Portioned Static-Priority Scheduling on Multiprocessors *

Shinpei Kato and Nobuyuki Yamasaki
School of Science for Open and Environmental Systems
Keio University, Japan
{shinpei,yamasaki}@ny.ics.keio.ac.jp

Abstract

This paper proposes an efficient real-time scheduling algorithm for multiprocessor platforms. The algorithm is a derivative of the Rate Monotonic (RM) algorithm, with its basis on the portioned scheduling technique. The theoretical design of the algorithm is well implementable for practical use. The schedulability of the algorithm is also analyzed to guarantee the worst-case performance. The simulation results show that the algorithm achieves higher system utilizations, in which all tasks meet deadlines, with a small number of preemptions compared to traditional algorithms.

1 Introduction

Embedded real-time applications in recent years have relied on a power of multiprocessor platforms. With the trend towards chip multiprocessing [20], processor clock frequency can be turned down to contain power consumption and heat generation for embedded computing. The concern herein is the guarantee of timing constraints for recurrent real-time computing. The well-known Rate Monotonic (RM) and Earliest Deadline First (EDF) algorithms [15], which are optimal in uniprocessor scheduling, are no longer optimal in multiprocessor scheduling [11]. They rather perform far poorly depending on properties of task sets. Therefore, alternative scheduling techniques specific for multiprocessor platforms have been discussed recently.

At present, the Pfair algorithms [1, 7, 8], the EKG algorithm [4], and the LLREF algorithm [9] are known optimal for multiprocessors. The worst-case bounds on the achievable system utilizations, with the guarantee of real-time constraints, offered by those algorithms are 100%. However, their computation complexities and scheduling overheads are controversial. Thus, more reasonable scheduling algorithms, which are not able to achieve utilization bounds of 100% but perform with less computation complexities and smaller numbers of task preemptions, are often preferable for practical use. For instance, the EDF-US[1/2] algorithm [5], the EDZL algorithm [10], and the Ehd2-SIP algorithm [12] are known to perform relatively well with small numbers of task preemptions, though their utilization bounds are down to 50% in the worst case. In fact, no algorithms except for the Pfair, EKG, and LLREF algorithms have ever achieved worst-case utilization bounds over 50%.

All the algorithms introduced above are categorized into dynamic-priority scheduling. In general, dynamic-priority scheduling suffers from the domino-effect problem that a deadline miss of a job causes another deadline miss of a following job. It also has a disadvantage of variational jitters in periodic executions, which are not desired in embedded control applications. The primary advantage of dynamic-priority scheduling has been its ability of achieving high utilization bounds, however such an advantage is not likely to stand on multiprocessors. Meanwhile, static-priority scheduling does not suffer from the domino-effect and periodic jitter problems, though its achievable utilization bound has been inferior to dynamic-priority scheduling. In recent years, Andersson *et al.* proved that static-priority scheduling is also able to achieve a utilization bound of 50% on multiprocessors, though no higher bounds cannot be obtained [3]. Therefore, this paper considers that static-priority scheduling may be more efficient than dynamic-priority scheduling for multiprocessor platforms.

This paper proposes an efficient real-time scheduling algorithm for multiprocessor platforms. The algorithm is a derivative of the RM algorithm. The objective of the algorithm to achieve high schedulable utilizations, i.e. system utilizations in which all recurrent real-time tasks are guaranteed to meet deadlines, with a small number of task preemptions. The worst-case utilization bound is also achieved 50%. In addition, this paper aims the theoretical design of the algorithm being implementable for practical use.

2 Related Work

Traditionally, there have been two approaches for scheduling real-time tasks on multiprocessors: *global scheduling* and *partitioned scheduling*. In global scheduling, all eligible tasks are stored in a single priority-ordered queue, and a global scheduler dispatches the same number of the highest priority tasks as processors from this queue. The relative order of the task priorities varies depending on which tasks are eligible, hence a task may migrate among processors. Meanwhile in partitioned scheduling, each task

*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.

is assigned to a single processor, on which each of its jobs will be executed, and processors are scheduled independently. Therefore, a task is executed on a dedicated processor and never migrates among processors.

