Semi-Partitioned Fixed-Priority Scheduling on Multiprocessors *

Shinpei Kato and Nobuyuki Yamasaki
Department of Information and Computer Science
Keio University, Yokohama, Japan
{shinpei,yamasaki}@ny.ics.keio.ac.jp

Abstract

This paper presents a new algorithm for fixed-priority scheduling of sporadic task systems on multiprocessors. The algorithm is categorized to such a scheduling class that qualifies a few tasks to migrate across processors, while most tasks are fixed to particular processors. We design the algorithm so that a task is qualified to migrate, only if it cannot be assigned to any individual processors, in such a way that it is never returned to the same processor within the same period, once it is migrated from one processor to another processor. The scheduling policy is then conformed to Deadline Monotonic. According to the simulation results, the new algorithm significantly outperforms the traditional fixed-priority algorithms in terms of schedulability.

1 Introduction

In recent years, the scheduling of real-time tasks has been revised for multiprocessor platforms, given that multicore technologies have proliferated in the marketplace. Real-time scheduling techniques for multiprocessors is mainly classified into *global scheduling* and *partitioned scheduling*. In global scheduling, all tasks are stored in a global queue, and the same number of the highest priority tasks as processors are selected for execution. This scheduling class contains optimal algorithms, such as Pfair [10, 9] and LLREF [12]. Any periodic task systems are scheduled successfully by those algorithms, if the processor utilization does not exceed 100%. In partitioned scheduling, on the other hand, tasks are first assigned to specific processors, and then executed on those processors without migrations. Partitioned scheduling is of advantage in that a problem of multiprocessor scheduling is reduced into a set of uniprocessor one, after tasks are partitioned.

In terms of fixed-priority algorithms that are often adopted by commodity real-time operating systems for practical use, partitioned scheduling may be more attractive than global scheduling. Andersson *et al.* proved in [6] that both fixed-priority Pfair [26] and Rate Monotonic [21] with a partitioning technique presented in [6] may cause deadlines to be missed, if the processor utilization is greater than 50%. To the best of our knowledge, all other fixed-priority algorithms based on global scheduling [3, 8, 2] and partitioned scheduling [13, 25, 24, 23, 22] have lower utilization bounds. Thus, we see little advantage of global scheduling over partitioned scheduling from the viewpoint of fixed-priority algorithms.

Recent work [1, 7, 4, 5, 15, 16] have made another class, called *semi-partitioned scheduling* in this paper, for the purpose of improving schedulability with succeeding the advantage of partitioned scheduling as much as possible. In semi-partitioned scheduling, most tasks are fixed to particular processors to reduce runtime overhead, while a few tasks migrate across processors to improve schedulability. According to the prior work, a class of semi-partitioned scheduling offers a significant improvement on schedulability, as compared to a class of partitioned scheduling, with less preemptions and migrations than a class of global scheduling. However, little work have studied on fixed-priority algorithms.

This paper presents a new algorithm for semi-partitioned fixed-priority scheduling of sporadic task systems on identical multiprocessors. In terms of schedulability, the new algorithm strictly dominates, and in general significantly outperforms, the traditional fixed-priority algorithms based on partitioned scheduling, while the scheduling policy is simplified to reduce the number of preemptions and migrations for practical use.

The rest of paper is organized as follows. The next section reviews prior work on semi-partitioned scheduling. The system model is defined in Section 3. Section 4 then presents a new algorithm based on semi-partitioned scheduling. Section 5 evaluates the effectiveness of the new algorithm. This paper is concluded in Section 6.

*This work is supported by the fund of Research Fellowships of the Japan Society for the Promotion of Science for Young Scientists. This work is also supported in part by the fund of Core Research for Evolutional Science and Technology, Japan Science and Technology Agency.

2 Prior Work

Recent work [1, 7, 4, 5, 15, 16] have made a class of semi-partitioned scheduling, in which most tasks are fixed to particular processors, while a few tasks may migrate across processors, to improve schedulability over partitioned scheduling with a smaller number of preemptions as well as migrations than global scheduling.

The concept of semi-partitioned scheduling is originally introduced by EDF-fm [1]. EDF-fm assigns the highest priority to migratory tasks in a static manner. The fixed tasks are then scheduled according to EDF, when no migratory tasks are ready for execution. Since EDF-fm is designed for soft real-time systems, the schedulability of a task set is not tightly guaranteed, while the tardiness is bounded.

EKG [7] is designed to guarantee all tasks to meet deadlines for implicit-deadline periodic task systems. Unlike EDF-fm, migratory tasks are executed in certain time slots, while fixed tasks are scheduled according to EDF. The achievable processor utilization is traded with the number of preemptions and migrations, by a parameter k. The configuration of $k = m$ on m processors leads EKG to be optimal, with more preemptions and migrations. In the later work [4, 5], EKG is extended for sporadic task systems as well as explicit-deadline systems. The extended algorithms are also parametric with respect to the length of the time slots reserved for migratory tasks.

EDDHP [15] is designed in consideration of reducing context switches. As EDF-fm, the highest priority is assigned to migratory tasks, and other fixed tasks have the EDF priorities, though it differs in that the scheduling policy guarantees all tasks to meet deadlines. It is shown by simulations that EDDHP outperforms the partitioned EDF algorithms in schedulability, with less preemptions than EKG. EDDP [16] is an extension of EDDHP in that the priority ordering is fully dynamic. The worst-case processor utilization is then bounded by 65% for implicit-deadline systems.

RMDP [17] is a fixed-priority version of EDDHP: the highest priority is given to migratory tasks, and other fixed tasks have the Rate Monotonic priorities. It is shown by simulations that RMDP improves schedulability over the traditional fixed-priority algorithms. The worst-case processor utilization is bounded by 50% for implicit-deadline systems. To the best of our knowledge, no other algorithms based on semi-partitioned scheduling consider fixed-priority assignments.

In the previous algorithms mentioned above, tasks may migrate across processors, even though there are no needs of migrations. It would be better to migrate tasks, only if they are required, because migration costs are not free. The scheduling policy for migratory tasks is also problematic. For instance, EKG generates two sets of preemptions and migrations in every interval of job arrivals, to schedule migratory tasks. EDDHP, EDDP, and RMDP may also generate many preemptions and migrations, particularly when migratory tasks have long inter-arrival time. We address those concerns in this paper.

