A Fixed-Priority Scheduling Algorithm for Multiprocessor Real-Time Systems

Shinpei Kato
The University of Tokyo
Japan

1. Introduction

Major chip manufacturers have adopted multicore technologies in recent years, due to the thermal problems that distress traditional single-core chip designs in terms of processor performance and power consumption. Nowadays, multiprocessor platforms have proliferated in the marketplace, not only for servers and personal computers but also for embedded machines. The research on real-time systems has been therefore renewed for those multiprocessor platforms, especially in the context of real-time scheduling.

Real-time scheduling techniques for multiprocessors are mainly classified into *partitioned scheduling* and *global scheduling*. In the partitioned scheduling class, tasks are first assigned to specific processors, and then executed on those processors without migrations. In the global scheduling class, on the other hand, all tasks are stored in a global queue, and the same number of the highest priority tasks as processors are selected for execution.

The partitioned scheduling class has such an advantage that can reduce a problem of multiprocessor scheduling into a set of uniprocessor one, after tasks are partitioned. In addition, it does not incur runtime overhead as much as global scheduling, since tasks never migrate across processors. However, there is a disadvantage in theoretical scheduling performance, i.e., schedulability a likelihood of a system being schedulable. Specifically, the worst-case leads to that a periodic task system can cause deadline misses in partitioned scheduling, if the system utilization exceeds 50% (Lopez et al., 2004).

The global scheduling class is meanwhile attractive in the worst-case schedulability. In this class, Pfair (Baruah et al., 1996) and LLREF (Cho et al., 2006) are known to be optimal algorithms. Any task sets are scheduled successfully by those algorithms, if the processor utilization does not exceed 100%. However, the number of migrations and context switches is often criticized. This scheduling class also provides concise and efficient algorithms, such as EDZL (Cho et al., 2002) and EDCL (Kato & Yamasaki, 2008a), which perform with less preemptions than the optimal ones, but the absolute worst-case processor utilization is still 50%.

For the purpose of finding a balance point between partitioned scheduling and global scheduling, recent work have made available a new class, called *semi-partitioned scheduling* in this paper. In this scheduling class, most tasks are fixed to specific processors as partitioned scheduling to reduce the number of migrations, while a few tasks may migrate across processors to improve available processor utilization as much as possible.

In addition to scheduling classes, the real-time systems community often argue priority-driven scheduling policies. Commodity operating systems for practical use usually pre-

fer fixed-priority algorithms in terms of implementation simplicity and priority-based predictability. The most well-known fixed-priority algorithm is Rate Monotonic (RM) (Liu & Layland, 1973). Andersson et al. showed that RM based on global scheduling offers the bound on system utilization no greater than 33% (Andersson et al., 2001), while RM based on partitioned scheduling offers the one up to 50% (Andersson & Jonsson, 2003). So if we restrict our attention to fixed-priority algorithms, partitioned scheduling may be more efficient than global scheduling.

This chapter presents a new fixed-priority algorithm based on semi-partitioned scheduling. The presented algorithm has two major contributions. First, it allows tasks to migrate across processors only if they cannot be assigned (fixed) to any individual processors, to strictly dominate the previous algorithms based on classical partitioned scheduling. Second, its scheduling policy conforms Deadline Monotonic (DM) (Leung & Whitehead, 1982), which is a generalization of RM for arbitrary-deadline tasks, to make available the prior analytical results of DM (and RM). The contents of this chapter are based on the paper in (Kato & Yamasaki, 2009). The reminder of this chapter 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 chapter is concluded in Section 6.

2. Related Work

The concept of semi-partitioned scheduling was originally introduced by EDF-fm (Anderson et al., 2005). 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 (Andersson & Tovar, 2006) is designed to guarantee all tasks to meet deadlines for implicit-deadline periodic task systems. Here, a deadline is said to be implicit, if it is equal to a period. EKG differs from EDF-fm in that 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. The optimal parameter configuration leads to that any task sets are scheduled successfully with more preemptions and migrations.

