Preemptive Multiprocessor Scheduling Anomalies*

Björn Andersson and Jan Jonsson

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

Abstract

Preemptive scheduling of periodically arriving tasks on a multiprocessor is considered. We show that many common multiprocessor real-time scheduling algorithms suffer from scheduling anomalies, that is, deadlines are originally met, but a decrease in execution times or an increase in periods of tasks can cause deadlines to be missed. We propose a partitioned multiprocessor fixed-priority scheduling algorithm with the prominent features that (i) it does not suffer from such scheduling anomalies and (ii) if less than 41% of the capacity is used then deadlines are met.

1 Introduction

Analysis techniques for real-time systems often require exact knowledge of task characteristics, but this is usually not available, for example: the execution time of a task depends on input data (which is unknown) or the arrival time of a task depends on when an external event occurs (which is unknown). Fortunately, upper and lower bounds are often known, so in order to give guarantees that deadlines are met, an often-used approach is to make assumptions. For example: (i) assume that a task meets its deadline if it did so when all tasks executed at their maximum execution time, or (ii) assume that a task meets its deadline if it did so when all tasks arrived at their maximum arrival frequency. Situations where these assumptions do not hold are referred to as scheduling anomalies, and their existence jeopardizes timeliness or complicates the design process.

Anomalies neither occur in popular preemptive uniprocessor scheduling algorithms, such as rate-monotonic (RM) and earliest-deadline-first (EDF) [1], nor in multiprocessor systems where tasks are partitioned (statically assigned to processors and not allowed to migrate) and each processor uses an anomaly-free uniprocessor scheduling algorithm. Anomalies can occur in both uniprocessor and multiprocessor systems [2, 3, 4, 5, 6, 7] due to non-preemptive scheduling or due to restricted task migration because decreasing the execution time of a task changes the schedule and that can constrain future scheduling choices. Hence, if tasks were allowed to be preempted at any time and allowed to migrate during its execution then it would be tempting to believe that anomalies do not occur because future scheduling choices are not constrained. However, it was recently found [8] that period anomalies can occur even for scheduling algorithms that allow both preemption and migration, thus indicating the potential existence of anomalies in a large class of multiprocessor scheduling algorithms.

In this paper, we study execution-time and period anomalies in preemptive multiprocessor scheduling algorithms. Our objective is to find anomalies and avoid them without introducing too much additional pessimism in the analysis. To that end, we make the following contributions:

C1. We show that many existing preemptive multiprocessor scheduling algorithms suffer from execution-time or period anomalies. No observations have previously been reported for scheduling algorithms that permit both preemption and migration. The existence of anomalies are known in bin-packing problems [9] and because of similarities between bin-packing and multiprocessor real-time scheduling [10] one could believe that anomalies can also occur in partitioned fixed-priority preemptive scheduling. However, we show that there are additional reasons why anomalies can occur, thereby making anomalies more frequent than indicated by previous research.

C2. We propose a new partitioned multiprocessor fixed-priority scheduling algorithm that does not suffer from anomalies. No previous fixed-priority multiprocessor scheduling algorithm has been proven to avoid anomalies. Our new algorithm, called RM-DU-NFS, partitions tasks using a variant of next-fit bin-packing and applies fixed-priority scheduling on each processor. Although the idea of DU (decreasing utilization [11]) and NFS (Next-Fit Scheduling [10]) have been used before, they have not been combined into one algo-

*Supported by the Swedish Foundation for Strategic Research via the national Swedish Real-Time Systems research initiative ARTES.
rithm, something that we show is important to avoid anomalies while at the same time provide a high system utilization bound (defined in Section 2). Anomaly-free next-fit bin-packing has been studied previously [12], but it has not been applied in real-time scheduling and if it would be applied in a straightforward way, it would not perform well (see Section 4).

C3. We prove that the system utilization bound of RM-DU-NFS is $\sqrt{2} - 1$. No such bound of a next-fit partitioning scheme has previously been proven; earlier work on utilization bounds of partitioned fixed-priority scheduling [11, 13] only studied first-fit and best-fit.

The remainder of this paper is organized as follows. Section 2 defines concepts and system models used. Section 3 shows examples of anomalies in preemptive multiprocessor scheduling. Section 4 discusses strategies for avoiding anomalies, and Section 5 describes our new algorithm. Section 6 discusses the results of the paper.