3 System Model

The system is composed of m identical processors P_1, P_2, ..., P_m and n sporadic tasks $T_1, T_2, ..., T_n$. Each task T_i is characterized by a tuple (c_i, d_i, p_i), where c_i is a worst-case computation time, d_i is a relative deadline, and p_i is a minimum inter-arrival time (period). The utilization of T_i is denoted by $u_i = c_i/p_i$. We assume such a constrained task model that satisfies $c_i \leq d_i \leq p_i$ for any T_i. Each task T_i generates an infinite sequence of jobs, each of which has a constant execution time c_i. A job of T_i released at time t has a deadline at time $t + d_i$. Any inter-arrival intervals of successive jobs of T_i are separated by at least p_i.

Each task is independent and preemptive. Any job is not allowed to be executed in parallel. Jobs produced by the same task must be executed sequentially, which means that every job of T_i is not allowed to begin before the preceding job of T_i completes. The costs of scheduler invocations, preemptions, and migrations are not modeled.

4 New Algorithm

We present a new algorithm, called **Deadline Monotonic with Priority Migration (DM-PM)**, based on the concept of semi-partitioned scheduling. In consideration of the migration and preemption costs, a task is qualified to migrate, only if it cannot be assigned to any individual processors, in such a way that it is never returned to the same processor within the same period, once it is migrated from one processor to another processor.

On uniprocessor platforms, Deadline Monotonic (DM) [20] has been known as an optimal algorithm for fixed-priority scheduling of sporadic task systems. DM assigns a higher priority to a task with a shorter relative deadline. This priority ordering follows Rate Monotonic (RM) [21] for periodic task systems with all relative deadlines equal to periods. Given that DM dominates RM, we design the algorithm based on DM.

4.1 Algorithm Description

As the traditional partitioning approaches [13, 25, 18, 6, 14], DM-PM assigns each task to a particular processor, according to kinds of bin-packing heuristics, upon which the schedulable condition for DM is satisfied. In fact, any heuristics are available for DM-PM. If there are no such processors, DM-PM tries to share the task among more than

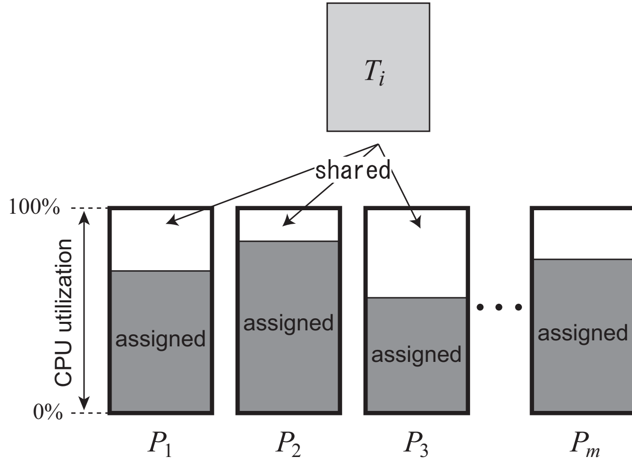


Figure 1. Example of sharing a task.

one processor, whereas a task set is decided to be unfeasible in the partitioning approaches. In the scheduling phase, the shared task is qualified to migrate across the processors among which the task is shared.

Figure 1 demonstrates an example of sharing a task among more than one processor. Let us assume that none of the m processors has spare capacity enough to accept full share of a task T_i. According to DM-PM, T_i is for instance shared among the three processors P_1, P_2, and P_3. In terms of utilization share, T_i is "split" into three portions. The share is always assigned to processors with lower indexes. The execution capacity is then given to each share so that the corresponding processors are filled to capacity. In other words, the processors have no spare capacity to receive other tasks, once a shared task is assigned to them. However, only the last processor to which the shared task is assigned may still have spare capacity, since the execution requirement of the last portion of the task is not necessarily aligned with the remaining capacity of the last processor. Thus, in the example, no tasks will be assigned to P_1 and P_2, while some tasks may be later assigned to P_3. In the scheduling phase, T_i migrates across P_1, P_2 and P_3. We will describe how to compute the execution capacity for each share in Section 4.2.

Here, we need to guarantee that multiple processors never execute a shared task simultaneously. To this end, DM-PM simplifies the scheduling policy as follows.

- A shared task is scheduled by the highest priority within the execution capacity on each processor.
- Every job of the shared task is released on the processor with the lowest index, and it is sequentially migrated to the next processor when the execution capacity is consumed on one processor.
- Partitioned tasks are then scheduled according to DM.

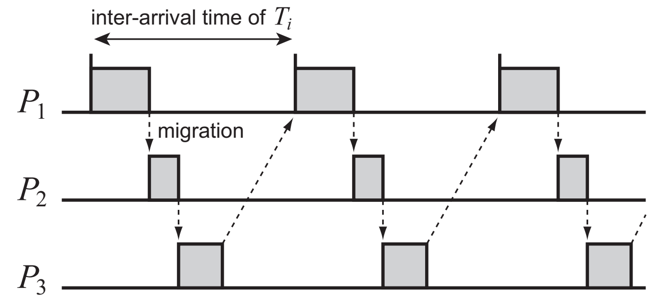


Figure 2. Example of scheduling a shared task

Figure 2 illustrates an example of scheduling a shared task T_i whose share is assigned to three processors P_1, P_2, and P_3. Let $c'_{i,1}$, $c'_{i,2}$, and $c'_{i,3}$ be the execution capacity assigned to T_i on P_1, P_2, and P_3 respectively. Since P_1 is a processor with the lowest index, every job of a T_i is released on P_1. Since T_i has the highest priority, it is immediately executed within the time interval of length $c'_{i,1}$. When $c'_{i,1}$ is consumed, T_i is migrated to the next processor P_2, and then executed by the highest priority within $c'_{i,2}$. T_i is finally migrated to the next processor P_3 when $c'_{i,2}$ is consumed on P_2, and then executed in the same manner.

According to the scheduling policy of DM-PM, the execution of a shared task T_i is repeated exactly at its inter-arrival time on every processor, because it is executed by the highest priority within a time interval of a constant length on each processor. A shared task T_i can be thus regarded as an independent task with an execution time $c'_{i,k}$ and a minimum inter-arrival time p_i, to which the highest priority is given, on every processor P_k. As a result, all tasks are still scheduled strictly in order of fixed-priority.