In the later work (Andersson & Bletsas, 2008), EKG is extended for sporadic task systems. Here, a task is said to be sporadic, if its job arrivals are separated at least length equal to its period. The extended algorithm is also parametric with respect to the length of the time slots reserved for migratory tasks. EDF-SS (Andersson et al., 2008) is a further extension of the algorithm for arbitrary-deadline systems. Here, a deadline is said to be arbitrary, if it is not necessarily equal to a period. It is shown by simulations that EDF-SS offers a significant improvement on schedulability over EDF-FFD (Baker, 2005), the best performer among partitioned scheduling algorithms.

EDDHP (Kato & Yamasaki, 2007) is designed in consideration of reducing preemptions, as compared to EKG. In EDDHP, 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 unlike EDF-fm. It is shown by simulations that EDDHP outperforms partitioned EDF-based algorithms, with less preemptions than EKG. EDDP (Kato & Yamasaki, 2008b) 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 (Kato & Yamasaki, 2008c) is a fixed-priority version of EDDHP: the highest priority is given to migratory tasks, and other fixed tasks have the RM priorities. It is shown by simulations that RMDP improves schedulability over 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.

We have several concerns for the previous algorithms mentioned above. First, tasks migrate across processors, even though they can be assigned to individual processors. Hence, we are not sure that those algorithms are truly more effective than classical partitioned scheduling approaches. Then, such tasks may migrate in and out of the same processor many times within the same period, which is likely to cause the cache hit ratio to decline. The number of context switches is also problematic due to repetition of migrations. In addition, optional techniques for EDF and RM, such as synchronization and dynamic voltage scaling, may not be easily available, since the scheduling policy is more or less modified from EDF and RM. In this chapter, we aim at addressing those concerns.

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) 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) 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 classical partitioning approaches Andersson & Jonsson (2003); Dhall & Liu (1978); Fisher et al. (2006); Lauzac et al. (1998); Oh & Son (1995), DM-PM assigns each task to a particular processor, using 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 is going to share the task among more than one processor, whereas a task set is decided to be unfeasible in the classical partitioning approaches. In a scheduling phase, such a shared task is qualified to migrate across those multiple processors.
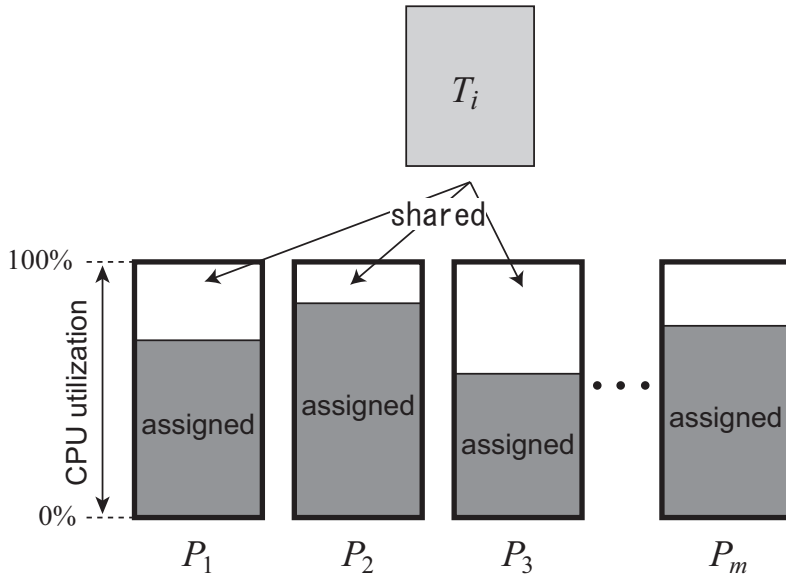

Fig. 1. Example of sharing a task.

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 a 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.

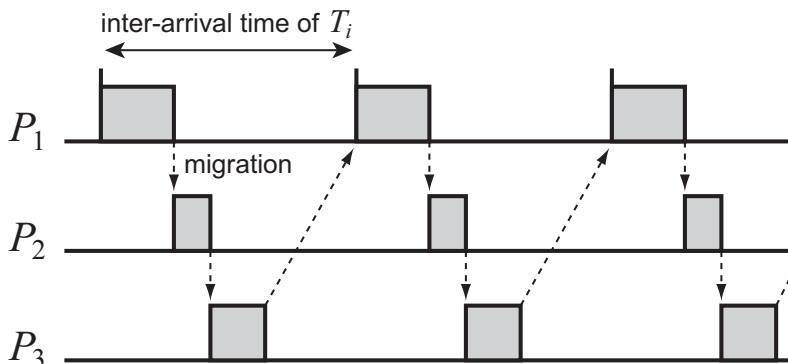- Fixed tasks are then scheduled according to DM.

