Static-priority scheduling on multiprocessors*

Bjorn Andersson Sanjoy Baruah Jan Jonsson

Abstract

The preemptive scheduling of systems of periodic tasks on a platform comprised of several identical multiprocessors 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; for the special case of harmonic periodic task systems, the algorithm is proven to successfully schedule any system with a worst-case utilization of no more than half the platform capacity.

Keywords. Multiprocessor scheduling; periodic tasks; global scheduling; static priorities.

1 Introduction

Over the years, the preemptive periodic task model [17, 16] has proven remarkably useful for the modelling of recurring processes 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.) that facilitate the application systems designer who is able to model his or her real-time application system 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

*Supported in part by the National Science Foundation (Grant Nos. CCR-9704206, CCR-9972105, CCR-9988327, and ITR-0082866).

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 property 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* [16], 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.

[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.

Partitioned versus global scheduling. In this paper, we will study the static-priority scheduling of systems of periodic tasks on m identical multiprocessors, $m \geq 2$. In scheduling such systems, there are (at least) two distinct approaches possible.

- 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 [15] that the partioned 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 result of Leung and Whitehead [15] provides a very strong motivation to study both the partitioned and the non-partitioned approaches to static-priority multiprocessor scheduling, since neither approach is strictly better than the other.

This research. The partitioned approach to static-priority multiprocessor scheduling has been extensively studied (see [18] for an excellent overivew). 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. 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, and have presented the result and its proof in the appendix.

For the special case of *harmonic* periodic task systems – task sets in which the periods T_i and T_j of any two tasks τ_i and τ_j satisfy the relationship that either T_i is an integer multiple of T_j, or T_j is an integer multiple of T_i – we show that our algorithm offers an even better performance guarantee. Specifically, we prove that our algorithm successfully schedules any harmonic periodic task system τ with utilization $U(\tau) \leq m^2/(2m-1)$ on m identical processors; as $m \to \infty$, this bound approaches $m/2$ from above.

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 present an algorithm based upon Algorithm RM-US[m/(3m-2)], optimized for scheduling harmonic task sets. In Section 5, we describe a series of experiments we have conducted to evaluate the performance of Algorithm RM-US[m/(3m-2)] on randomly-generated task sets. In Section 6, we briefly review related research on the topic of multiprocessor real-time scheduling, and conclude in Section 7 with a brief summary of the results contained in this paper. Some proofs are postponed to the appendix.

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 [6, 5, 4] 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 [6, 5, 4]. 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; a proof may be found in Section A in the appendix.

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.)

[2]Resource augmentation as a technique for improving the performance on on-line scheduling algorithms was formally proposed by Kalyanasundaram and Pruhs [13].

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

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

■

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

Predictable scheduling algorithms. Ha and Liu [11, 12, 10] 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 using 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 <u>at most</u> 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, 12, 10] 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)]

In this section, we 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

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

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 Therorem 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

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

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

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

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 $(\ell-1)T_kU_k$ 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

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

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

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

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

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

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

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

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.,

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

Let us now simplify the lhs of Inequality 10 above:

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

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

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

to be scheduled on a platform of 3 identical unit-speed processors. The utilizations of these five tasks are ≈ 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:

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

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

$$U_i \leq \frac{m}{3m-2} \leq \frac{m_o}{3m_o-2} \ . \tag{14}$$

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).

4 Harmonic task systems

In Section 3, we studied the static-priority global multiprocessor scheduling of systems of periodic tasks. 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 now present a variant of Algorithm RM-US[m/(3m-2)], which we call Algorithm RM-US[m/(2m-1)], and prove below a stronger bound on the performance of Algorithm RM-US[m/(2m-1)].

First, let us refine the definition of **light** systems for harmonic task systems. Specifically, let us call a harmonic task system τ *light on* m *processors* if $U(\tau) \leq m^2/(2m-1)$ and $U_i \leq m/(2m-1)$ for all $\tau_i \in \tau$.

Algorithm RM-US[m/(2m-1)] assigns (static) priorities to tasks in harmonic periodic task system τ according to the following rule:

if $U_i > \frac{m}{2m-1}$ **then** τ_i has the highest priority (ties broken arbitrarily)

if $U_i \leq \frac{m}{2m-1}$ **then** τ_i has rate-monotonic priority.

Theorem 5 Any periodic task system τ with utilization $U(\tau) \leq m^2/(2m-1)$ will be scheduled to meet all deadlines on m unit-speed processors by Algorithm RM-US[m/(2m-1)].

Proof Sketch:

- The analog of Theorem 3 – that any harmonic periodic task system τ that is light on m processors will be scheduled to meet all deadlines on m processors by Algorithm RM — can be proved in a manner that closely parallels the proof of Theorem 3.
  - It may be verified that the proof of Lemma 3.1 goes through unchanged if $\tau^{(k)}$ is assumed to be a light harmonic periodic task system; Lemma 3.1 is therefore applicable to light harmonic periodic task systems as well.
  - The proof of the analog of Lemma 3.2 is provided in Section B in the appendix; the crucial difference arises from the simplification that results in Inequality 6. While in obtaining Inequality 6 we replaced $\left\lceil \frac{\ell T_k}{T_j} \right\rceil$ by $\left(\frac{\ell T_k}{T_j} + 1 \right)$, notice that in the harmonic tasks case we may simply replace $\left\lceil \frac{\ell T_k}{T_j} \right\rceil$ by $\frac{T_k}{T_j}$; this is because in a harmonic task set it is guaranteed that T_k is an integer multiple of T_j. The remainder of the proof is merely algebraic manipulation and proceeds directly; details may be found in the appendix.
- The correctness of Theorem 5 for light harmonic task systems now follows directly from the correctness of this analog of Lemma 3.2, by induction.
- To prove Theorem 5 for arbitrary harmonic task systems, we use techniques identical to those used in Section 3.2. I.e., we consider the scheduling on m processors of any task system τ with $U(\tau) \leq m^2/(2m-1)$; for such a system, (i) we "inflate" to unity the utilizations of all tasks in τ that have utilizations $> m/(2m-1)$ (ii) we prove that, as a consequence of the correctness of the theorem on light systems (as described above), we may conclude that the remaining tasks will be successfully scheduled by Algorithm RM-US[m/(2m-1)] on the remaining processors to meet all deadlines,and (iii) use the result of Ha and Liu (Theorem 2) to conclude that Algorithm RM-US[m/(3m-2)] will therefore successfully schedule the entire task system.