Now, we move on the case in which one processor executes two shared tasks. Let us assume that another task T_j is shared among three processors T_3, T_4, and T_5, under the assumption that a former task T_i has been already assigned to three processors P_1, P_2, and P_3 but P_3 is not filled to capacity yet, as shown in Figure 3. Here, we need to break a tie between two shared tasks T_i and T_j assigned to the same processor P_3. DM-PM is designed so that ties are broken in favor of the one assigned later to the processor. Thus, in the example, T_j has a higher priority than T_i on P_3.

Figure 4 depicts an example of scheduling two shared tasks T_i and T_j, based on the tie-breaking rule above, that are assigned to processors as shown in Figure 3. Jobs of T_i and T_j are generally executed by the highest priority. However, the second job of T_i is blocked by the second job of T_j, when it is migrated to P_3 from P_2, because T_j has a higher priority. The third job of T_i is also preempted and blocked by the third job of T_j. Here, we see the reason why ties are broken between two shared tasks in favor of the one

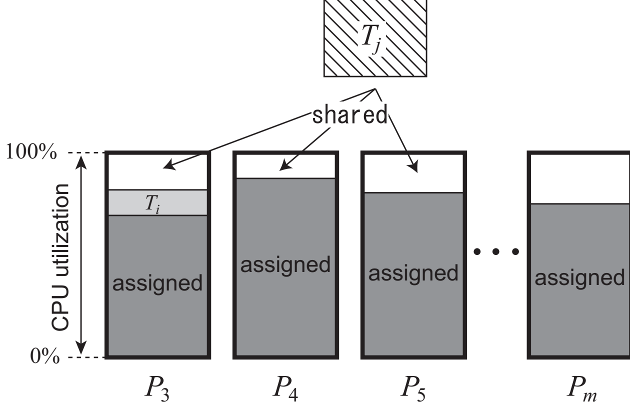


Figure 3. Example of assigning two shared tasks to one processor.

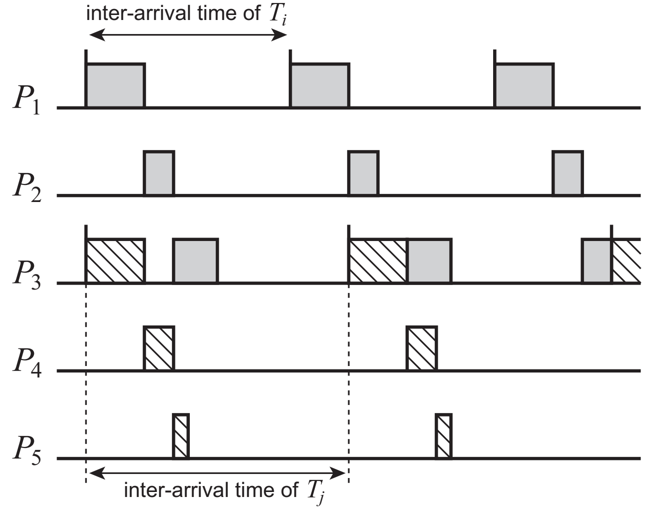


Figure 4. Example of scheduling two shared tasks on one processor.

assigned later to the processor. The execution of T_i is not affected very much, even if it is blocked by T_j, since P_3 is a last processor for T_i to execute. Meanwhile, P_3 is a first processor for T_j to execute, and thus the following execution would be affected very much, if it is blocked on P_3.

Implementation of DM-PM is fairly simplified as compared to the previous algorithms based on semi-partitioned scheduling, because all we have to renew implementation of DM is to set a timer, when a job of a shared task T_i is released on or is migrated to a processor P_k at time t, so that the scheduler will be invoked at time $t + c'_{i,k}$ to preempt the job of T_i for migration. If P_k is a last processor for T_i to execute, we do not have to set a timer. On the other hand, many high-resolution timers are required for implementation of the previous algorithms [7, 4, 15, 16, 17].

4.2 Execution Capacity of Shared Tasks

We now describe how to compute the execution capacity of a shared task on each processor. The amount of execution capacity must guarantee that timing constraints of all tasks are not violated, while processor resource is given to the shared task as much as possible to improve schedulability. To this end, we make use of response time analysis.

It has been known [21] that the response time of tasks is never greater than the case in which all tasks are released at the same time, so-called *critical instant*, in fixed-priority scheduling. As we mentioned before, DM-PM guarantees that all tasks are scheduled strictly in order of priority, the worst-case response time is also obtained at the critical instant. Henceforth, we assume that all the tasks are released at the critical instant t_0.

Consider two tasks T_i and T_j, regardless of whether they are fixed tasks or shared tasks. T_i is assigned a lower priority than T_j. Let $I_{i,j}(d_i)$ be the maximum interference (blocking time) that T_i receives from T_j within a time interval of length d_i. Since we assume that all tasks meet deadlines, a job of T_i is blocked by T_j for at most $I_{i,j}(d_i)$. Given the release at the critical instant t_0, it is clear that the total amount of time consumed by a task within any interval $[t_0, t_1)$ is maximized, when the following two conditions hold.

1. The task is released periodically at its minimum inter-arrival time.
2. Every job of the task consumes exactly c_i time units without being preempted right after its release.

The formula of $I_{i,j}(d_i)$, the maximum interference that T_i receives from T_j within d_i, is derived as follows. According to [11], the maximum interference that a task receives from another task depends on the relation among execution time, period, and deadline. Hereinafter, let $F = \lfloor d_i/p_j \rfloor$ denote the maximum number of jobs of T_j that complete within a time interval of length d_i.

We first consider the case of $d_i \geq F p_j + c_j$, in which the deadline of T_i occurs while T_j is not executed, as shown in Figure 5. In this case, $I_{i,j}(d_i)$ is obtained by Equation (1).

$$I_{i,j}(d_i) = F c_j + c_j = (F + 1)c_j \tag{1}$$

We next consider the case of $d_i \leq F p_j + c_j$, in which the deadline of T_i occurs while T_j is executed, as shown in Figure 6. In this case, $I_{i,j}(d_i)$ is obtained by Equation (2).

$$I_{i,j}(d_i) = d_i - F(p_j - c_j) \tag{2}$$

For the sake of simplicity of description, the notation of $I_{i,j}(d_i)$ unifies Equation (1) and Equation (2) afterwards.