Fig. 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. Every job of T_i is released on P_1 that has the lowest index. Since T_i is scheduled by the highest priority, it is immediately executed until it consumes $c'_{i,1}$ time units. When $c'_{i,1}$ is consumed, T_i is migrated to the next processor P_2, and then scheduled by the highest priority again. T_i is finally migrated to the last processor P_3 when $c'_{i,2}$ is consumed on P_2, and then executed in the same manner.

The scheduling policy of DM-PM above implies that the execution of a shared task T_i is repeated exactly at its inter-arrival time on every processor, because it is scheduled by the highest priority on each processor until the constant execution capacity is consumed. 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 scheduled strictly in order of fixed-priority, though the scheduling policy is slightly modified from DM.

We next need to consider 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, following that a former task T_i has been assigned to three processors P_1, P_2, and P_3, i.e. P_3 is not filled to capacity yet as explained in the previous example with Figure 3. We here need to break a tie between two shared tasks T_i and T_j assigned to the same processor P_3, since they both have the highest priority. DM-PM is for this 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 in a scheduling phase.

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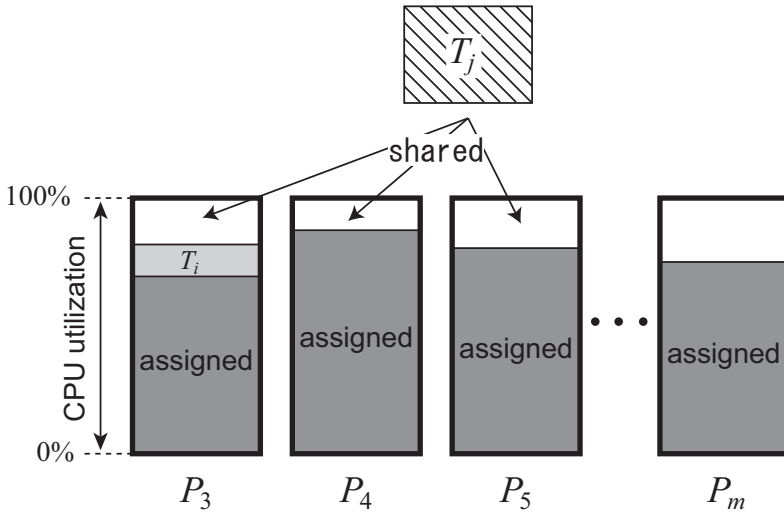
Fig. 3. Example of assigning two shared tasks to one processor.

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 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 Andersson & Bletsas (2008); Andersson & Tovar (2006); Kato & Yamasaki (2007; 2008b;c).

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 Liu & Layland (1973) 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
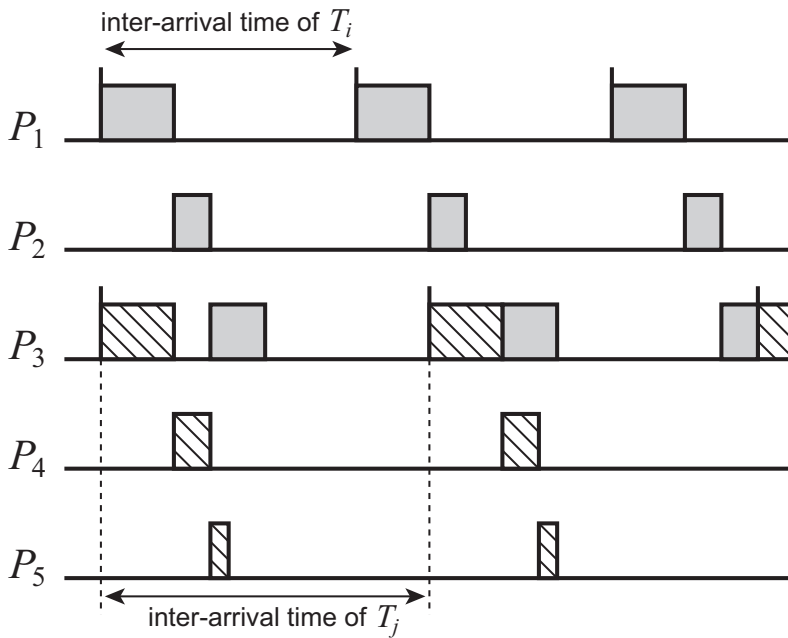
Fig. 4. Example of scheduling two shared tasks on one processor.