Letting M be the number of processors, the global RM algorithm can miss a deadline even for the case in which tasks utilize only $1/M$ of the system [11]. Baker generalized that a set of periodic tasks, all with deadlines equal to periods, is guaranteed to be schedulable using the RM algorithm, if the total utilization of the tasks does not exceed $M(1-u_{max})/2+u_{min}$, where u_{max} and u_{min} are the maximum and minimum utilizations of every individual task respectively [6]. Andersson *et al.* invented a global scheduling algorithm called RM-US$[M^2/(3M-2)]$ [2], which places the highest priority to the tasks with utilizations higher than $M^2/(3M-2)$ and places the RM priorities to the other tasks. This prioritization boosts the worst-case utilization bound to $M/(3M-2)$. Hereinafter, RM-US$[M^2/(3M-2)]$ is denoted by RM-US for simplicity of description. Ramamurthy *et al.* proposed another global scheduling algorithm called Weight-Monotonic (WM) [21], which extends the Pfair scheduling method to take the static-priority assignment. In [3], Andersson *et al.* proved that the worst-case utilization bound of the WM algorithm is 50% that is higher than the RM and RM-US algorithms and no static-priority algorithms can transcend this bound. However, the WM algorithm generates a large number of task preemptions due to the characteristic of Pfair scheduling.

In general, partitioned scheduling approaches are preferred to global scheduling approaches for practical use, because no task migrations occur in partitioned scheduling and the scheduling problem can be reduced to a set of uniprocessor ones once the tasks are partitioned. The most well-known algorithm is RM-FF [11], which use the First-Fit (FF) heuristic to partition the tasks. Appending a little complexity, the RM-FFDU algorithm [19], which conducts the FF heuristic after sorting the tasks in decreasing order of utilizations, usually performs better than the RM-FF algorithm. R-BOUND-MP [14] is another efficient algorithm that takes similar schedulability tests and heuristics, in which tasks are initially sorted in increasing order of periods. Oh and Baker proved that a task set is guaranteed to be schedulable by the RM-FF algorithm if the system utilization does not exceed $\sqrt{2}-1 \simeq 41\%$ [18]. More recently, because 41% was a lower bound and there was still room for improvement, Lopez *et al.* presented a tighter utilization bound for the RM-FF algorithm [17]. Using a similar technique, Lopez *et al.* also clarified the utilization bound of the RM-FFDU algorithm [16]. Andersson *et al.* considered using the next-fit-ring (NFR) heuristic for the R-BOUND-MP algorithm instead the FF heuristic, and then derived its worst-case utilization bound of 50% [3].

3 System Model

The system is a memory-shared multiprocessor composed of M processors: $P_1, P_2, ..., P_M$. The code and data of programs (tasks) are shared among the processors. The overhead of inter-processor communications is neglected. In other words, the cost of task migrations is not taken into account. Recent advancements of processor technology have somewhat allowed such an assumption. For example, the responsive multithreaded (RMT) processor, invented by Yamasaki [22], supports a hardware function to switch a context to another context in four clock cycles regardless the switching occurs whether between processors or within a processor. In addition, there is little point in taking notice of the cost of every context switch in terms of scheduling algorithms, since the cost highly depends on processor specifications. Thus, it is more variant to focus on how often context switches occur for the discussion of the run-time overhead. This paper takes the number of context switches as a performance metric of the run-time overhead. This paper also ignores the behavior of a cache system. The performance deterioration of computations due to transient degradation of the cache hit ratio, often caused by migrations, is out of focus. This sorts of concerns are turned over to the worst-case execution time analysis.

