have been extended to the multiprocessor context — environments in which there are several identical processors available upon which the real-time jobs may be executed.

The periodic task model. In the periodic model of hard real-time tasks, a task $\tau_i = (C_i, T_i)$ is characterized by two parameters — an execution requirement C_i and a period T_i — with the interpretation that the task generates a job at each integer multiple of T_i, and each such job has an execution requirement of C_i execution units, and must complete by a deadline equal to the next integer multiple of T_i. A periodic task system consists of several such periodic tasks that are to execute on a specified processor architecture. We assume that each job is independent in the sense that it does not interact in any manner (accessing shared data, exchanging messages, etc.) with other jobs of the same or another task. We also assume that the model allows for job *preemption*; i.e., a job executing on a processor may be preempted prior to completing execution, and its execution may be resumed later, at no cost or penalty.

In this paper, we will study the scheduling of systems of periodic tasks. Let $\tau = \{\tau_1, \tau_2, \ldots, \tau_n\}$ denote a *periodic task system*, in which each periodic task $\tau_i = (C_i, T_i)$ is characterized by its execution requirement and its period. For each task τ_i, define its *utilization* U_i to be the ratio of τ_i's execution requirement to its period: $U_i \stackrel{\text{def}}{=} C_i/T_i$. We define the utilization $U(\tau)$ of periodic task system τ to be the sum of the utilizations of all tasks in τ: $U(\tau) \stackrel{\text{def}}{=} \sum_{\tau_i \in \tau} U_i$. Without loss of generality, we assume that $T_i \leq T_{i+1}$ for all i, $1 \leq i < n$; i.e., the tasks are indexed according to period.

Dynamic and static priorities. *Run-time scheduling* is the process of determining, during the execution of a real-time application system, which job[s] should be executed at each instant in time. Run-time scheduling algorithms are typically implemented as follows: at each time instant, assign a **priority** to each active[1] job, and allocate the available processors to the highest-priority jobs.

With respect to certain run-time scheduling algorithms, it is possible that some tasks τ_i and τ_j both have active jobs at times t_1 and t_2 such that at time t_1, τ_i's job has higher priority than τ_j's while at time t_2, τ_j's job has higher priority than τ_i's. Run-time scheduling algorithms that permit such "switching" of the order of priorities between tasks are known as **dynamic** priority algorithms.

By contrast, **static** priority algorithms satisfy the prop-

*Supported in part by the National Science Foundation (Grant Nos. CCR-9704206, CCR-9972105, CCR-9988327, and ITR-0082866) and by the Swedish Foundation for Strategic Research via the national Swedish Real-Time Systems research initiative ARTES.

†Department of Computer Engineering, Chalmers University of Technology, SE–412 96 Göteborg, Sweden – {ba,janjo}@ce.chalmers.se

‡Department of Computer Science, University of North Carolina, Chapel Hill, NC 27599, USA – baruah@cs.unc.edu

[1] Informally, a job becomes *active* at its ready time, and remains so until it has executed for an amount of time equal to its execution requirement, or until its deadline has elapsed.

erty that for every pair of tasks τ_i and τ_j, whenever τ_i and τ_j both have active jobs, it is always the case that the same task's jobs have priority. An example of a static-priority scheduling algorithm is the *rate-monotonic scheduling algorithm* [15], which assigns each task a priority inversely proportional to its period — the smaller the period, the higher the priority, with ties broken arbitrarily but in a consistent manner: if τ_i and τ_j have equal periods and τ_i's job is given priority over τ_j's job once, then all of τ_i's jobs are given priority over τ_j's jobs.

It is beyond the scope of this document to compare and contrast the relative advantages and disadvantages of static-priority versus dynamic-priority scheduling. Observe that in the context of *static-priority* scheduling, the run-time scheduling problem — determining during run-time which jobs should execute at each instant in time — is exactly equivalent to the problem of assigning priorities to the tasks in the system, since once the priorities are assigned run-time scheduling consists of simply choosing the currently active jobs with the highest priorities.

A hard-real-time task system is defined to be *static-priority feasible* if it can be scheduled by a static-priority run-time scheduler in such a manner that all jobs will always complete by their deadlines under all permissible circumstances. Given the specifications for a system of hard-real-time tasks, *static-priority feasibility analysis* is the process of determining whether the system is static-priority feasible.

Partitioned versus global scheduling. In this paper, we will study the static-priority scheduling of systems of periodic tasks on m identical processors, $m \geq 2$. To that end, there are two distinct approaches available.

- In **partitioned** scheduling, all jobs generated by a task are required to execute on the *same* processor.
- In **global** scheduling, *task migration* is permitted. That is, we do not require that all jobs of a task execute on the same processor; rather, we permit different jobs to execute on different processors. In addition, *job migration* is also permitted — a job that has been preempted on a particular processor may resume execution on the same or a different processor. We assume that there is no penalty associated with either task or job migration. However, *job-level parallelism* is expressly forbidden; i.e., it is not permitted that more than one processor be executing a job at any given instant in time.

In the partitioned approach, static-priority scheduling requires that (i) the set of tasks τ be partitioned among the m available processors, and (ii) a total order be defined among the tasks within each partition. Then at each instant during run-time, the active job generated by the highest-priority task within each partition is chosen for execution on the corresponding processor; if there is no active job in a partition, then the corresponding processor is left idle. In the global approach, on the other hand, we must define a total order among all the tasks in τ, and at each instant during run-time choose for execution the m highest-priority active jobs (with some processors remaining idle if there are fewer than m active jobs).

It has been proven by Leung and Whitehead [14] that the partitioned and global approaches to static-priority scheduling on multiprocessors are *incomparable*, in the sense that (i) there are task systems that are feasible on m processors under the partitioned approach but for which no priority assignment exists which would cause all jobs of all tasks to meet their deadlines under global scheduling on m processors; and (ii) there are task systems that are feasible on m processors under the global approach, but which cannot be partitioned into m distinct subsets such that each individual partition is uniprocessor static-priority feasible. This observation provides a very strong motivation to study both the partitioned and global approaches to static-priority multiprocessor scheduling, since neither approach is strictly better than the other.