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 Buttazzo (1997), 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).
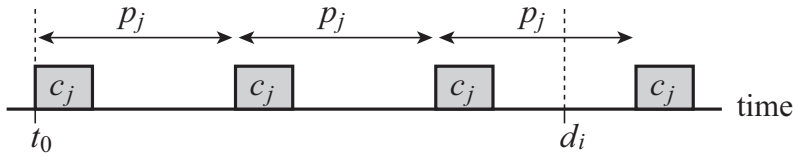
$$I_{i,j}(d_i) = d_i - F(p_j - c_j) \tag{2}$$

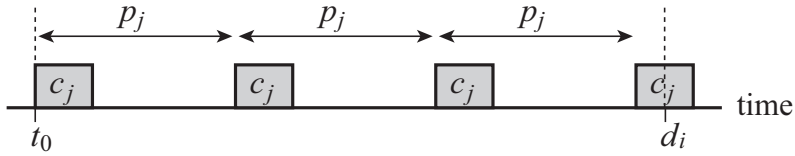Fig. 5. Case 1: $d_i \geq Fp_j + c_j$.

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

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

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

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).

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

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.

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

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 $G = \lceil d_i/p_s \rceil$.

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

In the end, we describe how to assign tasks to processors. As most partitioning algorithms Dhall & Liu (1978); Fisher et al. (2006); Lauzac et al. (1998); Oh & Son (1995) do, each task is

```
 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:
```

Fig. 7. Pseudo code of splitting T_s.

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.

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 proces-

sors. 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 optimally 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. This leads to that 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 processors so that they are filled to capacity, except for a last processor to which T_s is assigned. Thereby for optimization, we 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 the beginning of the next period, 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 unless 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 equal to periods, a set of tasks is scheduled on each processor P_k successfully, 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 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. Note that 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 bound 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 bound 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% Andersson & Jonsson (2003), our outcome seems 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.

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

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 Kato & Yamasaki (2008c), FBB-FDD Fisher et al. (2006), 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 Dhall & Liu (1978). 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 algo-

rithms, except for RMDP, that are based on semi-partitioned scheduling. We thus consider that those algorithms are worthwhile to compare with DM-PM.

The fixed-priority algorithms based on global scheduling, such as Andersson (2008); Andersson et al. (2001); Baker (2006), are not included in a series of simulations, because the previous report Kato & Yamasaki (2008c) on simulation-based evaluation of fixed-priority algorithms testified that their schedulability is in general worse than the ones based on partitioned scheduling.

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.

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

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 simulated only the case for $[u_{min}, u_{max}] = [0.1, 1.0]$. 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.
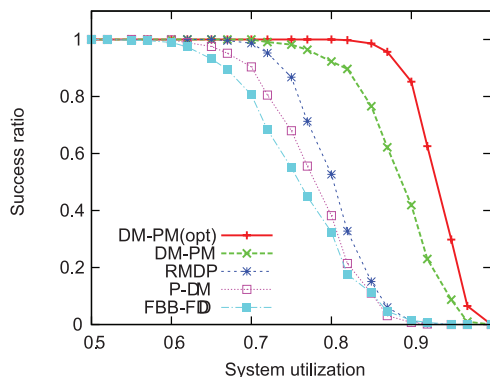
5.2 Simulation Results