The system has a set of N periodic tasks, denoted by $\Gamma = \{\tau_1, \tau_2, ..., \tau_N\}$. The ith periodic task is defined by $\tau_i(C_i, T_i)$ where C_i is its worst-case execution time and T_i is its period ($C_i \leq T_i$). The processor utilizaion of τ_i is defined by $U_i = C_i/T_i$. A task generates a sequence of jobs periodically. The jth job of τ_i is denoted by $\tau_{i,j}$ that is released at time $r_{i,j}$ and its deadline is equal to the release time of the next job, i.e. $d_{i,j} = r_{i,j+1} = r_{i,j} + T_i$. The total utilization of the give task set is defined by $U(\Gamma) = \sum_{\tau_i \in \Gamma} U_i$. Also, letting Λ be any subset of the given task set, the total utilization of the subset is denoted by $U(\Lambda) = \sum_{\tau_i \in \Lambda} U_i$. The set of the tasks executed on a processor P_x is denoted by Π_x. All the tasks are preemptive and independent. Therefore, no tasks synchronize with any task, and make critical sections such as I/O processing. Any job of a task cannot be executed in parallel, which means that, for any i and j, $\tau_{i,j}$ cannot be executed in parallel on more than one processor.

4 Scheduling Algorithm

This section proposes the *Rate Monotonic Deferrable Portion* (RMDP) scheduling algorithm that is a derivative of the RM algorithm [15], with its basis on the portioned scheduling technique [12]. The algorithm is designed well implementable for practical use. In the following subsections, the basic strategy of the portioned scheduling technique is first introduced. Then, the theoretical design of the RMDP algorithm is presented. The schedulable condition and the worst-case utilization bound of the RMDP algorithm are finally analyzed.

4.1 Portioned Scheduling Technique

The portioned scheduling strategy [12] is formed of the task assigning phase and the task scheduling phase. In the task assigning phase, each task is assigned to a particular processor like the partitioning approach, as long as the task does not cause the total utilization of the processor to ex-

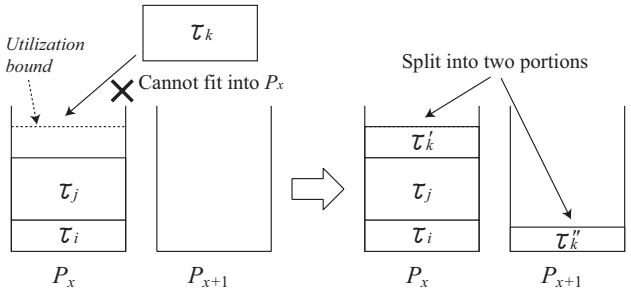

Figure 1. Portioning example

ceed its utilization bound. If the total utilization surpasses the utilization bound, the task is virtually split into two portions in the sense of utilization, whereas in the partitioning approach the task is just assigned to another processor which can receive the full utilization of the task. As for the virtually-split task, one portion is assigned to the processor to which the tasks are being assigned, and the other portion is assigned to the next processor to which the following tasks will be assigned. Notice that "virtually-split" means that the task is not really divided into two blocks, but its utilization is shared on the two processors. In this paper, such a task splitting is defined *portioning*.

Figure 1 shows an example of portioning. The height of a box in which the name of a task is indicated is the processor utilization of the task. The example presumes that τ_i and τ_j are already assigned to P_x and τ_k is about to be assigned, but if assigned it causes the total utilization of P_x to exceed its utilization bound. The partitioning approach in this case just assigns τ_k to another processor, such as P_{x+1}, while the portioning approach virtually splits τ_k into two portions τ'_k and τ''_k. In this paper, τ'_k is defined the *first portion* of τ_k and τ''_k is defined the *second portion* of τ_k. Then, τ'_k is assigned to P_x and τ''_k is assigned to P_{x+1} respectively. As a result, the portioning approach obviously improves the utilization of P_x compared to the partitioning approach.