There is however, one important difference between the partitioned and global approaches. While the partitioned approach can rely on well-known optimal uniprocessor priority-assignment schemes, it is not clear as to what priority-assignment scheme should be used for the global approach. The following example by Dhall [10] demonstrates that traditional uniprocessor priority-assignment schemes do not work well for global multiprocessor scheduling. Consider a periodic task set with task priorities assigned in inverse proportion to their periods (i.e., rate monotonic scheduling):

$$\begin{aligned}\tau \stackrel{\text{def}}{=} & \{\tau_1 = (2\epsilon, 1), \tau_2 = (2\epsilon, 1), \ldots, \\ & \tau_m = (2\epsilon, 1), \tau_{m+1} = (1, 1+\epsilon)\}\end{aligned}$$

to be scheduled on a platform of m identical unit-speed processors. In this case, τ_{m+1} will have the lowest priority. When τ_{m+1} arrives at the same time as all higher-priority tasks, it is scheduled at time 2ϵ after it has arrived and will therefore miss its deadline. This example shows that, as $\epsilon \to 0$, the utilization of the task set becomes $U = 1$ no matter how many processors are used. Thus, with rate-monotonic scheduling on a multiprocessor, it is possible to find a task set that is unschedulable although it consumes only an arbitrarily small fraction of the capacity of the multiprocessor platform. In the following, we will refer to this phenomenon as *Dhall's effect*.

This research. The partitioned approach to static-priority multiprocessor scheduling has been extensively studied (see, for example, [13, 18, 17]). In this paper, we present a global static-priority scheduling algorithm for scheduling

systems of periodic tasks. We prove that this algorithm successfully schedules any periodic task system τ with utilization $U(\tau) \leq m^2/(3m-2)$ on m identical processors — as $m \to \infty$, this bound approaches $m/3$ from above; hence, it follows that our algorithm successfully schedules any periodic task system with cumulative utilization $\leq m/3$ on m identical processors (and consequently also avoids Dhall's effect). We consider our proof of this result to be interesting in its own right, in that we exploit an interesting result of Phillips *et al.* [19] (Theorem 1 below) that bounds from below the amount of execution that must be performed by any multiprocessor work-conserving scheduling algorithm; we expect that this result will prove useful for determining other useful properties of multiprocessor systems.

Organization of this paper. The remainder of this paper is organized as follows. In Section 2, we briefly describe two major results that we will be using in the remainder of this paper. In Section 3, we present Algorithm **RM-US[m/(3m-2)]**, our static-priority multiprocessor algorithm for scheduling arbitrary periodic task systems, and prove that Algorithm **RM-US[m/(3m-2)]** successfully schedules any periodic task system with utilization $\leq m^2/(3m-2)$ on m identical processors. In Section 4, we assess the performance of Algorithm **RM-US[m/(3m-2)]** using theoretical and experimental analysis. We conclude in Section 5 with a brief summary of the results contained in this paper.

2 Results we will use

Some very interesting and important results in real-time multiprocessor scheduling theory were obtained in the mid 1990's. We will make use of two of these results in this paper; these two results are briefly described below.

Resource augmentation. It has previously been shown [7, 6, 5] that on-line real-time scheduling algorithms tend to perform extremely poorly under overloaded conditions. Phillips, Stein, Torng, and Wein [19] explored the use of *resource-augmentation* techniques for the on-line scheduling of real-time jobs[2]; the goal was to determine whether an on-line algorithm, if provided with faster processors than those available to a clairvoyant algorithm, could perform better than is implied by the bounds derived in [7, 6, 5]. Although we are not studying on-line scheduling in this paper — all the parameters of all the periodic tasks are assumed a priori known — it nevertheless turns out that a particular result from [19] will prove very useful to us in our study of static-priority multiprocessor scheduling. We present this result below.

[2]Resource augmentation as a technique for improving the performance on on-line scheduling algorithms was formally proposed by Kalyanasundaram and Pruhs [12].

The focus of [19] was the scheduling of individual jobs, and not periodic tasks. Accordingly, let us define a **job** $J_j = (r_j, e_j, d_j)$ as being characterized by an arrival time r_j, an execution requirement e_j, and a deadline d_j, with the interpretation that this job needs to execute for e_j units over the interval $[r_j, d_j)$. (Thus, the periodic task $\tau_i = (C_i, T_i)$ generates an infinite sequence of jobs with parameters $(k \cdot T_i, C_i, (k+1) \cdot T_i)$, $k = 0, 1, 2, \ldots$; in the remainder of this paper, we will often use the symbol τ itself to denote the infinite set of jobs generated by the tasks in periodic task system τ.)

Let I denote any set of jobs. For any algorithm A and time instant $t \geq 0$, let $W(A, m, s, I, t)$ denote the amount of work done by algorithm A on jobs of I over the interval $[0, t)$, while executing on m processors of speed s each. A **work-conserving** scheduling algorithm is one that never idles a processor while there is some active job awaiting execution.

Theorem 1 (Phillips et al.) For any set of jobs I, any time-instant $t \geq 0$, any work-conserving algorithm A, and any algorithm A', it is the case that

$$W(A, m, (2 - \frac{1}{m}) \cdot s, I, t) \geq W(A', m, s, I, t). \quad (1)$$

■

That is, an m-processor work-conserving algorithm completes at least as much execution as any other algorithm, if provided processors that are $(2 - 1/m)$ times as fast.

Predictable scheduling algorithms. Ha and Liu [11] have studied the issue of predictability in the multiprocessor scheduling of real-time systems from the following perspective.

Definition 1 (Predictability) Let A denote a scheduling algorithm, and $I = \{J_1, J_2, \ldots, J_n\}$ any set of n jobs, $J_j = (r_j, e_j, d_j)$. Let f_j denote the time at which job J_j completes execution when I is scheduled by algorithm A.

Now, consider any set $I' = \{J'_1, J'_2, \ldots, J'_n\}$ of n jobs obtained from I as follows. Job J'_j has an arrival time r_j, an execution requirement $e'_j \leq e_j$, and a deadline d_j (i.e., job J'_j has the same arrival time and deadline as J_j, and an execution requirement no larger than J_j's). Let f'_j denote the time at which job J_j completes execution when I is scheduled using algorithm A. Scheduling algorithm A is said to be **predictable** if and only if for any set of jobs I and for any such I' obtained from I, it is the case that $f'_j \leq f_j$ for all j.

■

Informally, Definition 1 recognizes the fact that the specified execution-requirement parameters of jobs are typically

only *upper bounds* on the actual execution-requirements during run-time, rather than the exact values. For a predictable scheduling algorithm, one may determine an upper bound on the completion-times of jobs by analyzing the situation under the assumption that each job executes for an amount equal to the upper bound on its execution requirement; it is guaranteed that the actual completion time of jobs will be no later than this determined value.