■

5 Experimental Evaluation

The purpose of this section is to show that, although RM-US[m/(3m-2)] can fail to meet deadlines at a system utilization that is slightly higher than m/(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, non-partitioning and non-partitioning pfair [20]. Section 5.1 describes the experimental setup in terms of simulation parameters and scheduling algorithms used. Section 5.2 presents the results from the experiments and the observations made. Finally, Section 5.3 compares the theoretical utilization bounds for the scheduling algorithms used.

5.1 Experimental Setup

Our experimental setup is similar to the experimental setup in [20], but for completeness the setup is described below.

Each simulation experiment represents simulation of 900 task sets, organized in 30 different buckets, each with 30 task sets. 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. 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. 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.)

The periods and the execution time of a task are selected randomly. The period of a task is drawn from a set of discrete periods, each period having the same probability of being selected. In our experiments, we draw the period of task i from one of the following two different period sets: $T_i \in \{100, 200, 300, 400, ..., 1000\}$ and $T_i \in \{2, 4, 6, 8, ..., 20\}$. Note that, in [20], the type of period sets used in the experiments was not stated at all. Since we study synchronous task sets, all generated tasks arrive for the first time at time 0 and are scheduled until time $lcm(T_1, T_2, \ldots, T_n)$[3].

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 or equal to 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

[3]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 $t - lcm(T_1, T_2, \ldots, T_n)$.

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 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.

We evaluate one partitioning scheme, R-BOUND-MPrespan, one pfair non-partitioning scheme: WMpfair [20], and three non-partitioning schemes, RM [17], adaptiveTkC [1] and RM-US[m/(3m-2)]. R-BOUND-MPrespan is a modification of the R-BOUND-MP scheme [14] where a necessary and sufficient schedulability test is used during task-to-processor assignment instead of the sufficient test used in the original version. Since the partitioning and non-partitioning 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 if it met all its deadlines during $[0, lcm(T_1, T_2, \ldots, T_n)$. 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].

5.2 Experimental Results

The results of the experiments for different values of the parameters F and A are shown in Figures 1 through 6 in Section C (in the appendix). 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 around 80%, while the corresponding theoretical bound is 34%. Note that, when $m = 32$ and $A = 0.5$, RM-US[m/(3m-2)] has 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 0.5 for both the uniform distribution and the binomial distribution, thus resulting in a very large population of tasks with that utilization. A similar behavior was observed in [20].

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 set with a system utilization less than m/(3m-2) while RM can potentially fail due to *Dhall's effect* [9]. 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 $32/(3 * 32 - 3) \approx 0.34$, which means that RM and RM-US[m/(3m-2)] produce the same priority assignment and hence 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. When the task periods are drawn from the set of long periods, WMpfair performs significantly better than both R-BOUND-MPrespan and adaptiveTkC. The reason is that when periods are long (relative to the time unit base), WMpfair approximates processor sharing, which is optimal. When task periods are drawn from the set of short periods, WMpfair offer a performance similar to R-BOUND-MPrespan and adaptiveTkC.

It is worth noting that all scheduling algorithms perform well on one processor. The reason for this is that all the evaluated scheduling algorithms, except WMpfair and R-BOUND-MPrespan, perform scheduling in the same manner as RM. The reason why R-BOUND-MP schedules tasks differently than RM is because of a special task set transformation. In the beginning of R-BOUND-MP-algorithm, a task set τ is transformed into $\tau_i\prime$, to make the ratio between periods lower. After the transformation, it may then be the case that there exist task sets such that τ is schedulable on one processor, but $\tau\prime$ is not.

Note that, for 32 processors with $F = 0.1$ and $A = 0.01$, the success ratio of RM is heavily changing. The reason is that, with these parameters, 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.

Finally, we make a comment on the performance of WMpfair. The simulation results show that the success ratio of WMpfair is higher for $T_i \in \{100, 200, 300, 400, ..., 1000\}$ than it is for $T_i \in \{2, 4, 6, 8, ..., 20\}$. As mentioned above, this effect occurs because WMpfair approximates processor sharing (which is optimal) when periods are large enough (in comparison to the time unit base). For all other scheduling algorithms, the periods do not affect the success ratio. One notable exception is when $F = 1$, $m = 1$ and the system utilization is greater than 95%. In this case, $T_i \in \{2, 4, 6, 8, ..., 20\}$ yields higher success ratio because of the way task sets are generated. The reason is as follows. Most of the task sets have only 2 tasks (more tasks often yields a utilization higher than 1, and are hence rejected). Often the only way of achieving a utilization within the bounds is to generate task sets until the periods are harmonious. Those task sets can be scheduled by all algorithms except WMpfair.

5.3 Utilization bounds

We conclude our performance evaluation by comparing the theoretical bounds of the studies scheduling schemes. To this end, we will begin by deriving an upper guarantee bound of system utilization. Consider the task set $\tau = \{(T_1 = 2L-1, C_1 = L), (T_2 = 2L-1, C_2 = L), \ldots, T_m = 2L-1, C_m = L), T_{m+1} = 2L-1, C_{m+1} = L)\}$ 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 approaches (partitioning, non-partitioning and pfair non-partitioning) of using static-priority scheduling, 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 non-partitioning (whether pfair or not), 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, for any particular scheduling algorithm addressed in this paper, the utilization bound cannot be higher than 1/2.