Fig. 8. Results of simulations ($m = 4$ and $[u_{min}, u_{max}] = [0.1, 1.0]$).
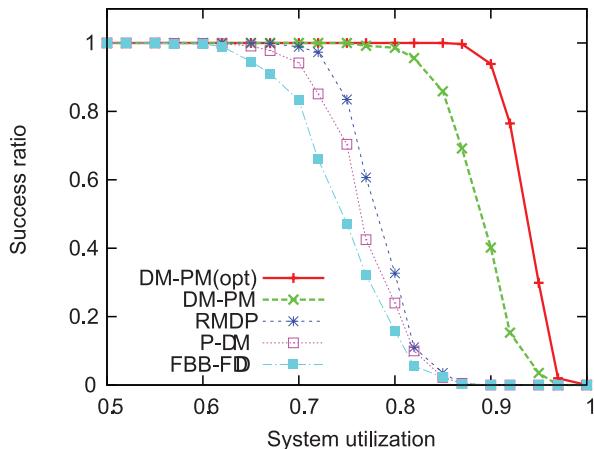
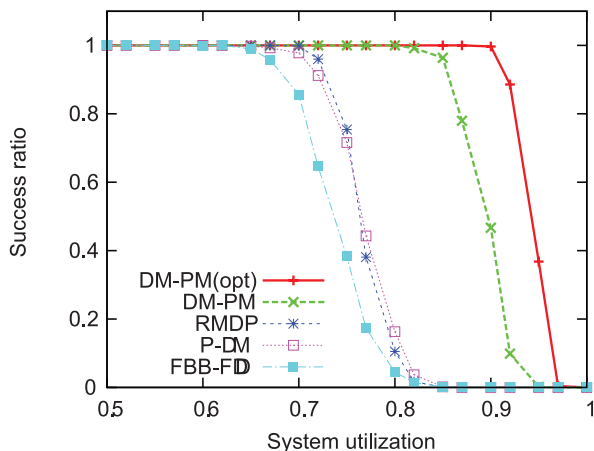Fig. 9. Results of simulations ($m = 8$ and $[u_{min}, u_{max}] = [0.1, 1.0]$).

Fig. 10. Results of simulations ($m = 16$ and $[u_{min}, u_{max}] = [0.1, 1.0]$).

Figure 8, 9, and 10 show the results of simulations with $[u_{min}, u_{max}] = [0.1, 1.0]$ on 4, 8, and 16 processors respectively. 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 more or less return failure at system utilization around 0.6 to 0.7. It has been reported Lehoczky et al. (1989) that the average case of achievable processor utilization for DM, as well as RM, is about 88% on uniprocessors. Hence, the optimized DM-PM reflects the schedulability of DM on multiprocessors. Even without optimization, DM-PM is able to schedule all task sets when system utilization is smaller than 0.7 to 0.8.

On the whole, the performance of DM-PM is better as the number of processors is greater. That is because tasks are shared among processors more successfully, if there are more processors, when they cannot be assigned to any individual processors. Although RMDP is also able to share tasks among processors, it is far inferior to DM-PM, while it outperforms FBB-FDD and P-DM that are based on classical partitioned scheduling. The difference between DM-PM and RMDP clearly demonstrates the effectiveness of the approach considered in this paper. 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.

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 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 outcome 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. The parameters used in simulations are limited, but we can easily 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.

7. References

Anderson, J., Bud, V. & Devi, U. (2005). An EDF-based Scheduling Algorithm for Multiprocessor Soft Real-Time Systems, *Proceedings of the Euromicro Conference on Real-Time Systems*, pp. 199–208.

Andersson, B. (2008). Global Static-Priority Preemptive Multiprocessor Scheduling with Utilization Bound 38%, *Proceedings of the International Conference on Principles of Distributed Systems*, pp. 73–88.

Andersson, B., Baruah, S. & Jonsson, J. (2001). Static-priority Scheduling on Multiprocessors, *Proceedings of the IEEE Real-Time Systems Symposium*, pp. 193–202.

Andersson, B. & Bletsas, K. (2008). Sporadic Multiprocessor Scheduling with Few Preemptions, *Proceedings of the Euromicro Conference on Real-Time Systems*, pp. 243–252.

Andersson, B., Bletsas, K. & Baruah, S. (2008). Scheduling Arbitrary-Deadline Sporadic Task Systems Multiprocessors, *Proceedings of the IEEE Real-Time Systems Symposium*, pp. 385–394.