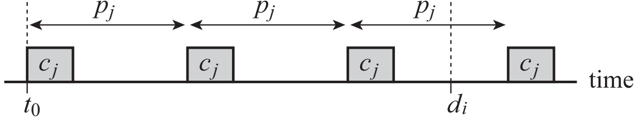


Figure 5. Case 1: $d_i \geq Fp_j + c_j$

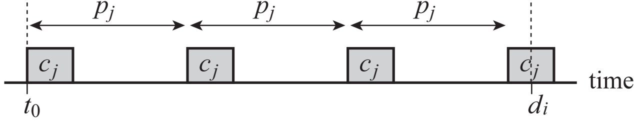


Figure 6. Case 2: $d_i \leq Fp_j + c_j$

The worst-case response time $R_{i,k}$ of each task T_i on P_k is then given by Equation (3), where $\mathcal{P}_k$ is a set of tasks that have been assigned to P_k, and $\mathcal{H}_i$ is a set of tasks that have priorities higher than or equal to T_i.

$$R_{i,k} = \sum_{T_j \in \mathcal{P}_k \cap \mathcal{H}_i} I_{i,j}(d_i) + c_i \tag{3}$$

We then consider the total amount of time that a shared task competes with another task. Let T_s be a shared task, and P_k be a processor to which the share of T_s is assigned. As we mention in Section 4.1, a shared task T_s can be regarded as an independent task with an execution time $c'_{s,k}$ and a minimum inter-arrival time p_s, to which the highest priority is given, on every processor P_k. The maximum total amount $W_{s,k}(d_i)$ of time that T_s competes with a task T_i on P_k within a time interval of length d_i is therefore obtained by Equation (4).

$$W_{s,k}(d_i) = \left\lceil \frac{d_i}{p_s} \right\rceil c'_{s,k} \tag{4}$$

In order to guarantee all tasks to meet deadlines, the following condition must hold for every task T_i on every processor P_k to which a shared task T_s is assigned.

$$R_{i,k} + W_{s,k}(d_i) \leq d_i \tag{5}$$

It is clear that the value of $c'_{s,k}$ is maximized for $R_{i,k} + W_{s,k}(d_i) = d_i$. Finally, $c'_{s,k}$ is given by Equation (6), where $G = \lceil d_i/p_s \rceil$.

$$c'_{s,k} = \min \left\{ \frac{d_i - R_{i,k}}{G} \,\middle|\, T_i \in \mathcal{P}_k \right\} \tag{6}$$

In the end, we describe how to assign tasks to processors. As most partitioning algorithms [13, 25, 18, 14] do, each task is assigned to the first processor upon which a schedulable condition is satisfied. The schedulable condition of T_i for P_k here is defined by $R_{i,k} \leq d_i$. If T_i does not satisfy the schedulable condition, its utilization share is going to be split across processors.

```
 1. for each P_k ∈ Π
 2.   c_req := c_s;
 3.   c'_s,k := 0;
 4.   for each T_i ∈ 𝒫_k
 5.     if T_i is a shared task then
 6.       x := (d_i − c_i)/⌈d_i/p_s⌉;
 7.     else
 8.       x := (d_i − R_i,k)/⌈d_i/p_s⌉;
 9.     end if
10.     if x < c'_s,k then
11.       c'_s,k := max(0, x);
12.     end if
13.   end for
14.   if c'_s,k ≠ 0 then
15.     𝒫_k := 𝒫_k ∪ {T_s}:
16.     c_req := c_req − c'_s,k:
17.     if c_req = 0 then
18.       Π := Π \ {P_k}:
19.       return SUCCESS:
20.     else if c_req < 0 then
21.       c'_s,k := c'_s,k + c_req:
22.       return SUCCESS:
23.     else
24.       Π := Π \ {P_k}:
25.     end if
26.   end if
27. end for
28. return FAILURE:
```

Figure 7. Pseudo code of splitting T_s.

Figure 7 shows the pseudo code of splitting T_s. Π is a set of processors processors that have spare capacity to accept tasks. c_{req} is a temporal variable that indicates the remaining execution requirement of T_s, which must be assigned to some processors. For each processor, the algorithm computes the value of $c'_{s,k}$ until the total of those $c'_{s,k}$ reaches c_s. The value of each $c'_{s,k}$ is based on Equation (6). Notice that if T_i is a shared task that has been assigned to P_k before T_s, the temporal execution capacity is not denoted by $(d_i - c'_{i,k})/\lceil d_i/p_i \rceil$ but by $(d_i - c_i)/\lceil d_i/p_i \rceil$ (line 6), because a job of T_i released at time t always completes at time $t + c_i$ given that T_i is assigned the highest priority. Otherwise, it is denoted by $(d_i - R_{i,k})/\lceil d_i/p_s \rceil$ (line 8). The value of $c'_{s,k}$ must be non-negative (line 11). If $c'_{s,k}$ is successfully obtained, the share of T_s is assigned to P_k (line 15). Now c_{req} is reduced to $c_{req} - c'_{s,k}$ (line 16). A non-positive value of c_{req} means that the utilization share of T_s has been entirely assigned to some processors. Thus, it declares success. Here,

a negative value of c_{req} means that the execution capacity has been excessively assigned to T_s. Therefore, we need to adjust the value of $c'_{s,k}$ for the last portion (line 21). If c_{req} is still positive, the same procedure is repeated.

4.3 Optimization

This section considers optimization of DM-PM. Remember again that a shared task T_s can be regarded as an independent task with an execution time $c'_{s,k}$ and a minimum inter-arrival time p_s, to which the highest priority is given, on every processor P_k. We realize from this characteristic that if T_s has the shortest relative deadline on a processor P_k, the resultant scheduling is conformed to DM, though the execution time of T_s is transformed into $c'_{s,k}$.

Based on the idea above, we consider such an optimization that sorts a task set in non-increasing order of relative deadline before the tasks are assigned to processors. As a result, all tasks that have been assigned to the processors before T_s always have longer relative deadlines than T_s. In other words, T_s always has the shortest relative deadline at this point.

T_s may not have the shortest relative deadline on a processor P_k, if other tasks are later assigned to P_k. Remember that those tasks have shorter relative deadlines than T_s, since a task set is sorted in non-increasing order of relative deadline. According to DM-PM, T_s is assigned to each P_k so that P_k is filled to capacity, except that P_k is a last processor to which T_s is assigned. We therefore need to concern only such a last processor P_l that executes T_s.

In fact, there is no need to forcefully give the highest priority to T_s on P_l, because the next job of T_s will be released at its next release time, regardless of its completion time, whereas it is necessary to give the highest priority to T_s on the preceding processors, because T_s is never executed on the next processor before the execution capacity is consumed. We thus modify DM-PM for optimization so that the prioritization rule is strictly conformed to DM. As a result, a shared task would have a lower priority than fixed tasks, if they are assigned to the processor later.