Among the studied approaches, only RM-US[m/(3m-2)] has a tight utilization bound, namely the derived bound of $m/(3m-2)$. For adaptiveTkC, the utilization 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 utilization bound is known to be no greater $1/m$ [9]. For WM, no known utilization bound has hitherto been proven; however, due to the reasoning above the utilization bound cannot be higher than 1/2.

We can thus conclude that, in general, no static-priority scheduling algorithm on a multiprocessor can achieve a utilization bound that is greater than 50%. To that end, it is interesting to note that the utilization bound of a static multiprocessor scheduling algorithm has previously been shown to be 41% or higher [18].

6 Related Work

The problem of scheduling a given set of periodic tasks on identical multiprocessor machines was posed by Liu [17] in 1969. Liu derived conditions under which the earliest deadline first scheduling algorithm would successfully schedule such a system; these conditions translate into a sufficient (albeit not necessary) feasibility test. Much later, Baruah et al. [3] obtained a necessary and sufficient conditions for determining feasibility, as well as an optimal scheduling algorithm for successfully scheduling feasible systems.

The partioned approach to the static-priority multiprocessor scheduling of periodic task systems has also been extensively studied [9, 7, 8, 18]. Much of this research has considered the problem of using bin-packing like algorithms for partitioning a given set of periodic tasks among a set of processors such that each partition is uniprocessor feasible under the rate-monotonic algorithm; e.g., Dhavri and Dhall [7] presented an efficient algorithm which they proved would partition a set of periodic tasks into no more than twice as many partitions that an optimal algorithm would (equivalently, they devised an efficient algorithm for partitioned static-priority multiprocessor scheduling that uses at most twice as many processors as an optimal algorithm would). More recently, Oh and Baker [18] presented a partitioned static-priority multiprocessor scheduling algorithm that schedules any task system with utilization $\leq m(\sqrt{2}-1)$ on m processors — this represents a utilization of approximately 42% of the capacity of the multiprocessor platform. For $m = 2$ and $m = 3$, Algorithm RM-US[m/(3m-2)] offers a superior bound; however as $m \to \infty$, the Oh & Baker bound, at 42%, proves superior to Algorithm RM-US[m/(3m-2)]'s 33%. (We would like to point out that the results in the current paper remain significant despite this – since Leung and Whitehead [15] have proven that the partioned and global approaches are in general incomparable, it behooves us to better understand both kinds of scheduling systems.)

7 Conclusions

We have studied the preemptive scheduling of systems of periodic tasks on a platform comprised of several identical multiprocessors. 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. For the special case of harmonic periodic task systems, we obtained 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/(3m-2)] on m identical processors.

References

[1] ANDERSSON, B., AND JONSSON, J. Fixed-priority preemptive multiprocessor scheduling: To partition or not to partition. In *Proceedings of the International Conference on Real-Time Computing Systems and Applications* (Cheju Island, South Korea, December 2000), IEEE Computer Society Press, pp. 337–346.

[2] ANDERSSON, B., AND JONSSON, J. Some insights on fixed-priority preemptive non-partitioned multiprocessor scheduling. In *Proceedings of the Real-Time Systems Symposium – Work-In-Progress Session* (Orlando, FL, November 2000).

[3] BARUAH, S., COHEN, N., PLAXTON, G., AND VARVEL, D. Proportionate progress: A notion of fairness in resource allocation. *Algorithmica 15*, 6 (June 1996), 600–625.

[4] BARUAH, S., HARITSA, J., AND SHARMA, N. On-line scheduling to maximize task completions. In *Proceedings of the Real-Time Systems Symposium* (San Juan, Puerto Rico, 1994), IEEE Computer Society Press.

[5] BARUAH, S., KOREN, G., MAO, D., MISHRA, B., RAGHUNATHAN, A., ROSIER, L., SHASHA, D., AND WANG, F. On the competitiveness of on-line real-time task scheduling. *Real-Time Systems 4* (1992), 125–144. Also in *Proceedings of the 12th Real-Time Systems Symposium, San Antonio, Texas, December 1991*.

[6] BARUAH, S., KOREN, G., MISHRA, B., RAGHUNATHAN, A., ROSIER, L., AND SHASHA, D. On-line scheduling in the presence of overload. In *Proceedings of the 32nd Annual IEEE Symposium on Foundations of Computer Science* (San Juan, Puerto Rico, October 1991), IEEE Computer Society Press, pp. 100–110.

[7] DAVARI, S., AND DHALL, S. K. On a real-time task allocation problem. In *Proceedings of the 19th Hawaii International Conference on System Science* (Honolulu, January 1985).

[8] DAVARI, S., AND DHALL, S. K. An on-line algorithm for real-time tasks allocation. In *Proceedings of the Real-Time Systems Symposium* (1986), pp. 194–200.

[9] DHALL, S. K., AND LIU, C. L. On a real-time scheduling problem. *Operations Research 26* (1978), 127–140.

[10] HA, R. *Validating timing constraints in multiprocessor and distributed systems*. PhD thesis, Department of Computer Science, University of Illinois at Urbana-Champaign, 1995. Available as Technical Report No. UIUCDCS-R-95-1907.

[11] HA, R., AND LIU, J. W. S. Validating timing constraints in multiprocessor and distributed real-time systems. Tech. Rep. UIUCDCS-R-93-1833, Department of Computer Science, University of Illinois at Urbana-Champaign, October 1993.

[12] HA, R., AND LIU, J. W. S. Validating timing constraints in multiprocessor and distributed real-time systems. In *Proceedings of the 14th IEEE International Conference on Distributed Computing Systems* (Los Alamitos, June 1994), IEEE Computer Society Press.