Andersson, B. & Jonsson, J. (2003). The Utilization Bounds of Partitioned and Pfair Static-Priority Scheduling on Multiprocessors are 50%, *Proceedings of the Euromicro Conference on Real-Time Systems*, pp. 33–40.

Andersson, B. & Tovar, E. (2006). Multiprocessor Scheduling with Few Preemptions, *Proceedings of the IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, pp. 322–334.

Baker, T. (2005). An Analysis of EDF Schedulability on a Multiprocessor, *IEEE Transactions on Parallel and Distributed Systems* **16**: 760–768.

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

Baruah, S., Cohen, N., Plaxton, C. & Varvel, D. (1996). Proportionate Progress: A Notion of Fairness in Resource Allocation, *Algorithmica* **15**: 600–625.

Buttazzo, G. (1997). *HARD REAL-TIME COMPUTING SYSTEMS: Predictable Scheduling Algorithms and Applications*, Kluwer Academic Publishers.

Cho, H., Ravindran, B. & Jensen, E. (2006). An Optimal Real-Time Scheduling Algorithm for Multiprocessors, *Proceedings of the IEEE Real-Time Systems Symposium*, pp. 101–110.

Cho, S., Lee, S., Han, A. & Lin, K. (2002). Efficient Real-Time Scheduling Algorithms for Multiprocessor Systems, *IEICE Transactions on Communications* **E85-B**(12): 2859–2867.

Dhall, S. K. & Liu, C. L. (1978). On a Real-Time Scheduling Problem, *Operations Research* **26**: 127–140.

Fisher, N., Baruah, S. & Baker, T. (2006). The Partitioned Multiprocessor Scheduling of Sporadic Task Systems according to Static Priorities, *Proceedings of the Euromicro Conference on Real-Time Systems*, pp. 118–127.

Kato, S. & Yamasaki, N. (2007). Real-Time Scheduling with Task Splitting on Multiprocessors, *Proceedings of the IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, pp. 441–450.

Kato, S. & Yamasaki, N. (2008a). Global EDF-based Scheduling with Efficient Priority Promotion, *Proceedings of the IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, pp. 197–206.

Kato, S. & Yamasaki, N. (2008b). Portioned EDF-based Scheduling on Multiprocessors, *Proceedings of the ACM International Conference on Embedded Software*.

Kato, S. & Yamasaki, N. (2008c). Portioned Static-Priority Scheduling on Multiprocessors, *Proceedings of the IEEE International Parallel and Distributed Processing Symposium*.

Kato, S. & Yamasaki, N. (2009). Semi-Partitioned Fixed-Priority Scheduling on Multiprocessors, *Proceedings of the IEEE Real-Time and Embedded Technology and Applications Symposium*, pp. 23–32.

Lauzac, S., Melhem, R. & Mosses, D. (1998). An Efficient RMS Admission Control and Its Application to Multiprocessor Scheduling, *Proceedings of the IEEE International Parallel Processing Symposium*, pp. 511–518.

Lehoczky, J., Sha, L. & Ding, Y. (1989). The Rate Monotonic Scheduling Algorithm: Exact Charaterization and Average Case Behavior, *Proceedings of the IEEE Real-Time Systems Symposium*, pp. 166–171.

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

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

Lopez, J., Diaz, J. & Garcia, D. (2004). Utilization Bounds for EDF Scheduling on Real-Time
       Multiprocessor Systems, *Real-Time Systems* **28**: 39–68.
Oh, Y. & Son, S. (1995). Allocating Fixed-Priority Periodic Tasks on Multiprocessor Systems,
       Real-Time Systems **9**: 207–239.

Parallel and Distributed Computing

Edited by Alberto Ros

The 14 chapters presented in this book cover a wide variety of representative works ranging from hardware design to application development. Particularly, the topics that are addressed are programmable and reconfigurable devices and systems, dependability of GPUs (General Purpose Units), network topologies, cache coherence protocols, resource allocation, scheduling algorithms, peertopeer networks, largescale network simulation, and parallel routines and algorithms. In this way, the articles included in this book constitute an excellent reference for engineers and researchers who have particular interests in each of these topics in parallel and distributed computing.

How to reference

In order to correctly reference this scholarly work, feel free to copy and paste the following:

INTECH
open science | open minds