Since a periodic task system generates a set of jobs, Definition 1 may be extended in a straightforward manner to algorithms for scheduling periodic task systems: an algorithm for scheduling periodic task systems is predictable iff for any periodic task systems $\tau = \{\tau_1, \tau_2, \ldots, \tau_n\}$ it is the case that the completion time of each job when every job of τ_i has an execution requirement exactly equal to C_i is an upper bound on the completion time of that job when every job of τ_i has an execution requirement of at most C_i, for all $i,\ 1 \leq i \leq n$.

Ha and Liu define a scheduling algorithm to be **priority driven** if and only if it satisfies the condition that *for every pair of jobs J_i and J_j, if J_i has higher priority than J_j at some instant in time, then J_i* always *has higher priority than J_j*. Notice that any global static-priority algorithm for scheduling periodic tasks satisfies this condition, and is hence priority-driven. However, the converse is not true in that not all algorithms for scheduling periodic tasks that meet the definition of priority-driven are global static-priority algorithms (e.g., notice that the earliest deadline first scheduling algorithm, which schedules at each instant the currently active job whose deadline is the smallest, is a priority-driven algorithm, but is not a static-priority algorithm).

The result from the work of Ha and Liu [11] that we will be using can be stated as follows.

Theorem 2 (Ha and Liu) Any priority-driven scheduling algorithm is predictable.

■

3 Algorithm RM-US[m/(3m-2)]

We now present Algorithm **RM-US[m/(3m-2)]**, a static-priority global scheduling algorithm for scheduling periodic task systems, and derive a utilization-based sufficient feasibility condition for Algorithm **RM-US[m/(3m-2)]**; in particular, we will prove that any task system τ satisfying $U(\tau) \leq m^2/(3m-2)$ will be scheduled to meet all deadlines on m unit-speed processors by Algorithm **RM-US[m/(3m-2)]**. This is how we will proceed. In Section 3.1, we will consider a restricted category of periodic task systems, which we call "light" systems; we will prove that the multiprocessor **rate-monotonic** scheduling algorithm (we will henceforth refer to the multiprocessor rate-monotonic algorithm as Algorithm **RM**), which is a global static-priority algorithm that assigns tasks priorities in inverse proportion to their periods, will successfully schedule any light system. Then in Section 3.2, we extend the results concerning light systems to arbitrary systems of periodic tasks. We extend Algorithm **RM** to define a global static-priority scheduling algorithm which we call Algorithm **RM-US[m/(3m-2)]**, and prove that Algorithm **RM-US[m/(3m-2)]** successfully schedules any periodic task system with utilization at most $m^2/(3m-2)$ on m identical processors.

3.1 "Light" systems

Definition 2 A periodic task system τ is said to be a *light system on m processors* if it satisfies the following two properties

Property P1: For each $\tau_i \in \tau,\ U_i \leq \dfrac{m}{3m-2}$

Property P2: $U(\tau) \leq \dfrac{m^2}{3m-2}$

■

We will consider the scheduling of task systems satisfying Property P1 and Property P2 above, using the rate-monotonic scheduling algorithm (Algorithm **RM**).

Theorem 3 Any periodic task system τ that is light on m processors will be scheduled to meet all deadlines on m processors by Algorithm **RM**.

Proof: Let us suppose that ties are broken by Algorithm **RM** such that τ_i has greater priority than τ_{i+1} for all $i,\ 1 \leq i < n$. Notice that whether jobs of τ_k meet their deadlines under Algorithm **RM** depends only upon the jobs generated by the tasks $\{\tau_1, \tau_2, \ldots, \tau_k\}$, and are completely unaffected by the presence of the tasks $\tau_{k+1}, \ldots, \tau_n$. For $k = 1, 2, \ldots, n$, let us define the task-set $\tau^{(k)}$ as follows:

$$\tau^{(k)} \stackrel{\text{def}}{=} \{\tau_1, \tau_2, \ldots, \tau_k\}.$$

Our proof strategy is as follows. We will prove that Algorithm **RM** will schedule $\tau^{(k)}$ in such a manner that all jobs of the lowest-priority task τ_k complete by their deadlines. Our claim that Algorithm **RM** successfully schedules τ would then follow by induction on k.

Lemma 3.1 Task system $\tau^{(k)}$ is feasible on m processors each of computing capacity $(\frac{m}{2m-1})$.

Proof: Since $m \geq 2$, notice that $3m - 2 > 2m - 1$. Since $U_i \leq \frac{m}{3m-2}$ for each task τ_i (by Property P1 above), it follows that

$$U_i \leq \frac{m}{2m-1} \tag{2}$$

Similarly from $U(\tau) \leq \frac{m^2}{3m-2}$ (Property P2 above) and the fact that $\tau^{(k)} \subseteq \tau$, it can be derived that

$$\sum_{\tau_i \in \tau^{(k)}} U_i \leq \frac{m^2}{2m-1}. \quad (3)$$

As a consequence of Inequalities 2 and 3 we may conclude that $\tau^{(k)}$ can be scheduled to meet all deadlines on m processors each of computing capacity $(\frac{m}{2m-1})$: the processor-sharing schedule (which we will henceforth denote OPT), which assigns a fraction U_i of a processor to τ_i at each time-instant bears witness to the feasibility of $\tau^{(k)}$.
■ **End proof** (of Lemma 3.1)

Since $\frac{m}{2m-1} \times (2 - \frac{1}{m}) = 1$, it follows from Theorem 1, the existence of the schedule OPT described in the proof of Lemma 3.1, and the fact that Algorithm **RM** is work-conserving, that

$$W(\mathbf{RM}, m, 1, \tau^{(k)}, t) \geq W(\text{OPT}, m, \frac{m}{2m-1}, \tau^{(k)}, t) \quad (4)$$

for all $t \geq 0$; i.e., *at any time-instant t, the amount of work done on $\tau^{(k)}$ by Algorithm* **RM** *executing on m unit-speed processors is at least as much as the amount of work done on $\tau^{(k)}$ by* OPT *on m $\frac{m}{2m-1}$-speed processors.*

Lemma 3.2 All jobs of τ_k meet their deadlines when $\tau^{(k)}$ is scheduled using Algorithm **RM**.

Proof: Let us assume that the first $(\ell - 1)$ jobs of τ_k have met their deadlines under Algorithm **RM**; we will prove below that the ℓ'th job of τ_k also meets its deadline. The correctness of Lemma 3.2 will then follow by induction on ℓ, starting with $\ell = 1$.