[13] KALYANASUNDARAM, B., AND PRUHS, K. Speed is as powerful as clairvoyance. In *36th Annual Symposium on Foundations of Computer Science (FOCS'95)* (Los Alamitos, Oct. 1995), IEEE Computer Society Press, pp. 214–223.

[14] LAUZAC, S., MELHEM, R., AND MOSSE, D. An efficient RMS admission control algorithm and its application to multiprocessor scheduling. In *Proceedings of the International Parallel Processing Symposium* (April 1998), IEEE Computer Society Press, pp. 511–518.

[15] LEUNG, J., AND WHITEHEAD, J. On the complexity of fixed-priority scheduling of periodic, real-time tasks. *Performance Evaluation 2* (1982), 237–250.

[16] LIU, C., AND LAYLAND, J. Scheduling algorithms for multiprogramming in a hard real-time environment. *Journal of the ACM 20*, 1 (1973), 46–61.

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

[18] OH, D.-I., AND BAKER, T. P. Utilization bounds for N-processor rate monotone scheduling with static processor assignment. *Real-Time Systems: The International Journal of Time-Critical Computing 15* (1998), 183–192.

[19] PHILLIPS, C. A., STEIN, C., TORNG, E., AND WEIN, J. Optimal time-critical scheduling via resource augmentation. In *Proceedings of the Twenty-Ninth Annual ACM Symposium on Theory of Computing* (El Paso, Texas, 4–6 May 1997), pp. 140–149.

[20] RAMAMURTHY, S., AND MOIR, M. Static-priority static scheduling on multiprocessors. In *Proceedings of the Real-Time Systems Symposium* (Orlando, FL, November 2000), IEEE Computer Society Press.

A Proof of Theorem 1

The proof below is from [19]; it is restated here in the notation and terminology used in the rest of this paper.

The statement of the theorem is:
For any instance of jobs I, any time-instant $t \geq 0$, any work-conserving algorithm A, and any algorithm A', it is the case that

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

The proof is by contradiction. Suppose then that it is not true; i.e., there is some time-instant by which a work-conserving algorithm A executing on m speed-$(2 - \frac{1}{m}) \cdot s$ processors has performed strictly less work than some other algorithm A' executing on m speed-s processors. Let $J_j \in I$ denote a job with the earliest arrival time such that there is some time-instant t_o satisfying

$$W(A, m, (2 - \frac{1}{m}) \cdot s, I, t_o) < W(A', m, s, I, t_o),$$

and the amount of work done on job j by time-instant t_o in A is strictly less than the amount of work done of J_i by time-instant t_o in A'. One such J_i must exist, because there is a time $t < t_i$ such $W(A, m, (2 - \frac{1}{m}) \cdot s, I, t_o) = W(A', m, s, I, t_0)$ — $t = 0$ gives one such equality.

By our choice of r_j, it must be the case that

$$W(A, m, (2 - \frac{1}{m}) \cdot s, I, r_j) \geq W(A', m, s, I, r_j).$$

Therefore, the amount of work done by A' over $[r_j, t_o)$ is strictly more than the amount of work done by A over the same interval. The fact that the amount of work done on J_i in $[r_j, t_o)$ in A is less than the amount of work done on J_i in $[r_j, t_o)$ in A', implies that J_i does not complete before t_o.

Let x denote the cumulative length of time over the interval $[r_j, t_o)$ during which A is executing on all m processors; let $y \stackrel{\text{def}}{=} (t_o - r_j) - x)$ denote the length of time over this interval during which A idles some processor.

We make the following two observations.

- Since A is a work-conserving scheduling algorithm, job J_j, which has not completed by instant t_o in the schedule generated by A, must have executed for at least y time units by time t_o in the schedule generated by A; while it could have executed for at most $(x + y)$ time units in the schedule generated by A'; therefore,

$$(x + y) > \left(2 - \frac{1}{m}\right) \cdot y \ . \tag{15}$$

- The amount of work done by A over $[r_j, t_o)$ is at least

$$\left(2 - \frac{1}{m}\right) \cdot s(mx + y) \ ,$$

while the amount of work done by A' over this interval is at most

$$ms(x+y)\ ;$$

therefore, it must be the case that

$$m(x+y) > \left(2 - \frac{1}{m}\right) \cdot (mx+y)\ . \tag{16}$$

Adding $(m-1)$ times Inequality 15 to Inequality 16, we get

$$\begin{aligned}
&(m-1)(x+y) + m(x+y) > (m-1)\left(2-\frac{1}{m}\right)\cdot y + \left(2-\frac{1}{m}\right)\cdot(mx+y)\\
\equiv\ &(2m-1)(x+y) > \left(2-\frac{1}{m}\right)\cdot(my - y + mx + y)\\
\equiv\ &(2m-1)(x+y) > \left(2-\frac{1}{m}\right) m(x+y)\\
\equiv\ &(2m-1)(x+y) > (2m-1)(x+y)
\end{aligned}$$

which is a contradiction. ■

B Proof of the analog of Lemma 3.2 for harmonic task sets

The proof here is essentially a reproduction of the proof of Lemma 3.2 from Section 3.1, appropriately modified to take advantage of the unique additional properties of harmonic task sets. In this, recall that τ is a harmonic task set (and hence so is $\tau^{(k)}$). In particular, note that for all $\tau_j \in \tau^{(k)}$, it is the case that $\lceil T_k/T_j \rceil \ = \ T_k/T_j$.