In the task scheduling phase, τ_k is served by τ'_k on P_x and is served by τ''_k on P_{x+1}. In other words, letting C'_k and C''_k be the assigned execution times of τ'_k and τ''_k respectively, τ_k consumes C'_k time units on P_x and C''_k time units on P_{x+1} within every period T_k. The execution times of τ'_k and τ''_k are computed as follows.

$$\begin{aligned} C'_k &= T_k(U_b - U_i - U_j) \\ C''_k &= C_k - C'_k \end{aligned}$$

Note that τ'_k and τ''_k form the same task τ_k, thereby they cannot be scheduled simultaneously, since no jobs have parallelism under the assumption. Hence, the scheduling algorithm must be designed so that the first portion and the second portion of a split task are scheduled exclusively.

4.2 Task Assigning Algorithm

The task assigning algorithm of RMDP is straightforward. The algorithm assumes that the given task set is sorted so that the period of τ_i is smaller than or equal to that of τ_{i+1} for any i. Then, the algorithm assigns the tasks to the processors sequentially, and if some task causes the total utilization of a processor to exceed its utilization bound, it splits the task into two portions. The first portion is assigned to the processor which is caused to exceed its utilization bound, and the second portion is assigned to the next processor to which the following tasks will be assigned. This procedure is repeated until all the tasks are successfully assigned or no processors remain the spare capacities.

Assumption:
i is the index of the tasks.
x is the index of the processors.
n is the number of the tasks assigned to P_x.
$cs1$ is the execution time of the first portion.
$cs2$ is the execution time of the second portion.
ts is the period of a split task.
tm is the minimal period of the tasks assigned to P_x.
$U^*(P_x)$ is the utilization bound of P_x.
Γ is sorted so that $T_1 \leq T_2 \leq ... \leq T_n$.

```
1.  i = x = n = 1
2.  cs1 = cs2 = ts = tm = 0
3.  ∀Π_x = ∅
4.  U*_x = rmdp_bound(n, T_i, cs1, cs2, ts, tm)
5.  if U(Π_x) + U_i ≤ U*_x
6.     Π_x = Π_x ∪ τ_i
7.  else if x < M
8.     C'_i = {U*_x − U(Π_x)}T_i and C''_i = C_i − C'_i
9.     split τ_i into τ'_i(C'_i, T_i) and τ''_i(C''_i, T_i)
10.    Π_x = Π_x ∪ τ'_i and Π_{x+1} = {τ''_i}
11.    x = x + 1
12.    n = 0
13.    if i + 1 ≤ N
14.       cs1 = C'_i, cs2 = C''_i, ts = T_i, tm = T_{i+1}
15. else
16.    return FAILURE
17. i = i + 1
18. n = n + 1
19. if i ≤ N
20.    go back to step 4
21. return SUCCESS
```

Figure 2. RMDP assigning algorithm

Figure 2 shows the pseudo code of the RMDP assigning algorithm. When the initializations are done (lines 1 to 3), the algorithm calls the *rmdp_bound* function, which is indicated in Figure 3, to calculate the utilization bound of a processor P_x to which a task τ_i will be assigned (line 4). The content of the *rmdp_bound* function, i.e. the method of calculating the utilization bound, is particularly described in Section 5. Once the utilization bound U^*_x is calculated, the algorithm assigns τ_i to a processor P_x, as long as the total utilization of P_x will not exceed the utilization bound (lines 5 to 6). If τ_i causes P_x to exceed the utilization bound, τ_i is going to be split into τ'_i and τ''_i according to the following procedure. At first, the execution times of τ'_i and τ''_i are calculated (line 8). Then, τ_i is split into $\tau'_i(C'_i, T_i)$ and

Arguments: $(n, T_i, C'_s, C''_s, T_s, T_{min})$	
1.	**if** $C''_s = 0$
2.	return $n(2^{1/n} - 1)$
3.	**else**
4.	$L = \lceil (T_i - T_s + C'_s)/T_s \rceil$
5.	$U''s = C''_s/T_s$
6.	$R_s = T_{min}/T_s$
7.	return $U''_s + n\{(2 - LU''_s/R_s)^{1/n} - 1\}$

Figure 3. *rmdp_bound* **function**

$\tau''_i(C''_i, T_i)$ (line 9). τ'_i is assigned to P_x and τ''_i is assigned to P_{x+1} (line 10). The algorithm saves the values of C'_i,C''_i, T_i and T_{i+1} (line 14) for carrying out the *rmdp_bound* function in the next iteration. Finally, the algorithm goes back to the step 4 (line 20) as long as there are still tasks remaining to assign to the processors. When all the tasks are successfully assigned, the algorithm is succeeded (line 21).