The ℓ'th job of τ_k arrives at time-instant $(\ell - 1)T_k$, has a deadline at time-instant ℓT_k, and needs C_k units of execution. From Inequality 4 and the fact that the processor-sharing schedule OPT schedules each task τ_j for $(\ell - 1)T_k \cdot U_j$ units over the interval $[0, (\ell - 1)T_k)$, we have

$$W(\mathbf{RM}, m, 1, \tau^{(k)}, (\ell-1)T_k) \geq (\ell-1)T_k \left(\sum_{j=1}^{k} U_j \right) \quad (5)$$

Also, at least $(\ell - 1) \cdot T_k \cdot (\sum_{j=1}^{k-1} U_j)$ units of this execution by Algorithm **RM** was of tasks $\tau_1, \tau_2, \ldots, \tau_{k-1}$ — this follows from the fact that exactly $(\ell - 1)T_k U_k$ units of τ_k's work has been generated prior to instant $(\ell - 1)T_k$; the remainder of the work executed by Algorithm **RM** must therefore be generated by $\tau_1, \tau_2, \ldots, \tau_{k-1}$.

The cumulative execution requirement of all the jobs generated by the tasks $\tau_1, \tau_2, \ldots, \tau_{k-1}$ that arrive prior to the deadline of τ_k's ℓ'th job is bounded from above by

$$\begin{aligned} & \sum_{j=1}^{k-1} \left\lceil \frac{\ell T_k}{T_j} \right\rceil C_j \\ < & \sum_{j=1}^{k-1} \left(\frac{\ell T_k}{T_j} + 1 \right) C_j \\ = & \ell T_k \sum_{j=1}^{k-1} U_j + \sum_{j=1}^{k-1} C_j \end{aligned} \quad (6)$$

As we have seen above (the discussion following Inequality 5) at least $(\ell - 1) \cdot T_k \cdot \sum_{j=1}^{k-1} U_j$ of this gets done prior to time-instant $(\ell - 1)T_k$; hence, at most

$$\left(T_k \sum_{j=1}^{k-1} U_j + \sum_{j=1}^{k-1} C_j \right) \quad (7)$$

remains to be executed *after* time-instant $(\ell - 1)T_k$.

The amount of processor capacity left unused by $\tau_1, \ldots, \tau_{k-1}$ during the interval $[(\ell - 1)T_k, \ell T_k)$ is therefore no smaller than

$$m \cdot T_k - \left(T_k \sum_{j=1}^{k-1} U_j + \sum_{j=1}^{k-1} C_j \right) \quad (8)$$

Since there are m processors available, the cumulative length of the intervals over $[(\ell - 1)T_k, \ell T_k)$ during which $\tau_1, \ldots, \tau_{k-1}$ leave at least one processor idle is minimized if the different processors tend to idle simultaneously (in parallel); hence, a lower bound on this cumulative length of the intervals over $[(\ell - 1)T_k, \ell T_k)$ during which $\tau_1, \ldots, \tau_{k-1}$ leave at least one processor idle is given by $(m \cdot T_k - \left(T_k \sum_{j=1}^{k-1} U_j + \sum_{j=1}^{k-1} C_j \right))/m$, which equals

$$T_k - \frac{1}{m} \left(T_k \sum_{j=1}^{k-1} U_j + \sum_{j=1}^{k-1} C_j \right) \quad (9)$$

For the ℓ'th job of τ_k to meet its deadline, it suffices that this cumulative interval length be at least as large at τ_k's execution requirement; i.e.,

$$\begin{aligned} & T_k - \frac{1}{m}(T_k \sum_{j=1}^{k-1} U_j + \sum_{j=1}^{k-1} C_j) \geq C_k \\ \equiv \quad & \frac{C_k}{T_k} + \frac{1}{m}(\sum_{j=1}^{k-1} U_j + \sum_{j=1}^{k-1} \frac{C_j}{T_k}) \leq 1 \\ \Leftarrow \quad & \textit{(Since } T_k \geq T_j \textit{ for } j < k\textit{)} \\ & U_k + \frac{1}{m}(2 \sum_{j=1}^{k-1} U_j) \leq 1 \end{aligned} \quad (10)$$

Let us now simplify the lhs of Inequality 10 above:

$$U_k + \frac{1}{m}(2 \sum_{j=1}^{k-1} U_j)$$

$$
\begin{aligned}
&\leq U_k + \frac{1}{m}(2\sum_{j=1}^{k} U_j - 2U_k) \\
&\leq \textit{(By Property P2 of task system } \tau\textit{)} \\
&\quad U_k(1-\frac{2}{m}) + \frac{2m}{3m-2} \\
&\leq \textit{(By Property P1 of task system } \tau\textit{)} \\
&\quad \frac{m}{3m-2}(1-\frac{2}{m}) + \frac{2m}{3m-2} \qquad (11) \\
&= 1 \qquad (12)
\end{aligned}
$$

From Inequalities 10 and 12, we may conclude that the ℓ'th job of τ_k does meet its deadline.

■ **End proof** (of Lemma 3.2)

The correctness of Theorem 3 follows from Lemma 3.2 by induction on k, with $k = m$ being the base case (that $\tau_1, \tau_2, \ldots \tau_m$ meet all their deadlines directly follows from the fact that there are m processors available in the system).

■ **End proof** (of Theorem 3)

3.2 Arbitrary systems

In Section 3.1, we saw that Algorithm **RM** successfully schedules any periodic task system τ with utilization $U(\tau) \leq m^2/(3m-1)$ on m identical processors, *provided each $\tau_i \in \tau$ has a utilization $U_i \leq m/(3m-2)$*. We now relax the restriction on the utilization of each individual task; rather, we permit any $U_i \leq 1$ for each $\tau_i \in \tau$. That is, we will consider in this section the static-priority global scheduling of any task system τ satisfying the condition

$$U(\tau) \leq \frac{m^2}{3m-2} .$$

For such task systems, we define the static priority-assignment scheme Algorithm **RM-US[m/(3m-2)]** as follows.

Algorithm RM-US[m/(3m-2)] assigns (static) priorities to tasks in τ according to the following rule:

if $U_i > \frac{m}{3m-2}$ **then** τ_i has the highest priority (ties broken arbitrarily)

if $U_i \leq \frac{m}{3m-2}$ **then** τ_i has rate-monotonic priority.