Let us assume that the first $(\ell-1)$ jobs of τ_k have met their deadlines under Algorithm RM, and consider the ℓ'th job of τ_k. This job arrives at time-instant $(\ell-1)T_k$, has a deadline at time-instant ℓT_k, and needs C_k units of execution. As long as OPT schedules τ^k such that the tasks complete no later than their deadlines, the amount of work done by OPT in a time interval $(\ell-1)T_k$ is the utilization of the task set multiplied by the time interval. When processors have the speed $(\frac{m}{2m-1})$, the tasks complete no later than their deadlines according to Lemma 3.1. Therefore we have:

$$W(OPT, m, \frac{m}{2m-1}, \tau^k, t) = t \cdot (\sum_{j=1}^{k} U_j)\ .$$

Applying Inequality 4 yields:

$$W(\mathsf{RM}, m, 1, \tau^{(k)}, (\ell-1)T_k) \geq (\ell-1)T_k \left(\sum_{j=1}^{k} U_j\right) \tag{17}$$

Also, at least $(\ell-1)\cdot T_k \cdot (\sum_{j=1}^{k-1} U_j)$ units of this execution was of tasks $\tau_1, \tau_2, \ldots, \tau_{k-1}$ — this follows from the fact that exactly $(\ell-1)T_kU_k$ 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

$$\begin{aligned} & \sum_{j=1}^{k-1} \left\lceil \frac{\ell T_k}{T_j} \right\rceil C_j \\ < & \sum_{j=1}^{k-1} \frac{\ell T_k}{T_j} C_j \\ = & \ell T_k \sum_{j=1}^{k-1} U_j \end{aligned} \tag{18}$$

As we have seen above (the discussion following Inequality 17) 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

$$T_k \sum_{j=1}^{k-1} U_j \tag{19}$$

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

$$m \cdot T_k - \left(T_k \sum_{j=1}^{k-1} U_j \right) \tag{20}$$

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 $(m \cdot T_k - \left(T_k \sum_{j=1}^{k-1} U_j \right))/m$, which equals

$$T_k - \frac{1}{m} \left(T_k \sum_{j=1}^{k-1} U_j \right) \tag{21}$$

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.,

$$\begin{aligned} & T_k - \frac{1}{m}(T_k \sum_{j=1}^{k-1} U_j) \geq C_k \\ \equiv & \quad \frac{C_k}{T_k} + \frac{1}{m}(\sum_{j=1}^{k-1} U_j) \leq 1 \\ \equiv & \qquad \textit{(Since } T_k \geq T_j \textit{ for } j < k\textit{)} \\ & \quad U_k + \frac{1}{m}(\sum_{j=1}^{k-1} U_j) \leq 1 \end{aligned} \tag{22}$$

Let us now simplify the lhs of Inequality 22 above:

$$\begin{aligned} & U_k + \frac{1}{m}(\sum_{j=1}^{k-1} U_j) \\ \leq \quad & U_k + \frac{1}{m}(\sum_{j=1}^{k} U_j - 2U_k) \\ \leq \quad & \\ & U_k(1 - \frac{1}{m}) + \frac{m}{2m-1} \\ \leq \quad & \\ & \frac{m}{2m-1}(1 - \frac{1}{m}) + \frac{m}{2m-1} \\ = \quad & 1 \end{aligned} \tag{23}$$

From Inequalities 22 and 23, we may conclude that the ℓ'th job of τ_k does meet its deadline.

C Graphical depiction of experimental results

The results of the experiments described in Section 5.2 are graphed in Figures 1 through 6 below. 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).