The worst case problem. Particularly for implicit-deadline systems where relative deadlines are all equal to periods, a set of tasks is successfully scheduled on each processor P_k, if the processor utilization U_k of P_k satisfies the following well-known condition, where n_k is the number of the tasks assigned to P_k, because the scheduling policy of the optimized DM-PM is strictly conformed to DM.

$$U_k \leq n_k(2^{1/n_k} - 1) \tag{7}$$

The worst-case processor utilization is then derived as 69% for $n_k \rightarrow \infty$. Thus to derive the worst-case performance of DM-PM, we consider a case in which an infinite number of tasks, all of which have very long relative deadlines (close to ∞), meaning very small utilization (close to 0), have been already assigned to every processor. Therefore, the available processor utilization is at most 69% for all processors. Let T_s be a shared task with individual utilization ($u_s = c_s/p_s$) greater than 69%, and P_l be a last processor to which the utilization share of T_s is assigned. We then assume that another task T_i is later assigned to P_l. At this point, the worst-case execution capacity that can be assigned to T_i on P_l is $d_s - c_s = d_s(1 - u_s)$, due to $d_i \leq d_s$. Hence, the worst-case utilization share of T_i on P_l is obtained as follows.

$$u_i = \frac{d_s(1 - u_s)}{d_i} \geq (1 - u_s) \tag{8}$$

Now, we concern a case in which T_s has a very large value of u_s (close to 100%). The worst-case utilization share of T_i is then derived as $u_i = 1 - u_s \simeq 0$, regardless of the processor utilization of P_l. In other words, even though the processor resource of P_l is not fully utilized at all, P_l cannot accept any other tasks.

In order to overcome such a worst case problem, we next modify DM-PM for optimization so that the tasks with individual utilization greater than or equal to 50% are preferentially assigned to processors, before a task set is sorted in non-increasing order of relative deadline. Since no tasks have individual utilization greater than 50%, when T_s is shared among processors, the worst-case execution capacity of T_i is improved to $u_i = 1 - u_s \geq 50\%$. As a result, the optimized DM-PM guarantees that the processor utilization of every processor is at least 50%, which means that the entire multiprocessor utilization is also at least 50%. Given that no prior fixed-priority algorithms have utilization bounds greater than 50% [6], our result is sufficient. Remember that this is the worst case. The simulation-based evaluation presented in Section 5 shows that the optimized DM-PM generally performs much better than the worst case.

4.4 Preemptions Bound

The number of preemptions within a time interval of length L is bounded as follows. Let $N(L)$ be the worst-case number of preemptions within L for DM. Since preemptions may occur every time jobs arrive in DM, $N(L)$ is given by Equation (9), where τ is a set of all tasks.

$$N(L) = \sum_{T_i \in \tau} \left\lceil \frac{L}{p_i} \right\rceil \tag{9}$$

Let $N^*(L)$ then be the worst-case number of preemptions within L for DM-PM. It is clear that there are at most $m - 1$ shared tasks. Each shared task is migrated from one processor to another processor once in a period. Every time a

shared task is migrated from one processor to another processor, two preemptions occurs: one occurs on the source processor and the other occurs on the destination processor. Hence, $N^*(L)$ is given by Equation (9), where τ' is a set of tasks that are shared among multiple processors.

$$N^*(L) = N(L) + 2(m-1)\left\lceil \frac{L}{\min\{p_s \mid T_s \in \tau'\}} \right\rceil \qquad (10)$$

5 Evaluation

In this section, we show the results of simulations conducted to evaluate the effectiveness of DM-PM, as compared to the prior algorithms: RMDP [17], FBB-FDD [14], and Partitioned DM (P-DM). RMDP is an algorithm based on semi-partitioned scheduling, though the approach and the scheduling policy are different from DM-PM. FBB-FDD and P-DM are algorithms based on partitioned scheduling. FBB-FDD sorts a task set in non-decreasing order of relative deadline, and assigns tasks to processors based on a first-fit heuristic [13]. P-DM assigns tasks based on first-fit heuristic for simplicity without sorting a task set. The tasks are then scheduled according to DM. Note that FBB-FDD uses a polynomial-time acceptance test in a partitioning phase, while P-DM uses a response time analysis presented in Section 4.2.

To the best of our knowledge, FBB-FDD is the best performer among the fixed-priority algorithms based on partitioned scheduling. We are then not aware of any fixed-priority algorithms, except for RMDP, that are based on semi-partitioned scheduling. We thus consider that those algorithms are worthwhile to compare with DM-PM.

5.1 Simulation Setup

A series of simulations has a set of parameters: u_{sys}, m, u_{min}, and u_{max}. u_{sys} denotes system utilization. m is the number of processors. u_{min} and u_{max} are the minimum utilization and the maximum utilization of every individual task respectively.

For every set of parameters, we generate 1,000,000 task sets. A task set is said to be successfully scheduled, if all tasks in the task set are successfully assigned to processors. The effectiveness of an algorithm is then estimated by *success ratio*, which is defined as follows.

$$\frac{\textit{the number of successfully-scheduled task sets}}{\textit{the number of submitted task sets}}$$

The system utilization u_{sys} is set every 5% within the range of $[0.5, 1.0]$. Due to limitation of space, we have three sets of m such that $m = 4$, $m = 8$, and $m = 16$. Each task set $\mathcal{T}$ is then generated so that the total utilization $\sum_{T_i \in \mathcal{T}} u$ becomes equal to $u_{sys} \times m$. The utilization of every individual task is uniformly distributed within the range of $[u_{min}, u_{max}]$. Due to limitation of space, we have five sets of $[u_{min}, u_{max}]$ such that $[0.1, 1.0]$, $[0.25, 0.75]$, $[0.4, 0.6]$, $[0.5, 1.0]$, and $[0.1, 0.5]$. The first three setups generate task sets in which the average utilization is about 0.5, but the minimum and the maximum are different. The last two setups, meanwhile, generate such task sets that contain heavy tasks and light tasks respectively. The minimum inter-arrival time of each task is also uniformly distributed within the range of $[100, 10,000]$. For every task T_i, once u_i and p_i are determined, we compute the execution time of T_i by $c_i = u_i \times p_i$.