Example 1 As an example of the priorities assigned by Algorithm **RM-US[m/(3m-2)]**, consider a task system

$$
\begin{aligned}
\tau \stackrel{\text{def}}{=} \{&\tau_1 = (1,7), \tau_2 = (2,10), \tau_3 = (9,20), \\
&\tau_4 = (11,22), \tau_5 = (2,25)\}
\end{aligned}
$$

to be scheduled on a platform of 3 identical unit-speed processors. The utilizations of these five tasks are $\approx 0.143, 0.2, 0.45, 0.5$, and 0.08 respectively. For $m = 3$, $m/(3m-2)$ equals $3/7 \approx 0.4286$; hence, tasks τ_3 and τ_4 will be assigned highest priorities, and the remaining three tasks will be assigned rate-monotonic priorities. The possible priority assignments are therefore as follows (highest-priority task listed first):

$$\tau_3, \tau_4, \tau_1, \tau_2, \tau_5$$

or

$$\tau_4, \tau_3, \tau_1, \tau_2, \tau_5$$

Theorem 4 Any periodic task system τ with utilization $U(\tau) \leq m^2/(3m-2)$ will be scheduled to meet all deadlines on m unit-speed processors by Algorithm **RM-US[m/(3m-2)]**.

Proof: Assume that the tasks in τ are indexed according to the priorities assigned to them by Algorithm **RM-US[m/(3m-2)]**. First, observe that since $U(\tau) \leq m^2/(3m-2)$, while each task τ_i that is assigned highest priority has U_i strictly greater than $m/(3m-2)$, there can be at most $(m-1)$ such tasks that are assigned highest priority. Let k_o denote the number of tasks that are assigned the highest priority; i.e., $\tau_1, \tau_2, \ldots, \tau_{k_o}$ each have utilization greater than $m/(3m-2)$, and $\tau_{k_o+1}, \ldots \tau_n$ are assigned priorities rate-monotonically. Let $m_o \stackrel{\text{def}}{=} m - k_o$.

Let us first analyze the task system $\hat{\tau}$, consisting of the tasks in τ each having utilization $\leq m/(3m-2)$:

$$\hat{\tau} \stackrel{\text{def}}{=} \tau \setminus \tau^{(k_o)} .$$

The utilization of $\hat{\tau}$ can be bounded from above as follows:

$$
\begin{aligned}
U(\hat{\tau}) &= U(\tau) - U(\tau^{(k_o)}) \\
&< \frac{m^2}{3m-2} - k_o \cdot \frac{m}{3m-2} \\
&= \frac{m(m-k_o)}{3m-2} \\
&\leq \frac{(m-k_o)\cdot(m-k_o)}{3(m-k_o)-2} \\
&= \frac{m_o^2}{3m_o-2} \qquad (13)
\end{aligned}
$$

Furthermore, for each $\tau_i \in \hat{\tau}$, we have

$$U_i \leq \frac{m}{3m-2} \leq \frac{m_o}{3m_o-2} . \qquad (14)$$

From Inequalities 13 and 14, we conclude that $\hat{\tau}$ is a periodic task system that is light on m_o processors. Hence by Theorem 3, $\hat{\tau}$ can be scheduled by Algorithm **RM** to meet all deadlines on m_o processors.

Now, consider the task system $\tilde{\tau}$ obtained from τ by replacing each task $\tau_i \in \tau$ that has a utilization U_i greater

than $m/(3m-2)$ by a task with the same period, but with utilization equal to one:

$$\tilde{\tau} \stackrel{\text{def}}{=} \hat{\tau} \bigcup \left(\cup_{(C_i,T_i) \in \tau^{(k_o)}} \{(T_i, T_i)\} \right) .$$

Notice that Algorithm **RM-US[m/(3m-2)]** will assign identical priorities to corresponding tasks in τ and $\hat{\tau}$ (where the notion of "corresponding" is defined in the obvious manner). Also notice that when scheduling $\tilde{\tau}$, Algorithm **RM-US[m/(3m-2)]** will devote k_o processors exclusively to the k_o tasks in $\tau^{(k_o)}$ (these are the highest-priority tasks, and each have a utilization equal to unity) and will be executing Algorithm **RM** on the remaining tasks (the tasks in $\hat{\tau}$) upon the remaining $m_o = (m - k_o)$ processors. As we have seen above, Algorithm **RM** schedules the tasks in $\hat{\tau}$ to meet all deadlines; hence, Algorithm **RM-US[m/(3m-2)]** schedules $\tilde{\tau}$ to meet all deadlines of all jobs.

Finally, notice that an execution of Algorithm **RM-US[m/(3m-2)]** on task system τ can be considered to be an instantiation of a run of Algorithm **RM-US[m/(3m-2)]** on task system $\tilde{\tau}$, in which some jobs — the ones generated by tasks in $\tau^{(k_o)}$ — do not execute to their full execution requirement. By the result of Ha and Liu (Theorem 2), it follows that Algorithm **RM-US[m/(3m-2)]** is a *predictable* scheduling algorithm, and hence each job of each task during the execution of Algorithm **RM-US[m/(3m-2)]** on task system τ completes no later than the corresponding job during the execution of Algorithm **RM-US[m/(3m-2)]** on task system $\tilde{\tau}$. And, we have already seen above that no deadlines are missed during the execution of Algorithm **RM-US[m/(3m-2)]** on task system $\tilde{\tau}$.

■ **End proof** (of Theorem 4)

3.3 Harmonic task systems

In Section 3.2, we derived a performance bound for static-priority global multiprocessor scheduling of systems of periodic tasks with arbitrary periods. In **harmonic** periodic task systems, the periods T_i and T_j of any two tasks τ_i and τ_j are related as follows: either T_i is an integer multiple of T_j, or T_j is an integer multiple of T_i. With respect to the static-priority global multiprocessor scheduling of harmonic periodic task systems, we have also derived a variant of Algorithm **RM-US[m/(3m-2)]**, referred to as Algorithm **RM-US[m/(2m-1)]**, for which there exists a better sufficient utilization-based feasibility test — any harmonic task set with utilization $\leq m^2/(2m-1)$ is successfully scheduled by Algorithm **RM-US[m/(2m-1)]** on m identical processors. Due to space limitation, we have deferred the proof of the analog of Theorem 4 to [3].