Figure 4 depicts an example of task assignments by the RMDP assigning algorithm. Consider a task set Γ composed of eight tasks: $T_i \leq T_{i+1}$: $\tau_1(1,5)$, $\tau_2(2,5)$, $\tau_3(1,8)$, $\tau_4(5,10)$, $\tau_5(3,12)$, $\tau_6(2,12)$, $\tau_7(12,20)$, and $\tau_8(4,20)$. Note that Γ is already sorted so that $T_i \leq T_{i+1}$ is satisfied for any i where ties of periods are broken arbitrarily. As stated above, the algorithm sequentially assigns the tasks. Evaluating the *rmdp_bound* function with $n = 1$, the utilization bound of 100% is obtained, and hence τ_1 can be assigned to P_1. Next, the algorithm is supposed to evaluate the *rmdp_bound* function with $n = 2$. According to Kuo and Mok [13], the number of the tasks is reduced to the number of the harmonic chains in RM scheduling. A harmonic chain is a group of the tasks whose periods have the same least common multiple. Taking the harmonic chains into account, the 23rd line of Figure 2 can be extended so that it is carried out only if the period of the next task is not included in any harmonic chains. Thus, the example evaluates the *rmdp_bound* function with $n = 1$ due to $T_1 = T_2$, and τ_2 can be of course assigned to P_1. τ_3 does not have a period with the same least common multiple as τ_1 and τ_2, then $n = 2$ is input to the *rmdp_bound* function, while τ_4 has a period with the same least common multiple as them, and hence $n = 2$ is remained to the *rmdp_bound* function. The resulting utilization bound of 0.83 is obtained by the *rmdp_bound* function for $n = 2$. Since $U_1 + U_2 + U_3 + U_4 = 1.225$ exceeds 0.83, τ_4 is split into $\tau'_4(1,10)$ and $\tau''_4(4,10)$ so that $U_1 + U_2 + U_3 + U'_4 = 0.825 \leq 0.83$ and $U''_4 = U_4 - U'_4$. By the same token, the rest of the tasks are classified into $\Pi_2 = \{\tau''_4(4,10), \tau_5, \tau'_6(1,12)\}$ and $\Pi_3 = \{\tau''_6(1,12), \tau_7, \tau_8\}$, though the utilization bounds are calculated with taking into account that split tasks are assigned in those sets.

4.3 Task Scheduling Algorithm

This section presents the task scheduling algorithm of RMDP. Like partitioned scheduling, each processor has its own scheduler to schedule the assigned tasks on the processor. All the schedulers have the same scheduling poli-

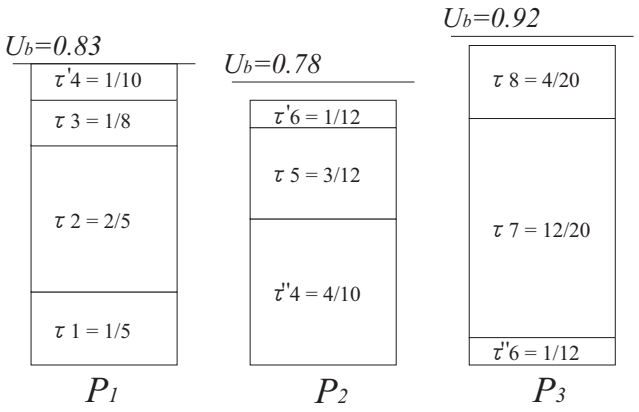

Figure 4. RMDP assigning example

cies. The following descriptions focus on the scheduling on a processor P_x where a task τ_i submits its second portion τ''_i and another task τ_k submits its first portion τ'_k. The schedulers on other processors perform in the same manner, since the same scheduling policies are applied.