Since RMDP is designed for implicit-deadline systems, for fairness we presume that all tasks have relative deadlines equal to periods. However, DM-PM is also effective to explicit-deadline systems where relative deadlines are different from periods.

5.2 Simulation Results

Figure 8 shows the results of simulations with $[u_{min}, u_{max}] = [0.1, 1.0]$. Here, DM-PM(opt) represents the optimized DM-PM. DM-PM substantially outperforms the prior algorithms. Particularly, the optimized DM-PM is able to schedule all task sets successfully, even though system utilization is around 0.9, while the prior algorithms sometimes return failure when it exceeds 0.6 to 0.7. It has been reported in [19] that the average case of achievable processor utilization for DM, as well as RM, is about 88% on uniprocessors. Thus, DM-PM reflects the schedulability of DM on multiprocessors. Even though a task set is not sorted, DM-PM is able to schedule all task sets when system utilization is smaller than 0.7 to 0.8.

The performance of DM-PM is better as the number of processor is greater, because tasks are more likely to be successfully shared among processors, if there are more processors, when they cannot be assigned to any individual processors. Since RMDP is also able to share tasks among processors, it outperforms FBB-FDD and P-DM that are based on classical partitioned scheduling. However, RMDP is still far inferior to DM-PM. Thus, we recognize the effectiveness of the approach considered in DM-PM. Note that P-DM outperforms FBB-FDD, because P-DM uses an acceptance test based on the presented response time analysis, while FBB-FDD does a polynomial-time test.

Figure 9 shows the results of simulations with $[u_{min}, u_{max}] = [0.25, 0.75]$. The relative order of performance in the simulated algorithms is mostly equal to the previous case of $[u_{min}, u_{max}] = [0.1, 1.0]$. However, the success ratio is entirely degraded, particularly for P-DM and FBB-FDD: they may return failure despite system utilization less than 0.6. Since the utilization of every individual task is never smaller than 0.25, partitioning is more likely to fail. For instance, even though two processors have spare

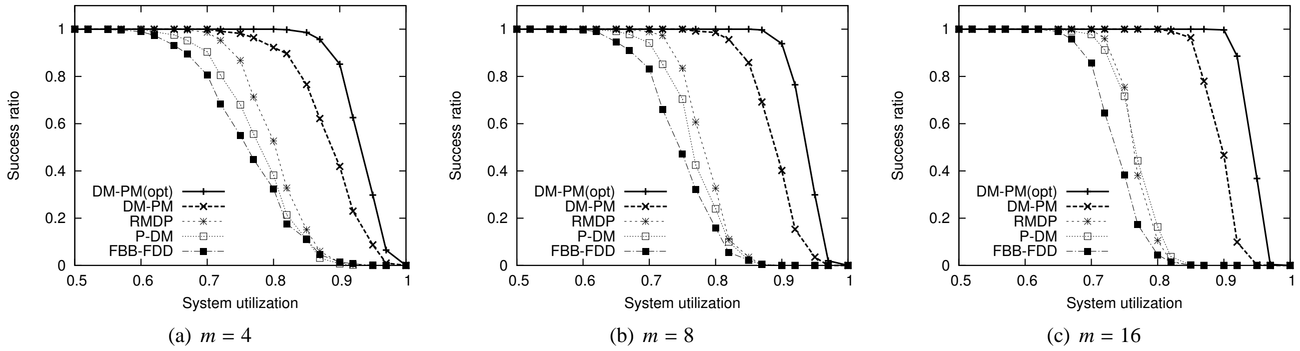


Figure 8. Results of simulations ($[u_{min}, u_{max}] = [0.1, 1.0]$).

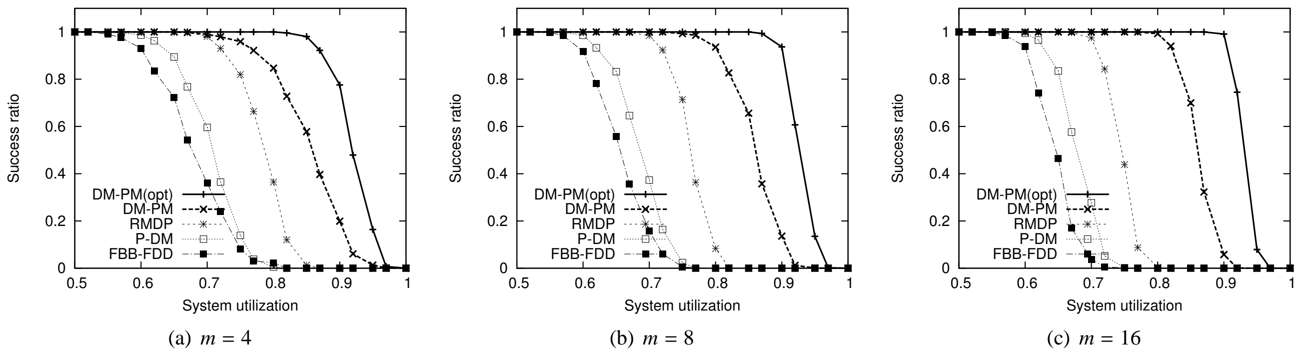


Figure 9. Results of simulations ($[u_{min}, u_{max}] = [0.25, 0.75]$).

capacity of utilization 0.2 respectively, i.e. the total spare capacity is utilization 0.4, a task with utilization 0.25 can be assigned to neither of them. However, in the previous case of $u_{min}, u_{max} = [0.1, 1.0]$, the utilization of every individual task is at least 0.1. Thus, the spare capacity of processors can be used more efficiently. Meanwhile, the success ratio for DM-PM and RMDP is not degraded as much as that for FBB-FDD and DM, because those algorithms are based on semi-partitioned scheduling.