4 Performance Evaluation

The purpose of this section is to show that, although **RM-US[m/(3m-2)]** can fail to meet deadlines at a utilization that is slightly higher than $m^2/(3m-2)$, it often performs much better than that for general task sets. To that end, we compare the performance of different techniques for static-priority preemptive scheduling on multiprocessors, namely partitioning, global, and global pfair [20]. To facilitate the comparison, the load of the task set is expressed in terms of the *system utilization*, $U_s(\tau)$, for a task set τ on m processors, which represents the fraction used of the total capacity of the multiprocessor platform: $U_s(\tau) \stackrel{\text{def}}{=} U(\tau)/m$.

4.1 The scheduling algorithms

We evaluate one partitioning scheme, R-BOUND-MPrespan, one pfair global scheme, WMpfair [20], and three global schemes, multiprocessor RM [16], adaptiveTkC [1] and **RM-US[m/(3m-2)]**. R-BOUND-MPrespan is a modification of the R-BOUND-MP scheme [13] where a necessary and sufficient schedulability test is used during task-to-processor assignment instead of the sufficient test used in the original version.

4.2 Theoretical comparison

Although Leung and Whitehead [14] have shown that the partitioned and global approaches are in general incomparable, it is still interesting to get a better understanding of the absolute and relative performance of the two approaches.

We begin by deriving an upper limit on the best possible system utilization bound for any static-priority multiprocessor scheduling algorithm. Consider the task set

$$\begin{aligned} \tau \stackrel{\text{def}}{=} & \{\tau_1 = (L, 2L-1), \tau_2 = (L, 2L-1), \ldots, \\ & \tau_m = (L, 2L-1), \tau_{m+1} = (L, 2L-1)\} \end{aligned}$$

to be scheduled on m processors (L is a positive integer) when all tasks arrive at time 0. For this task set, the system utilization is $L/(2L-1) + (L/(2L-1))/m$. For all studied static-priority scheduling approaches (partitioning, global and pfair global), deadlines will be missed for this task set. Partitioning will not succeed because it is necessary for two tasks to execute on one processor, and that processor will then have a utilization greater than 1; hence, the task set is unschedulable. For global scheduling, all m highest priority tasks will execute at the same time and occupy L time units during [0,2L-1). There will be $L-1$ time units available for a lower priority tasks, but the lowest priority task needs L time units and thus misses its deadline. By letting $L \to \infty$ and $m \to \infty$, the task set is unschedulable at a system utilization of $1/2$. Consequently, *the utilization guarantee bound for any static-priority multiprocessor scheduling algorithm (partitioned or global) cannot be higher than 1/2 of the capacity of the multiprocessor platform.*

Among the partitioned approaches, Oh and Baker [18] have shown that the First-Fit partitioning algorithm can schedule any task system with a system utilization $\leq \sqrt{2}-1$ on m processors. More recently, Lopez *et al.* [17] have

shown that some partitioning algorithms can schedule any task system with a system utilization $\leq (\sqrt{2}-1)(m+1)/m$. Among the global approaches, only **RM-US[m/(3m-2)]** has a tight system utilization bound, namely the derived bound of $m/(3m-2)$. For adaptiveTkC, the bound has been shown to be no greater than $2\,\frac{m}{3\,m-1+\sqrt{5\,m^2-6\,m+1}}$ [2]. For RM, the bound is known to be no greater than $1/m$ [10]. For WM, no known utilization bound has hitherto been proven; however, due to the reasoning above the system utilization bound of WM cannot be higher than 1/2.

Based on the expressions above for the system utilization bounds, we see that Algorithm **RM-US[m/(3m-2)]** is inferior to the best partitioning algorithms. For example, for a system with $m = 2$, the Lopez bound gives 62% whereas Algorithm **RM-US[m/(3m-2)]** gives 50%. As $m \to \infty$, the Lopez bound, at 42%, still proves superior to Algorithm **RM-US[m/(3m-2)]**'s 33%.

4.3 Experimental setup

Each simulation experiment represents simulation of 900 task sets, organized in 30 different buckets, each with 30 task sets[3]. Bucket i contains task sets with a system utilization greater than $U_{i,low} = (i-1)/30$, but no greater than $U_{i,high} = i/30$. For each bucket, we compute the success ratio as the number of successfully scheduled task sets in that bucket divided by the number of scheduled task sets in that bucket. Since the partitioning and global schemes use different strategies for assigning a task to a processor, the concept of 'successfully scheduled' needs to be clearly defined. For R-BOUND-MP, we consider a task to be successfully scheduled if and only if the schedulability test in the partitioning algorithm can guarantee that the task set on each uniprocessor is schedulable. For the other schemes, we consider a task to be successfully scheduled[4] if it met all its deadlines during $[0, lcm(T_1, T_2, \ldots, T_n))$.

The task set of each bucket i is generated by starting with a current task set that is empty, and then adding a new task to the current task set as long as the system utilization is lower than $U_{i,low}$. When the system utilization of the current task set has become higher than $U_{i,low}$, we decide whether or not the current task set should be inserted into the bucket. If the system utilization of the current task set is lower than $U_{i,high}$ and the number of tasks is greater than the number of processors, then the task set is put into the bucket; otherwise, a new task set is generated.

The periods and the execution times of the tasks are selected randomly. The period of a task is drawn from the set of discrete periods, $T_i \in \{100, 200, 300, 400, ..., 1000\}$, each period having the same probability of being selected[5]. We only study synchronous task sets, which means that all generated tasks arrive for the first time at time 0 and are scheduled until time $lcm(T_1, T_2, \ldots, T_n)$[6].

The execution time of a task is computed from the utilization of that task and rounded down to the nearest integer. The utilization of a task is given by either a uniform distribution or a binomial distribution. To determine which distribution to use, we generate a random variable with uniform distribution in the range $[0, 1]$. If the variable is less than F (a simulation parameter), we then choose the uniform distribution; otherwise, the binomial distribution is chosen. In case of a uniform distribution, the utilization of a task is drawn from the range $(0, 1]$. In case of a binomial distribution, the utilization of a task is generated in the following way. Perform 29 trials with the probability of success being A (another simulation parameter). Count the number of successes and divide by 29. Then add a random number with a uniform distribution in the range $[-1/29, 1/29]$. If the utilization of a task is less than or equal to zero, or greater than 1, then generate a task again. Note that, with this procedure, a high value of A makes it more likely that a task has a high utilization.

4.4 Experimental results

The results of the experiments on a system with $m = 32$ processors and $F = 0.1$ are shown[7] in Figure 1. For each experiment, we have plotted the performance (success ratio) as a function of the system utilization (with a resolution given by the bucket intervals) for different values of parameter A. From the plots, we draw the following conclusions.