2 Concepts and System model

We consider the problem of scheduling a task set $\tau = \{\tau_1, \tau_2, \ldots, \tau_n\}$ of $n$ independent\(^1\), periodically-arriving real-time tasks on $m$ identical processors. A task $\tau_i$ arrives periodically with a period of $T_i$. Each time a task arrives, a new instance of the task is created. We denote the $k^{th}$ instance of the task by $\tau_{i,k}$, where $k \in N$. Each instance of $\tau_i$ has an execution time of $C_i$. Each $\tau_i$ has a relative deadline $D_i$. If $D_i$ is not explicitly stated, it is assumed that $D_i = T_i$. The task $\tau_i$ arrives for the first time at time $s_i$. We assume that $T_i, C_i$ are positive real numbers and $s_i$ are real numbers. With no loss of generality we can assume $\min_{i=1 \ldots n} s_i = 0$. A task set is synchronous if $s_1 = s_2 = \ldots = s_n$. A task set is asynchronous if $s_i$ is not constrained by $s_1 = s_2 = \ldots = s_n$. In order to make our results as general as possible, we will, unless otherwise stated, assume asynchronous task sets.

Based on how the system resumes a task’s execution, two different methods can be used namely, the partitioned method and global scheduling. For the partitioned method using fixed-priority scheduling, the system behaves as follows. Each task is assigned to a processor, and then assigned a local (for the processor), unique and fixed priority. With no loss of generality, we assume that the tasks on each processor are numbered in the order of decreasing priority, that is, $\tau_1$ has the highest priority. On each processor, the task with the highest priority of those tasks which has arrived, but not completed, is executed. For the partitioned method, let $\tau^p$ denote the tasks that are assigned to processor $p$ and let $n_p$ denote the number of tasks that are assigned to processor $p$. The task $\tau_i$ is assigned to processor $\text{assigned}_{\text{processor}}(\tau_i)$. For global scheduling using fixed-priority scheduling, the system behaves as follows. Each task is assigned a global, unique and fixed priority. With no loss of generality, we assume that the tasks in $\tau$ are numbered in the order of decreasing priority, that is, $\tau_1$ has the highest priority. Of all tasks that have arrived, but not completed, the $m$ highest-priority tasks are executed\(^2\) in parallel on the $m$ processors. The behavior of dynamic-priority scheduling is similar to the behavior of fixed-priority scheduling with the obvious exception that, in dynamic-priority scheduling, the priorities of tasks can change at any time. A scheduling algorithm is optimal if that algorithm misses a deadline only if there does not exist a schedule that meets the deadlines. Analogously, in fixed-priority scheduling, a priority-assignment scheme is optimal if that scheme misses a deadline only if there does not exist a priority assignment that meets the deadlines.

The utilization $u_i$ of a task $\tau_i$ is $u_i = C_i/T_i$. The utilization $U$ of a task set is $U = \sum_{i=1}^{n} u_i$. Since we consider scheduling on a multiprocessor system, the utilization is not always indicative of the load of the system because the original definition of utilization is a property of the task set only, and does not consider the number of processors. We use the concept of system utilization, $U_s = U/m$. A task is schedulable iff all its instances complete no later than their deadlines. A task set is schedulable iff all its tasks are schedulable. A system utilization bound is a number such that all task sets with a system utilization that is lower than or equal to that number are schedulable.

We will make the following assumptions regarding the task execution model. Tasks can always be preempted, and there is no cost of a preemption, even if the task resumed on another processor. Tasks do not require exclusive access to any other resource than a processor.

3 Examples of anomalies

This section shows that anomalies can occur in many existing preemptive multiprocessor scheduling algorithms. For different scheduling algorithms there are different reasons why anomalies occur. However, if many scheduling algorithms are similar, and the cause for their anomalies are the same, we only present one example.

Period anomalies in global scheduling One reason why anomalies can occur in global scheduling is that an increase in period causes tasks to arrive at different times. These different times do not affect schedulability directly, and the schedule generated when the period increases, performs less work on the processors. However, the execution can be distributed differently. This change in distribution of execution

\(^1\)That is, the arrival time of a task does not depend on the time when other tasks were executed.