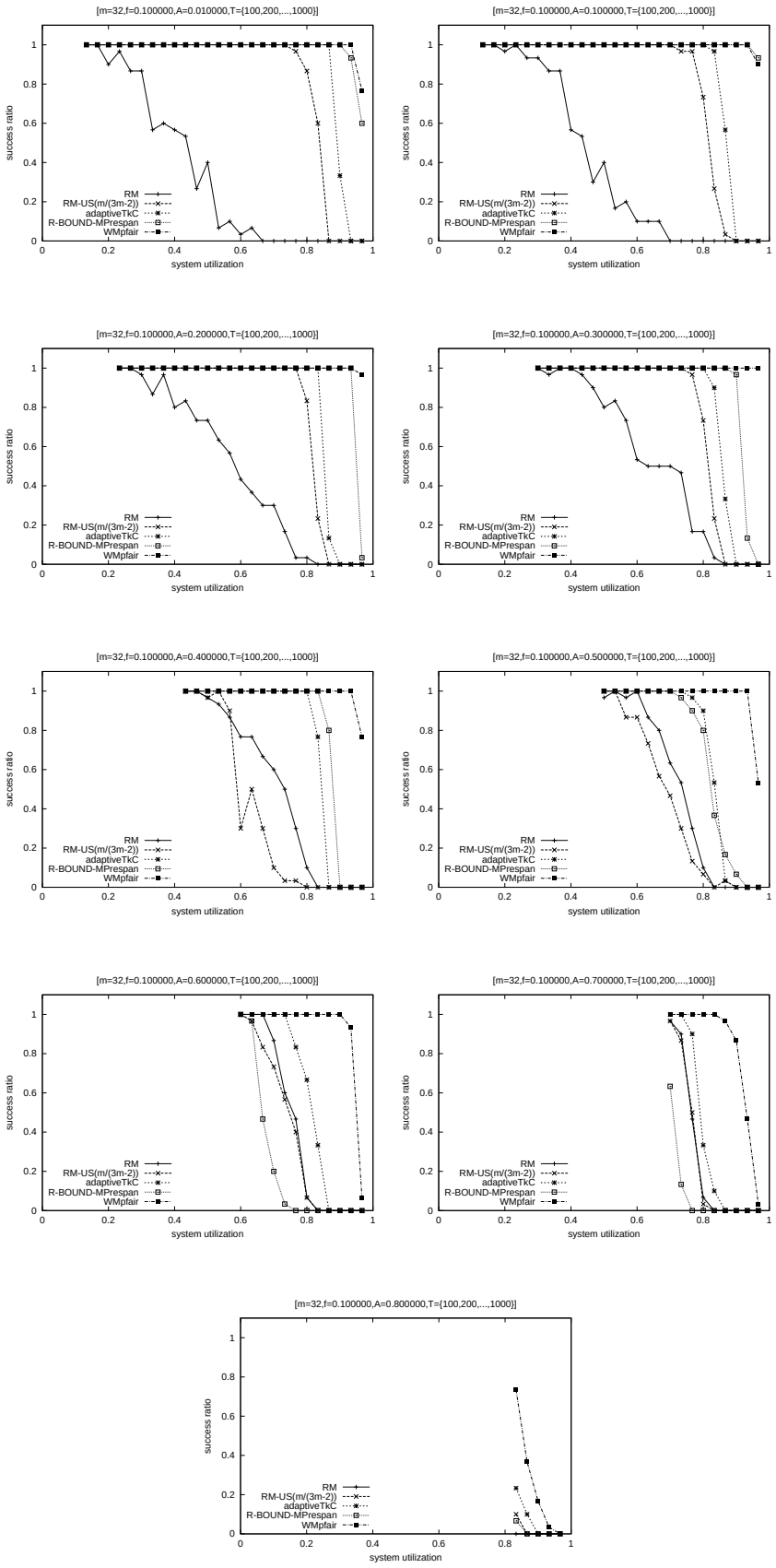

Figure 1: Success ratio as a function of system utilization for different scheduling algorithms on 32 processors, when F=0.1 and periods are selected as T = {100, 200, 300, 400, ..., 1000 }.

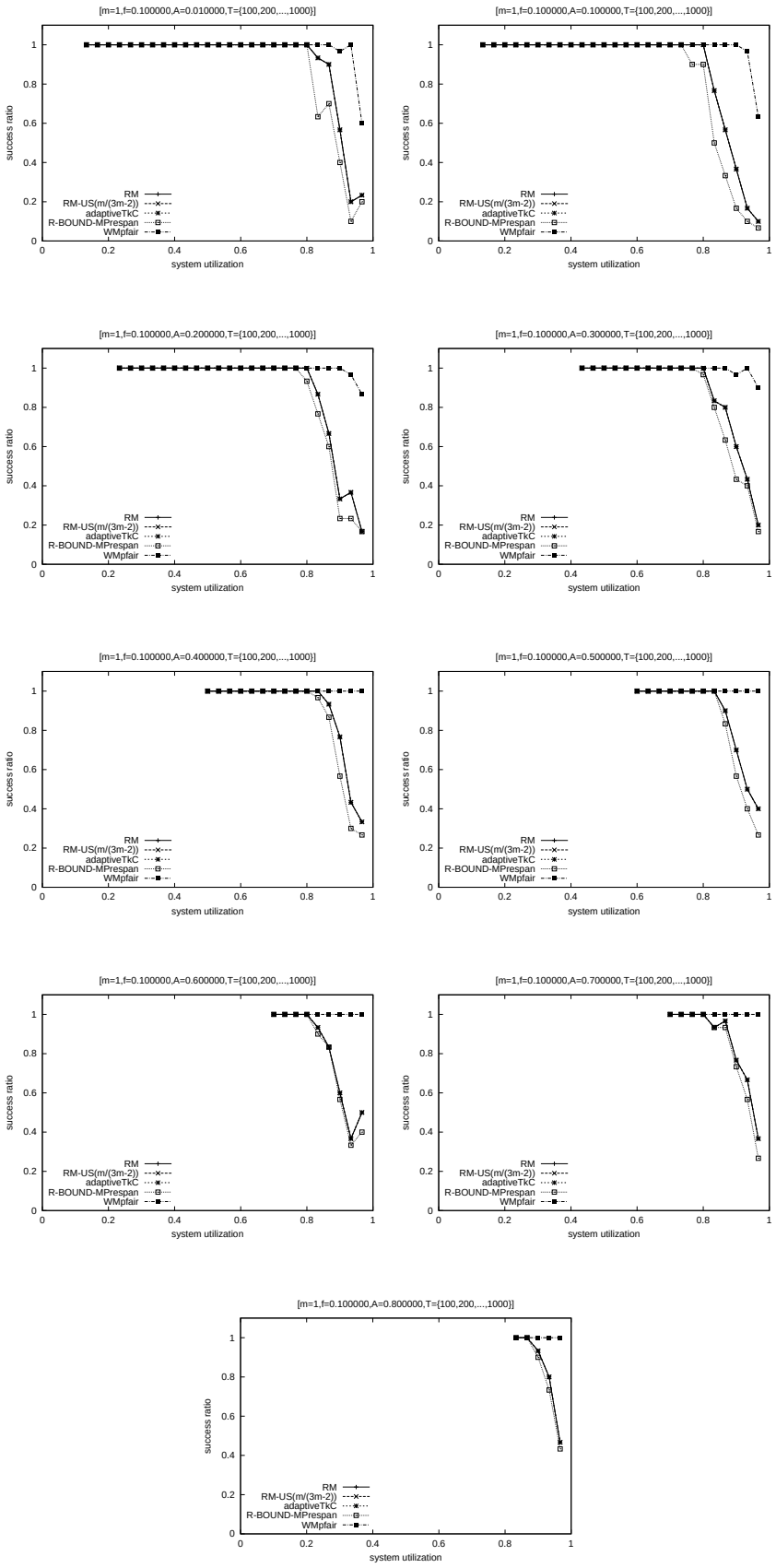

Figure 2: Success ratio as a function of system utilization for different scheduling algorithms on 1 processor, when F=0.1 and periods are selected as T = {100, 200, 300, 400, ..., 1000 }.