We first observe that **RM-US[m/(3m-2)]** often succeeds at much higher system utilizations than is suggested by its utilization bound. For example, for $m = 32$ processors and $A \leq 0.3$, **RM-US[m/(3m-2)]** breaks down at a system utilization of approximately 80%, while the corresponding theoretical bound is $32/(3*32-2) \approx 34\%$. Note that, when $A = 0.5$, **RM-US[m/(3m-2)]** suffers from a significant performance drop. Here, the breakdown utilization is as low as 50%. This phenomenon is actually an effect of the chosen experimental setup. With our choice of distributions, the expected value of the utilization of a task is approximately 50% for both the uniform distribution and the binomial distribution, thus resulting in a very large population of tasks with that utilization.

[3]The experimental setup used is similar to the experimental setup in [20]. Our experiment differs from that in [20] in that we only simulate 30 buckets with 30 tasks in each bucket (in contrast, [20] simulated 100 buckets with 100 tasks in each bucket.)

[4]Note that, in [20], WM was considered to be successfully scheduled if and only if a certain pfairness property was satisfied. Since all evaluated scheduling algorithms, except WM, was primarily designed for periodic scheduling rather than to satisfy the pfairness property, we chose to evaluate all scheduling algorithms under the assumption of periodic scheduling. Since the pfairness property is a stronger condition than periodicity, WM will show no worse performance in our study than in [20].

[5]In [20], the task periods used in the experiments were not stated at all.

[6]At time $t \geq lcm(T_1, T_2, \ldots, T_n)$, the tasks that execute is the same as the tasks that execute at $t - lcm(T_1, T_2, \ldots, T_n)$.

[7]Results from complementary experiments assuming other values of m and F can be found in [3].

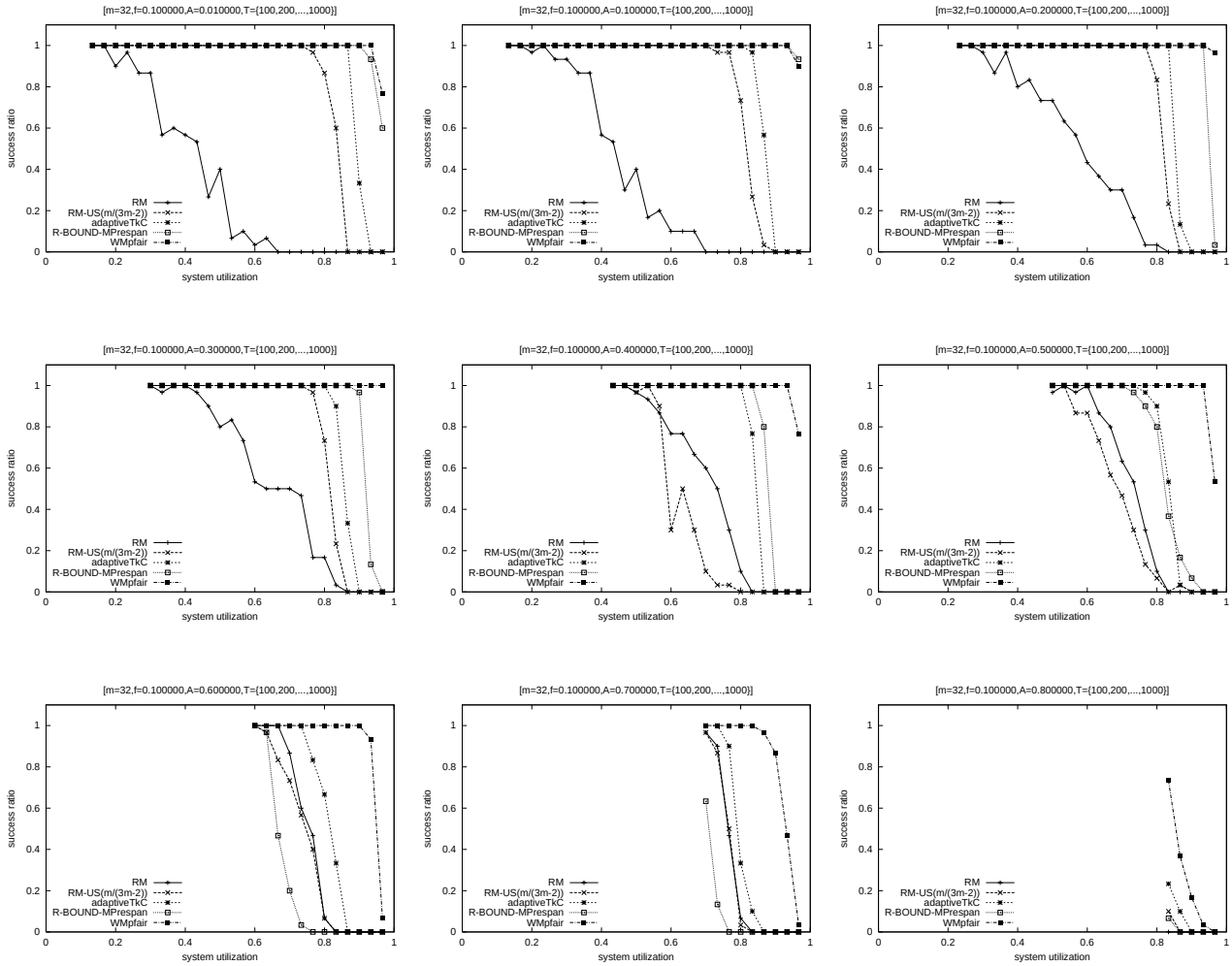

Figure 1: Success ratio as a function of system utilization for different scheduling algorithms on 32 processors and F=0.1.

We then observe that **RM-US[m/(3m-2)]** outperforms RM when many processors are available and A is small (tasks have a low average utilization). The reason for this is that **RM-US[m/(3m-2)]** always succeeds to schedule task sets with a system utilization less than $m/(3m-2)$ while RM can potentially fail due to Dhall's effect. As A becomes larger RM and **RM-US[m/(3m-2)]** offer comparable performance since most tasks then have a utilization greater than the guarantee bound of $m/(3m-2)$. For example, when $m = 32$ and $A \geq 0.7$, most tasks have a utilization greater than the corresponding bound of 34%, which means that RM and **RM-US[m/(3m-2)]** produce the same priority assignment and hence exhibit similar performance.