\(^2\)At each instant, the processor chosen for each of the $m$ tasks is arbitrary. If less than $m$ tasks should be executed simultaneously, some processors will be idle.
causes more instants when all processors are busy and that delays a lower priority tasks even more. This can happen in global fixed-priority scheduling \[8\], but as Observation 1 points out, such anomalies can also happen in dynamic priority scheduling for EDF. At every instant \(t\), EDF assigns the highest priority to the task with the closest (in time) deadlines, for example assigning the priority \(1/(d_i - t)\) to the tasks that have arrived but not completed. Here \(d_i\) is \(a_i + D_i\), where \(a_i\) is the absolute time of the largest arrival time of task \(t_i\) that is not greater than \(t\).

**Observation 1** In global EDF, there exist schedulable asynchronous task sets such that if the period of a task increases, a task misses a deadline.

**Example 1** Consider the following three periodic tasks: \((T_1 = 4, C_1 = 1), (T_2 = 5, C_2 = 1), (T_3 = 20, C_3 = 18)\) to be scheduled using global EDF on two processors. The task set is schedulable for \(s_1 = s_2 = s_3\) (Figure 1(a)) and also when \(s_1\) is arbitrary \([14]\). However, if \(T_1\) is increased to \(5\), then a deadline is missed (Figure 1(b)).

Global EDF is often considered a dynamic priority scheduling algorithm; it is in fact a job-fix priority scheduling algorithm, since at every time a task arrives, the priority ordering of tasks is unchanged until a new task arrives. Corollary 3.1 in \([7]\) states that job-fix priority scheduling does not suffer from execution-time anomalies. Hence, EDF does not suffer from execution-time anomalies. Unfortunately, the system utilization bound of EDF is \(0\) \([14]\).

A similar but different reason why period anomalies can occur in global scheduling is that an increase in period causes tasks to arrive at different times. These different times makes tasks to perform less work on the processors, and the execution is not distributed so that all processors are busy at the same time more frequently. However, just the fact that the arrival times are different causes a task to miss its deadline. This can happen even in global optimal scheduling, when \(D_i < T_i\). (For \(D_i = T_i\), anomalies cannot occur in global optimal scheduling, see Section 4.)

**Observation 2** In global optimal scheduling, where the deadline \(D_i < T_i\), there exist schedulable synchronous task sets such that if the period of a task increases, a task misses a deadline.

**Example 2** Consider the following three periodic tasks: \((T_1 = 4, D_1 = 2, C_1 = 1), (T_2 = 5, D_2 = 3, C_2 = 3), (T_3 = 10, D_3 = 8, C_3 = 7)\) to be scheduled using global optimal scheduling on two processors. The tasks are schedulable if \(s_1 = s_2 = s_3\) (Figure 2(a)). However, if \(T_1\) is increased to \(5\) the resulting task set is no longer schedulable because it is not possible to construct any schedule that makes the tasks to meet their deadlines. The reason is that \(T_2\) must execute immediately when it arrives, because otherwise \(T_2\) would miss a deadline. \(T_1\) must execute \(1\) time unit within \([0, 2]\) and \(1\) time unit within \([5, 7]\). Figure 2(b) illustrates the situation when \(T_2\) starts to execute at time \(0\) and at time \(5\). Regardless of when \(T_1\) executes within these intervals, the two first instances of \(T_1\) will execute at the same time as \(T_2\) executes. That is within the interval \([0, 8]\) there are at least \(2\) time units when both \(T_1\) and \(T_2\) executes. That is during \([0, 8]\) there are \(6\) time units or less available for \(T_3\) to execute. But \(T_3\) needs to execute \(7\) time units in the interval \([0, 8]\). Hence \(T_3\) misses its deadline.

Note that although an unschedulable synchronous task set is also an unschedulable asynchronous task set, the fact that a scheduling algorithm suffers from anomalies of a synchronous task set does not necessarily imply that there exist asynchronous task sets that suffer from anomalies.

**Period-based anomalies in bin-packing schemes** With the partition method, bin-packing is a common technique for assigning tasks to processors (see for example \([10, 13, 11]\)). All partitioning schemes that we will discuss use bin-packing. Bin-packing algorithms work as follows: (1) sort the tasks according to some criterion; (2) select the first task and an arbitrary processor; (3) attempt to assign the selected task to the selected processor by applying a schedulability test for the processor; (4) if the schedulability test fails, select the next available processor; if it succeeds, select the next task; (5) goto step 3. Partitioning schemes based on bin-packing do never miss deadlines, but they declare failure because a schedulability test in the scheduling algorithm cannot guarantee the task to meet deadlines. For that reason, we will say that the scheduling algorithm suffers from an anomaly if there is a task set that the algorithm declares failure for, but there is at least one task such that if its utilization is decreased, then the algorithm declares failure.