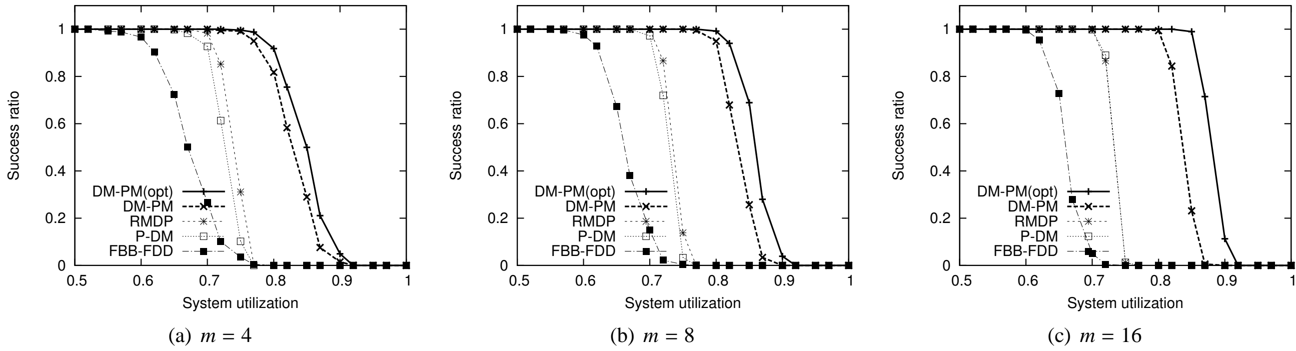
Figure 10 shows the results of simulations with $[u_{min}, u_{max}] = [0.4, 0.6]$. Since the utilization of every individual task is about 0.5, two tasks are likely to fill one processor to capacity. If every processor is mostly filled to capacity, there is not much advantage to share tasks. As a result, the performance of RMDP is not much better than DM. Since DM-PM splits a shared task across processors more efficiently than RMDP, it outperforms RMDP, but the amount of schedulability improvement is not as much as the previous two cases. From this result, we recognize that DM-PM prefers the case in which the utilization of every individual task is widely distributed.

Figure 11 shows the results of simulations with $[u_{min}, u_{max}] = [0.5, 1.0]$. The tasks are all heavy with utilization greater than 0.5. Note that no individual processors execute more than one task, because processor utilization cannot be greater than 1.0. As a result, the preciseness of approximations for a polynomial-time test of FBB-FDD is improved, and thus it offers competitive performance to P-DM that uses the presented response time analysis. As for the performance of DM-PM and RMDP, it is very similar to the first case of $[u_{min}, u_{max}] = [0.1, 1.0]$.

Figure 12 shows the results of simulations with $[u_{min}, u_{max}] = [0.1, 0.5]$. In contrast to the previous case of $[u_{min}, u_{max}] = [0.5, 1.0]$, the tasks are all light with utilization smaller than 0.5. As a result, the approximations for a polynomial-time test of FBB-FDD are likely imprecise, and thus it is far inferior to P-DM. It is also true that the number of tasks is greater than the previous cases, because there are only light tasks. It has been known by the formula [21] that the schedulability of fixed-priority algorithms is generally decreased as the number of tasks is increased. Hence, the success ratio of each algorithm is entirely declined.

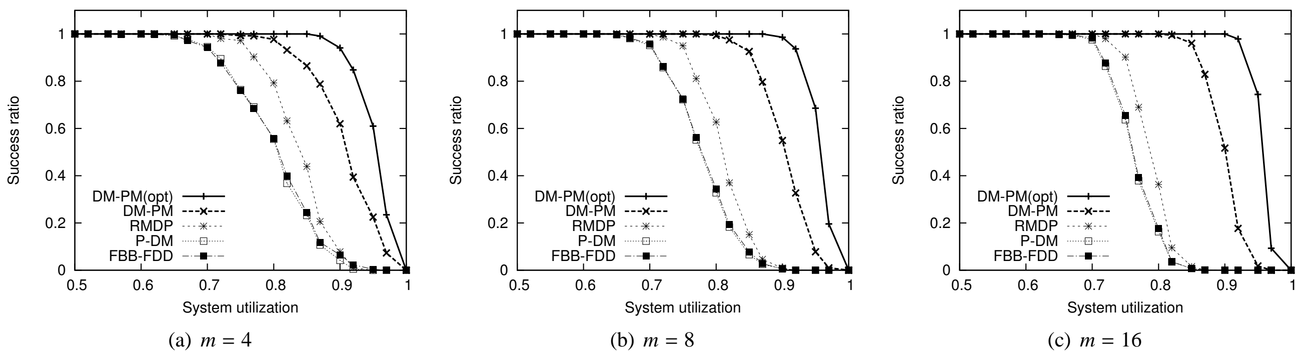
6 Conclusion

A new algorithm was presented for semi-partitioned fixed-priority scheduling of sporadic task systems on identical multiprocessors. We designed the algorithm so that



(a) $m = 4$ (b) $m = 8$ (c) $m = 16$

Figure 10. Results of simulations ($[u_{min}, u_{max}] = [0.4, 0.6]$).



(a) $m = 4$ (b) $m = 8$ (c) $m = 16$

Figure 11. Results of simulations ($[u_{min}, u_{max}] = [0.5, 1.0]$).

a task is qualified to migrate across processors, only if it cannot be assigned to any individual processors, in such a manner that it is never migrated back to the same processor within the same period, once it is migrated from one processor to another processor. The scheduling policy was then simplified to reduce the number of preemptions and migrations as much as possible for practical use.

We also optimized the algorithm to improve schedulability. Any implicit-deadline systems are successfully scheduled by the optimized algorithm, if system utilization does not exceed 50%. We are not aware of any fixed-priority algorithms that have utilization bounds greater than 50%. Thus, our result seems sufficient.

The simulation results showed that the new algorithm significantly outperforms the traditional fixed-priority algorithms regardless of the number of processors and the utilization of tasks. Especially for the case in which the utilization of every individual task is widely distributed, the new algorithm was able to schedule all task sets successfully, even though system utilization is close to 90%. The parameters used in simulations are limited, but we can estimate that the new algorithm is also effective to different environments.

In the future work, we will consider arbitrary-deadline systems where relative deadlines may be longer than periods, while we consider constrained-deadline systems where relative deadlines are shorter than or equal to periods. We are also interested in applying the presented semi-partitioned scheduling approach to dynamic-priority scheduling. The implementation problems of the algorithm in practical operating systems are left open.