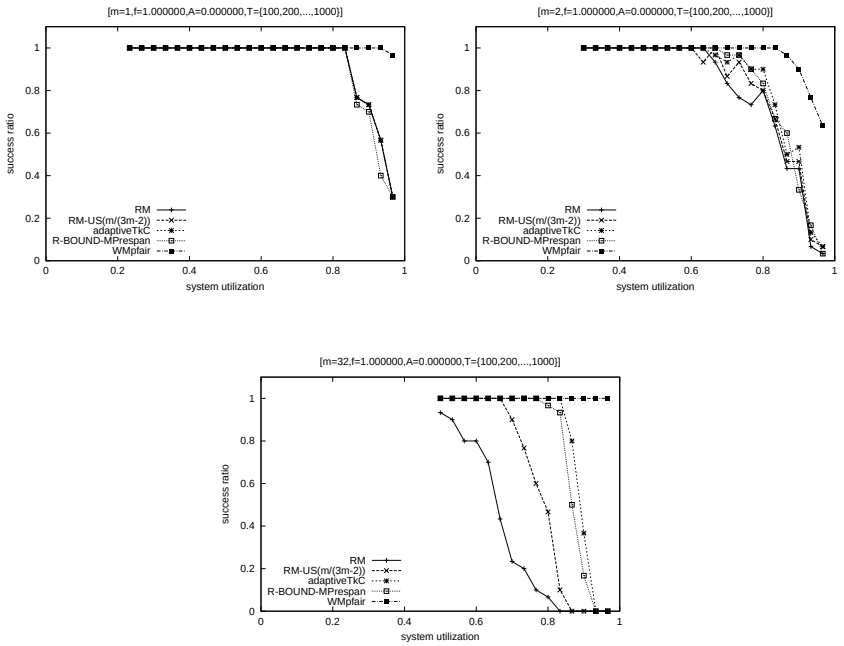

Figure 3: Success ratio as a function of system utilization for different scheduling algorithms on 1,2 and 32 processor(s), when F=1 and periods are selected as T = {100, 200, 300, 400, ..., 1000 }.

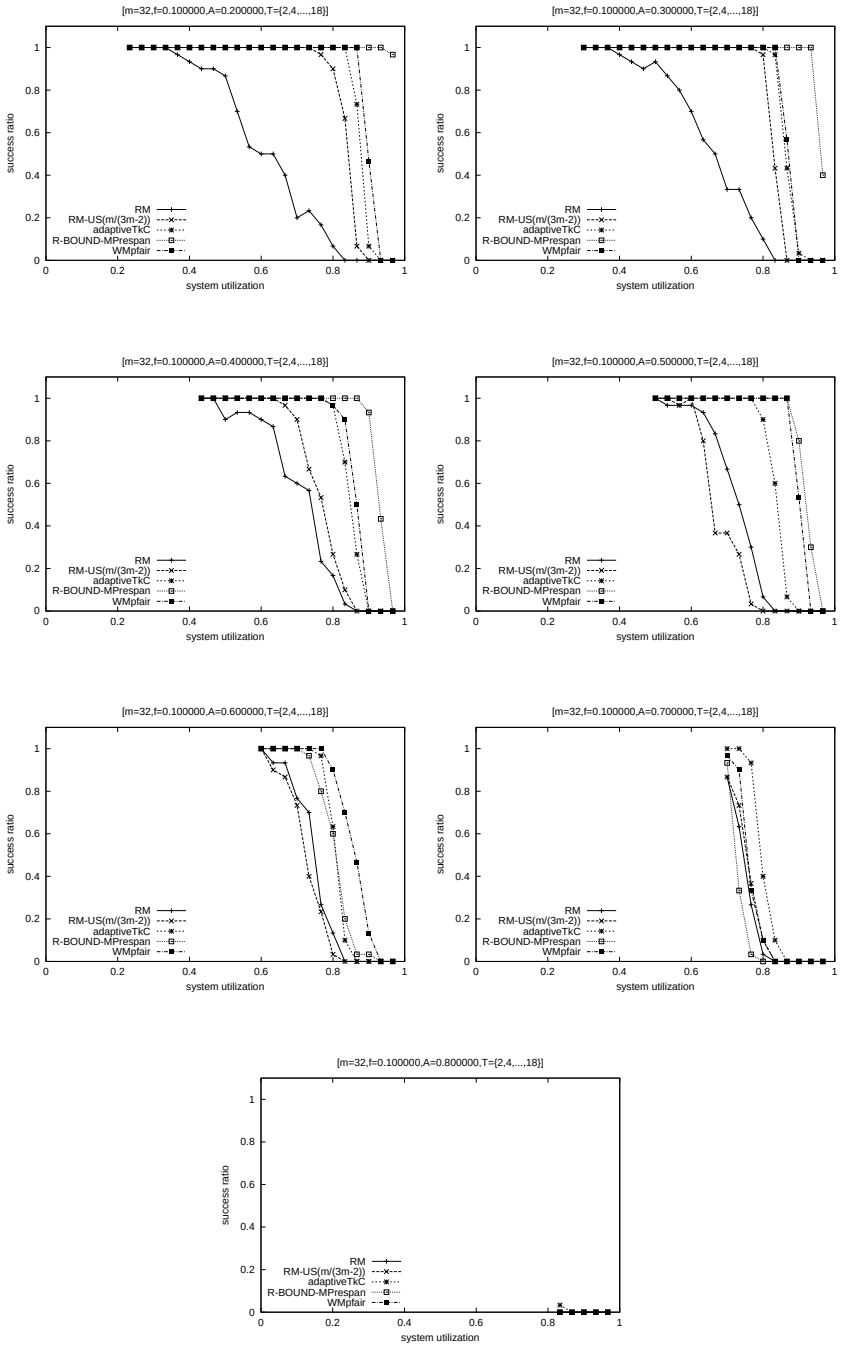

Figure 4: Success ratio as a function of system utilization for different scheduling algorithms on 32 processors, when F=0.1 and periods are selected as T = {2, 4, 6, 8, ..., 20 }.