The original bin-packing problem did not address processors and tasks, but rather putting items in bins, where items corresponds to tasks and bins correspond to processors. For systems, where bin sizes do not depend on the item sizes, the existence of bin packing anomalies has been shown for first-fit and first-fit decreasing \([9]\). Since EDF scheduling on a uniprocessor has a utilization bound of \(1\), which does not depend on the task set, there clearly exist anomalies for partitioned EDF. In the remainder of this section, we will discuss fixed-priority scheduling using partitioning.

One reason for the anomaly in bin-packing is that if the period increases, the schedulability test used becomes more pessimistic. That reason cannot happen if the schedulability test is a utilization-based test, where the utilization bound does only depend on the number of processors (and hence does not depend on execution times or periods). How-
Consider the following three periodic tasks: Example 3

If the period of a task increases, a task is not guaranteed to meet its deadline. However, if we increase the period of $\tau_1$ from 4 to 5 the resulting task set is no longer guaranteed by R-BOUND-MP to be schedulable. To see this, we can run the algorithm R-BOUND-MP. The task set is transformed to $(T_1 = 4, C_1 = 4), (T_2 = 4, C_2 = 2), (T_3 = 5, C_3 = 2)$. When assigning $\tau_1$ and $\tau_2$, the algorithm R-BOUND-MP behaves as previously, that is $\tau_1$ is assigned to processor $P_1$ and $\tau_2$ is assigned to processor $P_2$. Then $\tau_3$ is attempted to be assigned to processor $P_1$ and that attempt fails, so $\tau_3$ is attempted to be assigned to processor $P_2$. Now R-BOUND-MP behaves differently. The schedulability test fails because the ratio between periods are $5/4 = 1.25$, thereby R-BOUND can only guarantee a task set which have a utilization that is no greater than 85%. The utilization of $\tau_2$ and $\tau_3$ is 90%. Hence R-BOUND-MP cannot guarantee the task set to be schedulable.

It would be tempting to think that if a necessary and sufficient schedulability test is used then these anomalies cannot occur. However, anomalies can still occur for partitioning schemes that sort tasks according to periods, because when periods are changed, the order of how tasks are assigned to
processor does also change. One such technique is RMFFS improved by using a necessary and sufficient schedulability test. RMFFS [10] is a first-fit bin-packing algorithm that originally used a schedulability test that was similar to a utilization based test. Since we use a necessary and sufficient schedulability test, we can be sure that if the partitioning scheme fails, then a task will actually misses a deadline.

**Observation 4** For the partitioning scheme, RMFFS improved by using a necessary and sufficient schedulability test, there exist task sets that can be guaranteed to meet its deadline, but if the period of a task increases, a task misses its deadline.

**Example 4** Consider the following four periodic tasks: 
$(T_1 = 2, C_1 = 1), (T_2 = 3, C_2 = 2), (T_3 = 6, C_3 = 3), (T_4 = 7, C_4 = 1)$. RMFFS will first sort the task set according to its periods. That yields the task set: $(T_1 = 2, C_1 = 1), (T_2 = 3, C_2 = 2), (T_3 = 7, C_4 = 1), (T_3 = 8, C_3 = 3)$. $\tau_1$ will be assigned to processor $P_1$ and $\tau_2$ will be assigned to processor $P_2$. $\tau_3$ is assigned to processor $P_1$, but $\tau_3$ cannot be assigned to processor $P_1$ (because then the utilization would be 1.017, and $\tau_3$ cannot be assigned to processor $P_2$.

It turns out that all previously published partitioning schemes for fixed-priority preemptive scheduling, suffers from period anomalies as long as repartitioning is done when the task set changes. The reason for the anomalies are the two reasons given so far, or a similar reason as the execution-time anomaly in the next paragraph.

**Execution-time anomalies in bin-packing** Because a decrease in execution time of a task can make the schedulability test to succeed, when it would otherwise fail, the partitioning can become different when subsequent tasks are assigned to processors, making the task set to miss deadlines.

**Observation 5** For the partitioning scheme, RMFFS improved by using a necessary and sufficient schedulability test, there exist task sets that can be guaranteed to meet its...
deadline, but if the execution time of a task decreases, a task misses its deadline.