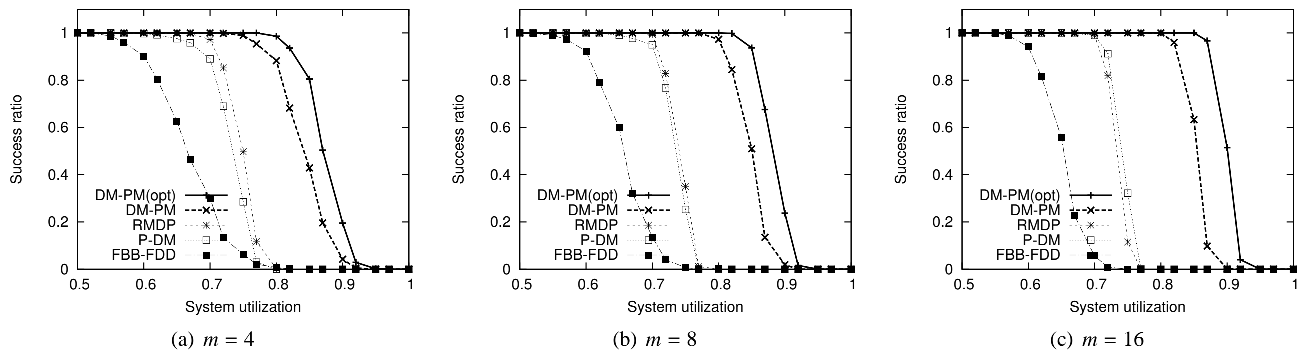
References

[1] J. Anderson, V. Bud, and U.C. Devi. An EDF-based Scheduling Algorithm for Multiprocessor Soft Real-Time Systems. In *Proceedings of the Euromicro Conference on Real-Time Systems*, pages 199–208, 2005.

[2] B. Andersson. Global Static-Priority Preemptive Multiprocessor Scheduling with Utilization Bound 38%. In *Proceedings of the International Conference on Principles of Distributed Systems*, pages 73–88, 2008.

[3] B. Andersson, S. Baruah, and J. Jonsson. Static-priority Scheduling on Multiprocessors. In *Proceedings of the IEEE Real-Time Systems Symposium*, pages 193–202, 2001.

[4] B. Andersson and K. Bletsas. Sporadic Multiprocessor Scheduling with Few Preemptions. In *Proceedings of the Eu-*



(a) $m = 4$ (b) $m = 8$ (c) $m = 16$

Figure 12. Results of simulations ($[u_{min}, u_{max}] = [0.1, 0.5]$).

romicro Conference on Real-Time Systems, pages 243–252, 2008.

[5] B. Andersson, K. Bletsas, and S. Baruah. Scheduling Arbitrary-Deadline Sporadic Task Systems Multiprocessors. In *Proceedings of the IEEE Real-Time Systems Symposium*, pages 385–394, 2008.

[6] B. Andersson and J. Jonsson. The Utilization Bounds of Partitioned and Pfair Static-Priority Scheduling on Multiprocessors are 50%. In *Proceedings of the Euromicro Conference on Real-Time Systems*, pages 33–40, 2003.

[7] B. Andersson and E. Tovar. Multiprocessor Scheduling with Few Preemptions. In *Proceedings of the IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, pages 322–334, 2006.

[8] T.P. Baker. An Analysis of Fixed-Priority Schedulability on a Multiprocessor. *Real-Time Systems*, 32:49–71, 2006.

[9] S. Baruah, N. Cohen, C.G. Plaxton, and D. Varvel. Proportionate Progress: A Notion of Fairness in Resource Allocation. *Algorithmica*, 15:600–625, 1996.

[10] S. Baruah, J. Gehrke, and C.G. Plaxton. Fast Scheduling of Periodic Tasks on Multiple Resources. In *Proceedings of the International Parallel Processing Symposium*, pages 280–288, 1995.

[11] G.C. Buttazzo. *HARD REAL-TIME COMPUTING SYSTEMS: Predictable Scheduling Algorithms and Applications*. Kluwer Academic Publishers, 1997.

[12] H. Cho, B. Ravindran, and E.D. Jensen. An Optimal Real-Time Scheduling Algorithm for Multiprocessors. In *Proceedings of the IEEE Real-Time Systems Symposium*, pages 101–110, 2006.

[13] S. K. Dhall and C. L. Liu. On a Real-Time Scheduling Problem. *Operations Research*, 26:127–140, 1978.

[14] N. Fisher, S. Baruah, and T. Baker. The Partitioned Multiprocessor Scheduling of Sporadic Task Systems according to Static Priorities. In *Proceedings of the Euromicro Conference on Real-Time Systems*, pages 118–127, 2006.

[15] S. Kato and N. Yamasaki. Real-Time Scheduling with Task Splitting on Multiprocessors. In *Proceedings of the IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, pages 441–450, 2007.

[16] S. Kato and N. Yamasaki. Portioned EDF-based Scheduling on Multiprocessors. In *Proceedings of the ACM International Conference on Embedded Software*, 2008.

[17] S. Kato and N. Yamasaki. Portioned Static-Priority Scheduling on Multiprocessors. In *Proceedings of the IEEE International Parallel and Distributed Processing Symposium*, 2008.

[18] S. Lauzac, R. Melhem, and D. Mosses. An Efficient RMS Admission Control and Its Application to Multiprocessor Scheduling. In *Proceedings of the IEEE International Parallel Processing Symposium*, pages 511–518, 1998.

[19] J.P. Lehoczky, L. Sha, and Y. Ding. The Rate Monotonic Scheduling Algorithm: Exact Charaterization and Average Case Behavior. In *Proceedings of the IEEE Real-Time Systems Symposium*, pages 166–171, 1989.

[20] J. Leung and J. Whitehead. On the Complexity of Fixed-Priority Scheduling of Periodic Real-Time Tasks. *Performance Evaluation, Elsevier Science*, 22:237–250, 1982.

[21] C. L. Liu and J. W. Layland. Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment. *Journal of the ACM*, 20:46–61, 1973.

[22] J.M. Lopez, J.L. Diaz, and D.F. Garcia. Minimum and Maximum Utilization Bounds for Multiprocessor Rate-Monotonic Scheduling. *IEEE Transactions on Parallel and Distributed Systems*, 28:39–68, 2004.

[23] J.M. Lopez, M. Garcia, J.L. Diaz, and D.F. Garcia. Utlization Bounds for Multiprocessor Rate-Monotonic Scheduling. *Real-Time Systems*, 24:5–28, 2003.

[24] D. Oh and T. Baker. Utlization Bounds for N-Processor Rate Monotonic Scheduling with Static Processor Assignment. *Real-Time Systems*, 15:183–192, 1998.

[25] Y. Oh and S. Son. Allocating Fixed-Priority Periodic Tasks on Multiprocessor Systems. *Real-Time Systems*, 9:207–239, 1995.

[26] S. Ramamurthy and M. Moir. Static-Priority Periodic Scheduling on Multiprocessors. In *Proceedings of the IEEE Real-Time Systems Symposium*, pages 69–78, 2000.