The basic policy of the scheduling algorithm is that the tasks are scheduled according to the RM algorithm except for the case in which τ''_i is ready but its corresponding first portion τ'_i is on execution on a neighbor processor P_{x-1}. In this case, the RMDP scheduler dispatches a task with the second highest priority so as not to execute the first and the second portions of τ_i simultaneously. In other words, the second portion is deferred with the highest priority. Therefore, the prioritization discipline of the RMDP algorithm is not static in a true sense.

Figure 5 shows the pseudo code of the RMDP scheduler. Every time any tasks are released on P_x, the *schedule_*P_x function is invoked (lines 1 to 7). Let t be such a time. At time t, if the split tasks τ_i and τ_k are released, the scheduler needs to reset the variables t_i, t_k, e_i and e_k, in order to track the remaining execution times of those tasks. Note that τ_i and τ_k are not allowed to consume the processor time of P_x over the capacities C''_i and C'_k respectively, but the tasks cannot recognize voluntarily that they exhaust the capacities. Thus, the scheduler needs to track the remaining execution times of those tasks to preempt their executions using timer functions.

The scheduler function proceeds as follows. First of all, if the currently-running task is either a split portion of τ_i or τ_j, the scheduler saves its remaining execution time (lines 11 to 13). Then, it selects a task with the shortest period (line 15). If the selected highest-priority task is τ_i that has its second portion on the processor (line 16), the scheduler examines whether τ_i is currently in execution on a neighbor processor P_{x+1} (line 17). If it is in execution, the scheduler re-selects a task with the second shortest period (line 18). Otherwise, τ_i is going to be dispatched and executed, so the scheduler updates the timer to invoke the *second_end* function at time $t + e_i$, which preempts the execution of τ_i (line 20). Notice that the *second_end* function may not be invoked at time $t + e_i$, since it may be updated if τ_i is preempted and later dispatched again within the same period.

Assumption:
τ_i is a split task between P_{x-1} and P_x.
τ_k is a split task between P_x and P_{x+1}.
t is the current time.
t_i is the last time at which τ_i is dispatched on P_x.
t_k is the last time at which τ_k is dispatched on P_x.
e_i is the remaining execution time of τ_i.
e_k is the remaining execution time of τ_k.

```
1.  function system_tick
2.    if any tasks are released on P_x
3.      if τ_i is released
4.        e_i = C_i'' and t_i = t
5.      if τ_k is released
6.        e_k = C_k' and t_k = t
7.      call schedule_P_x
8.  function schedule_P_x
9.    if P_x is idling
10.     go to step 15
11.   if τ_i is currently running
12.     e_i = e_i − (t − t_i)
13.   else if τ_k is currently running
14.     e_k = e_k − (t − t_k)
15.   let τ_j be a task with the shortest period
16.   if τ_j refers to τ_i
17.     if τ_i is in execution on P_{x−1}
18.       let τ_j be a task with the second shortest period
19.     else
20.       update the timer to invoke p2_finished at t + e_i
21.       t_i = t
22.   else if τ_j refers to τ_k
23.     update the timer to invoke p1_finished at t + e_k
24.     t_k = t
25.     if τ_k is in execution on P_{x+1}
26.       invoke schedule_P_{x+1} on P_{x+1}
27.   dispatch and execute τ_j on P_x
```