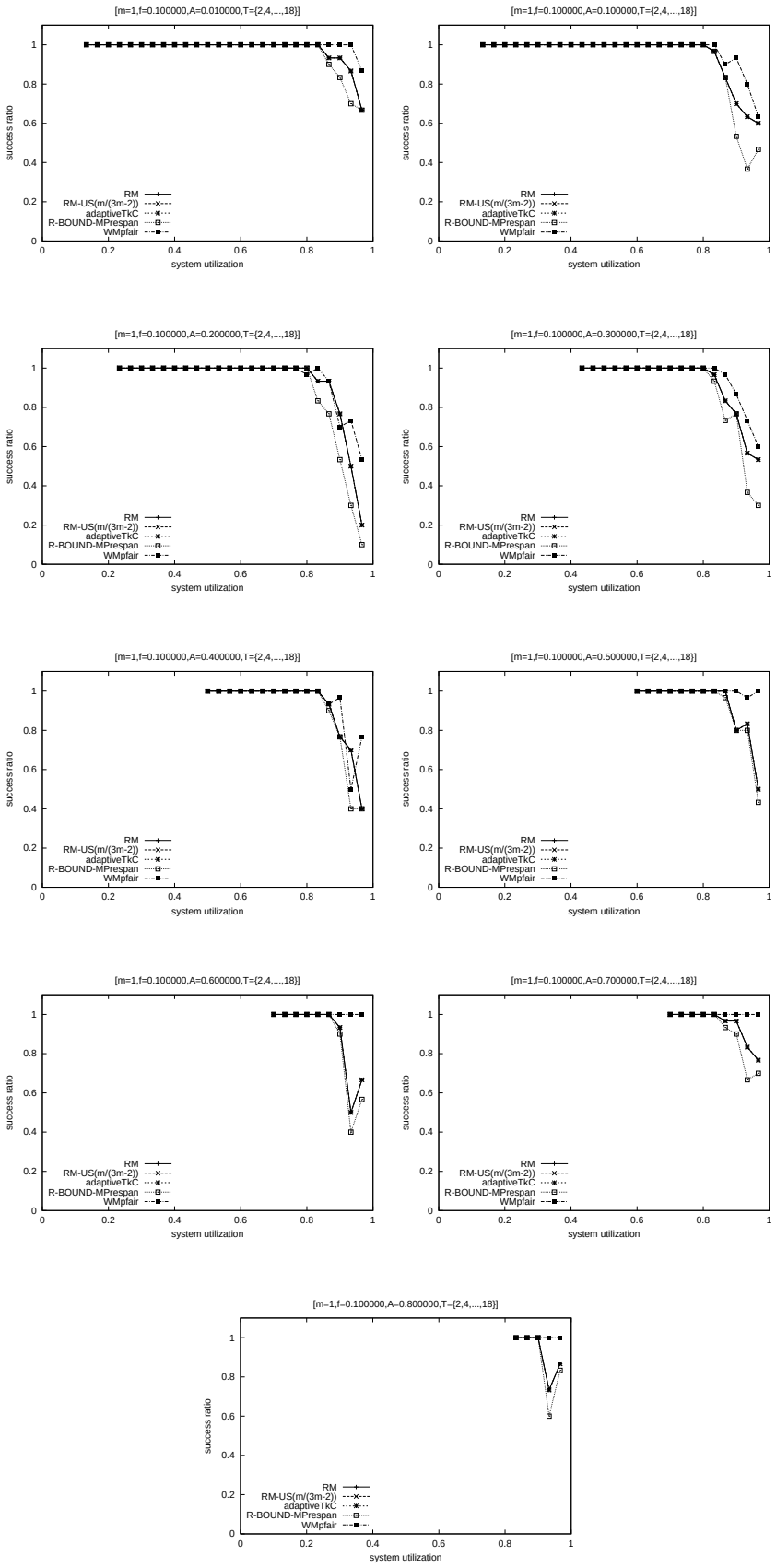

Figure 5: Success ratio as a function of system utilization for different scheduling algorithms on 1 processor, when F=0.1 and periods are selected as T = $\{2, 4, 6, 8, ..., 20\}$.

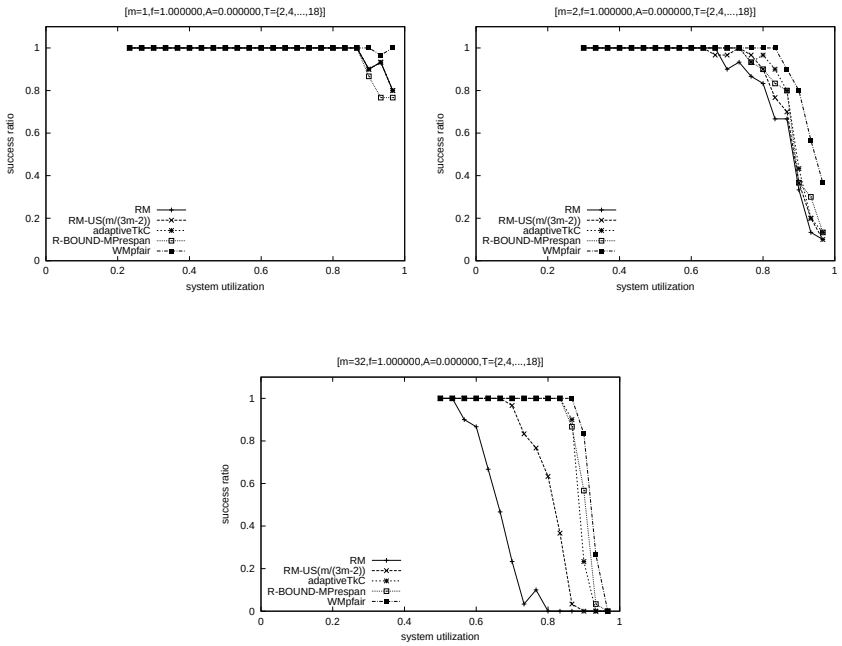

Figure 6: Success ratio as a function of system utilization for different scheduling algorithms on 1,2 and 32 processor(s), when F=1 and periods are selected as T = {2, 4, 6, 8, ..., 20 }.