**Example 5** Consider the following four periodic tasks: $(T_1 = 5, C_1 = 3), (T_2 = 7, C_2 = 4), (T_3 = 8, C_3 = 3), (T_4 = 10, C_4 = 4)$ to be scheduled on two processors using RMFFS improved by using a necessary and sufficient schedulability test. Tasks are sorted according to their periods. That does not change the task set. $\tau_1$ will be assigned to processor $P_1$ and $\tau_2$ will be assigned to processor $P_2$. $\tau_3$ is tested to be assigned to processor $P_1$, but fails. $\tau_3$ is then tested to be assigned to processor $P_2$, and succeeds. $\tau_4$ is tested to be assigned to processor $P_1$, and succeeds.

If the execution time of $\tau_1$ is decreased by 1, we obtain the task set: $(T_1 = 5, C_1 = 2), (T_2 = 7, C_2 = 4), (T_3 = 8, C_3 = 3), (T_4 = 10, C_4 = 4)$. Tasks are sorted according to their periods. That does not change the task set. $\tau_1$ will be assigned to processor $P_1$ and $\tau_2$ will be assigned to processor $P_2$. $\tau_3$ is tested to be assigned to processor $P_1$, and succeeds. $\tau_4$ is tested to be assigned to processor $P_1$, but fails. Then $\tau_4$ is tested to be assigned to processor $P_2$, but fails again. \( \square \)

Similar execution-time anomalies can occur for many other partitioning schemes [14].

**4 Solutions**

Having observed that anomalies can occur for preemptive multiprocessor scheduling, the question arises regarding how to deal with them. We conceive the following approaches:

- Perform system adjustments such that anomalies cannot occur. For example, if the system suffers from period anomalies, but it does not suffer from execution-time anomalies, then we can perform system adjustments on execution times.

- Use a scheduling algorithm that is designed to dynamically detects anomalies and avoid them.

- Accept only such task sets that cannot suffer from anomalies.

- Use a scheduling algorithm that is designed so that anomalies cannot occur.

Since the first approach transfers the problem of anomalies to another parameter, we only discuss the last three approaches below.

**Designed to detect anomalies** When designing algorithms to determine whether a certain property holds it is often the case that only solutions that are of the same granularity (a multiple of the greatest common divisor) as parameters describing the problem instance need to be explored. If that assumption holds, and if the the parameters describing the problem instance are bounded, then the number of computational steps (computational complexity) of an algorithm is bounded, by simply enumerating all combinations (which are bounded). Unfortunately, such an approach is not always possible in anomaly detection (Example 6).

**Example 6** For global fixed-priority preemptive scheduling, the following asynchronous task set is schedulable $(T_1 = 8, C_1 = 4), (T_2 = 20, C_2 = 12), (T_3 = 32, C_3 = 20)$ on two processors, assuming global RM. Any combination of increases in period or decreases in execution times that are multiples of 4, causes the task set to be schedulable. However increasing $T_1$ by a value less than the granularity (for example, 1) makes the task set unschedulable. This example is also applicable to optimal priority assignment.

As the example shows, it seems difficult to design a necessary and sufficient condition to determine whether a task set suffers from scheduling anomalies. However, as will be described below, sufficient conditions for anomaly-free task sets are easier to design.

**Accept tasks** If a scheduling algorithm has a utilization bound, then task sets with a utilization lower than or equal to the utilization bound can neither suffer from execution-time anomalies nor period anomalies. Consequently, in order to avoid anomalies, accept only task sets that have a utilization lower than or equal to this bound. Unfortunately, such an approach introduces additional pessimism, since there are anomaly-free task sets with a utilization that is higher than the bound.

**Designed to avoid anomalies** We conceive three ways of designing scheduling algorithms to avoid anomalies: optimal scheduling, no repartitioning and anomaly-free repartitioning.

Scheduling algorithms that are optimal for $D_i = T_i$ have the property that $\sum_{i=1}^{n} \frac{C_i}{T_i} \leq m \Rightarrow \text{schedulable}$. Obviously, such algorithms do not suffer from anomalies. However, as we saw in Observation 2, scheduling algorithms that are optimal for $D_i \neq T_i$ can still suffer from anomalies.

Scheduling algorithms that partition a task set and apply an anomaly-free uniprocessor scheduling algorithm on each processor but do not repartition the task set when the task set changes, will not suffer from anomalies because the system behaves as a set of uniprocessors. Unfortunately, the system utilization bound of such an approach is zero [14].