We can also see that **RM-US[m/(3m-2)]** performs worse than WMpfair and adaptiveTkC for systems with a large number of processors. However, the difference in performance is typically no more than 20%, which shows that **RM-US[m/(3m-2)]** does not suffer from the drawbacks of RM. **RM-US[m/(3m-2)]** also performs worse than R-BOUND-MPrespan as long as $A \leq 0.5$. For higher values of A, the fundamental limitations of the assignment strategy used in R-BOUND-MPrespan (a bin-packing algorithm) reveal themselves and causes a significant performance drop. Note that WMpfair performs significantly better than both R-BOUND-MPrespan and adaptiveTkC. The reason is that the chosen task periods are long relative to the time unit base, which means that WMpfair approximates optimal processor sharing. However, when task periods are drawn from a set of shorter periods, WMpfair offers a performance similar to R-BOUND-MPrespan and adaptiveTkC [3].

Note that, for small values of A, the success ratio of RM is heavily changing. The reason is that, with these parame-

ters, most tasks are likely to have a low utilization, but there is a 10% probability for each task that its utilization will be drawn from a uniform distribution and hence have a higher likelihood of becoming large. Now, if there is a task with a large utilization in a population of low-utilization tasks, then RM can fail to meet deadlines at low system utilization due to Dhall's effect.

5 Conclusions

We have studied the preemptive scheduling of systems of periodic tasks on a platform comprised of several identical processors. To this end, we have presented two major theoretical results. First, we have proposed Algorithm **RM-US[m/(3m-2)]**, a new static-priority multiprocessor algorithm for scheduling periodic task systems. We proved that Algorithm **RM-US[m/(3m-2)]** successfully schedules any periodic task system with utilization $\leq m^2/(3m-2)$ on m identical processors. We have also shown that, in general, no static-priority scheduling algorithm (partitioned or global) on a multiprocessor can achieve a utilization bound that is greater than 50% of the platform capacity.

To assess the performance of Algorithm **RM-US[m/(3m-2)]** with respect to other static-priority multiprocessor scheduling algorithm, we have also provided a theoretical and experimental evaluation. The results from this evaluation shows that Algorithm **RM-US[m/(3m-2)]** outperforms (in terms of success ratio) the partitioned approach when there is a large population of tasks with high utilization. In general, however, Algorithm **RM-US[m/(3m-2)]** is inferior to the best partitioned algorithm in terms of both utilization bound and success ratio. We would like to point out that the results in the current paper remain significant despite this — since it has been shown [14] that the partitioned and global approaches are in general incomparable, it behooves us to better understand both kinds of scheduling systems.

References

[1] B. Andersson and J. Jonsson. Fixed-priority preemptive multiprocessor scheduling: To partition or not to partition. In *Proc. of the Int'l Conf. on Real-Time Computing Systems and Applications*, pages 337–346, Cheju Island, South Korea, Dec. 2000.

[2] B. Andersson and J. Jonsson. Some insights on fixed-priority preemptive non-partitioned multiprocessor scheduling. In *Proc. of the Real-Time Systems Symposium – Work-In-Progress Session*, Orlando, Florida, Nov. 2000.

[3] B. Andersson, S. Baruah, and J. Jonsson. Static-priority scheduling on multiprocessors. Tech. Rep. UNC-CS TR01-016, Department of Computer Science, University of North Carolina at Chapel Hill, May 2001.

[4] S. Baruah, N. Cohen, G. Plaxton, and D. Varvel. Proportionate progress: A notion of fairness in resource allocation. *Algorithmica*, 15(6):600–625, June 1996.

[5] S. Baruah, J. Haritsa, and N. Sharma. On-line scheduling to maximize task completions. In *Proc. of the Real-Time Systems Symposium*, pages 228–237, San Juan, Puerto Rico, Dec. 1994.

[6] 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. *Real-Time Systems*, 4(2):125–144, June 1992.

[7] S. Baruah, G. Koren, B. Mishra, A. Raghunathan, L. Rosier, and D. Shasha. On-line scheduling in the presence of overload. In *Proc. of the 32nd Annual IEEE Symposium on Foundations of Computer Science*, pages 100–110, San Juan, Puerto Rico, Oct. 1991.

[8] S. Davari and S. K. Dhall. On a real-time task allocation problem. In *Proc. of the 19th Hawaii Int'l Conf. on System Science*, pages 8–10, Honolulu, Hawaii, Jan. 1985.

[9] S. Davari and S. K. Dhall. An on-line algorithm for real-time tasks allocation. In *Proc. of the Real-Time Systems Symposium*, pages 194–200, New Orleans, Louisiana, Dec. 1986.

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

[11] R. Ha and J. W. S. Liu. Validating timing constraints in multiprocessor and distributed real-time systems. In *Proc. of the IEEE Int'l Conf. on Distributed Computing Systems*, pages 162–171, Poznan, Poland, June 1994.

[12] B. Kalyanasundaram and K. Pruhs. Speed is as powerful as clairvoyance. In *36th Annual Symposium on Foundations of Computer Science*, pages 214–223, Oct. 1995.

[13] S. Lauzac, R. Melhem, and D. Mossé. An efficient RMS admission control algorithm and its application to multiprocessor scheduling. In *Proc. of the Int'l Parallel Processing Symposium*, pages 511–518, Apr. 1998.

[14] J. Y.-T. Leung and J. Whitehead. On the complexity of fixed-priority scheduling of periodic, real-time tasks. *Performance Evaluation*, 2(4):237–250, Dec. 1982.

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

[16] C. L. Liu. Scheduling algorithms for multiprocessors in a hard real-time environment. *JPL Space Programs Summary 37-60 II*, 28–31, 1969.

[17] J. M. Lopez, J. L. Diaz and D. F. Garcia. Minimum and Maximum Utilization Bounds for Multiprocessor RM Scheduling. In *Proc. of the Euromicro Conf. on Real-Time Systems*, June 2001.

[18] D.-I. Oh and T. P. Baker. Utilization bounds for N-processor rate monotone scheduling with static processor assignment. *Real-Time Systems*, 15(2):183–192, Sep. 1998.

[19] C. A. Phillips, C. Stein, E. Torng, and J. Wein. Optimal time-critical scheduling via resource augmentation. In *Proc. of the Twenty-Ninth Annual ACM Symposium on Theory of Computing*, pages 140–149, El Paso, Texas, May 1997.

[20] S. Ramamurthy and M. Moir. Static-priority static scheduling on multiprocessors. In *Proc. of the Real-Time Systems Symposium*, pages 69–78, Orlando, Florida, Nov. 2000.