Anomaly-free bin-packing can be achieved using a next-fit policy [12]. It is easy to see that such a policy can also be used in real-time scheduling for partitioning of tasks that are scheduled by EDF on each uniprocessor because in EDF each processor have a utilization bound of 1 and hence can
Lemma 1 RM-DU-NFS declares success for a given $\tau \iff$ there exists an assignment of $\tau$ that is SD.

Proof: Follows directly from the algorithm. $\Box$

Lemma 2 Consider two task sets $\tau_{before}$ and $\tau_{after}$, where $\tau_{after}$ differs from $\tau_{before}$ only in that there is a task $\tau_i$ such that $u_i^{after} \leq u_i^{before}$. If there exists an assignment of $\tau_{before}$ that is SD then there exists an assignment of $\tau_{after}$ that is SD.

Proof: If tasks in $\tau_{after}$ have the same assignment as tasks in $\tau_{before}$, then for all $p \in [1..m]$: $\sum_{\tau_j \in \tau_{after}, \tau_j \neq \tau_i} C_j / T_j \leq n_p \cdot (2^{\lceil \ln 2 \rceil} - 1)$. However, with such an assignment of $\tau_{after}$, $\tau_{after}$ is not necessarily SD, because it could be that $u_i^{after}$ has a lower utilization than tasks assigned a processor with higher index. We will now show that by swapping the assignment of $\tau_{after}$ tasks, it is possible to achieve an assignment of $\tau_{after}$ that is SD.

Consider those tasks $\tau_j \in \tau_{before}$ that satisfy $u_j^{before} \geq u_j^{after}$. Those tasks are assigned to the processors $P_k, \ldots, P_l$. For each of the processors $P_g \in \{P_{k+1}, \ldots, P_l\}$, move the task with the highest utilization from processor $P_g$ to $P_{g-1}$. Also move task $\tau_j^{before}$ from $P_k$ to processor $P_l$. The number of tasks on each processor is unaffected, so the utilization bounds of each processor are unaffected. The utilization of tasks on each of the processors $P_g \in \{P_k, \ldots, P_l\}$ do not increase, and for the other processors the utilization do not change. This new assignment of $\tau_{after}$ also satisfies $\min_{\tau_j \in \tau_{after}} C_j / T_j \geq \max_{\tau_j \in \tau_{after}} C_j / T_j$. Hence the new assignment of $\tau_{after}$ is SD. $\Box$

Lemma 3 Consider two task sets $\tau_{before}$ and $\tau_{after}$, where $\tau_{after}$ differs from $\tau_{before}$ only in that there is a task $\tau_i$ such that $u_i^{after} \leq u_i^{before}$ if RM-DU-NFS declares success for $\tau_{before}$, then RM-DU-NFS declares success for $\tau_{after}$.

Proof: By applying Lemma 1 and Lemma 2, we can reason as follows:

RM-DU-NFS declares success for $\tau_{before}$. $\Rightarrow$
There exists an assignment of $\tau_{before}$ that is SD. $\Rightarrow$
There exists an assignment of $\tau_{after}$ that is SD. $\Rightarrow$
RM-DU-NFS declares success for $\tau_{after}$. $\Box$

Lemma 4 Consider a task set $\tau$ such that RM-DU-NFS has declared success when $\tau$ is applied. If $u_i$ is decreased for
any subset of $\tau$, then RM-DU-NFS will declare success.

**Proof:** Apply Lemma 3 repeatedly for each task that decreased its utilization.

**Theorem 1** Consider a task set $\tau$ such that RM-DU-NFS has declared success when $\tau$ is applied. If $T_i$ is increased or $C_i$ is decreased for any subset of $\tau$, then RM-DU-NFS will declare success.

**Proof:** Apply Lemma 4.

**Theorem 1** Consider a task set $\tau$ such that RM-DU-NFS has declared success when $\tau$ is applied. If $T_i$ is increased or $C_i$ is decreased for any subset of $\tau$, then RM-DU-NFS will declare success.

**Proof:** Apply Lemma 4.

---

6 Discussion and Future work

We have shown that anomalies can occur in many well-established preemptive multiprocessor scheduling algorithms and also presented a new scheduling algorithm RM-DU-NFS that does not suffer from anomalies. We think that the ideas of RM-DU-NFS can be used in the design of other anomaly-free resource management algorithms.

One question left open by us is whether there are asynchronous task sets that suffer from anomalies in optimal scheduling algorithms assuming that $D_i$ is not restricted by $D_i = T_i$.

**References**