Figure 5. RMDP scheduling algorithm

Time $t + e_i$ is just the earliest time at which τ_i can exhaust C_i''. As long as the remaining execution time of τ_i is tracked, τ_i is guaranteed not to overrun the capacity C_i'' on P_x. After updating the timer, the scheduler must save the current time as the last dispatched time of τ_i for tracking its remaining execution time. By the same token, if the selected task is τ_k that has its first portion on the processor (line 22), the scheduler updates the other timer to invoke the *first_end* function at time $t + e_k$, which preempts the execution of τ_k, and saves the current time as the last dispatched time of τ_k (lines 23 to 24). Unlike the case of τ_i, if the corresponding second portion of τ_k is currently in execution on P_{x+1} (line 25), the scheduler needs to invoke the scheduler function that is operating on P_{x+1} to let it reschedule the tasks on P_{x+1} (line 26), since the second portion must be deferred while the corresponding first portion is executed. Finally, the scheduler executes the selected task on P_x (line 27).

The job finishing functions are indicated in Figure 6. When any jobs except for those of the split tasks are completed, the jobs call the *job_end* function (lines 1to 3). The *first_end* and *second_end* functions are called only through the timers. Those functions call the scheduler to reschedule

```
1.  function job_end
2.    remove the caller task from a ready set
3.    call schedule_P_x
4.  function first_end
5.    remove τ_r from a ready set
6.    call schedule_P_x
7.    if τ_k is ready on P_{x+1}
8.      invoke schedule_P_{x+1} on P_{x+1}
9.  function second_end
10.   remove τ_i from a ready set
11.   call schedule_P_x
```

Figure 6. Job finishing functions

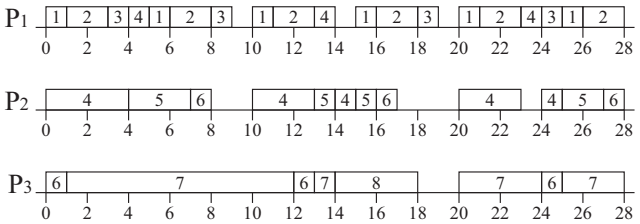

Figure 7. RMDP scheduling example

the tasks (lines 3, 6, and 11). Only when τ_k consumes the capacity C_k' on P_x but has not consumed the capacity C_k'' on P_{x+1}, the scheduler invokes the scheduler operating on P_{x+1} to execute τ_k on P_{x+1}, since τ_k has the highest priority on P_{x+1} but has been deferred.

Figure 7 indicates how the three task sets of $\Pi_1 = \{\tau_1(1,5),\ \tau_2(2,5),\ \tau_3(1,8),\ \tau_4'(1,10)\}$, $\Pi_2 = \{\tau_4''(4,10),\ \tau_5(3,12),\ \tau_6'(1,12)\}$ and $\Pi_3 = \{\tau_6''(1,12),\ \tau_7(12,20),\ \tau_8(4,20)\}$, are scheduled by the RMDP algorithm. Since Π_1 does not include the second portion of a split task, the task set is scheduled completely according to the RM policy. Meanwhile, Π_2 includes τ_4'' and it is interfered by its corresponding first portion τ_4'. At time $t = 0$, τ_4'' is scheduled on P_2 and then completed at time $t = 4$ without any interference from τ_4', because τ_4' is scheduled at time $t = 4$ on P_1. As for the second job of τ_4, on the other hand, τ_4'' is scheduled at time $t = 10$ on P_2 but is preempted at time $t = 13$, because τ_4' is scheduled at this time on P_1. Hence, τ_5 is scheduled instead of τ_4'' and τ_4'' is later resumed when τ_4' is completed at time $t = 14$. The third job of τ_4 also has the same situation. In this case, there are no ready tasks on P_1 when τ_4'' is preempted at time $t = 23$. Therefore, the time slot is left idle. Although Π_3 also includes the second portion τ_6'', the tasks on P_3 can be scheduled without any restrictions within the example, since τ_6' and τ_6'' are never overlapped in RM scheduling.

5 Schedulability Analysis

This section provides the schedulability analysis of the RMDP algorithm to guarantee the worst-case performance. The analysis has the following assumptions. In the task assigning phase, some task τ_s is split and its second portion τ_s'' is assigned to a processor P_x. Then, the following n

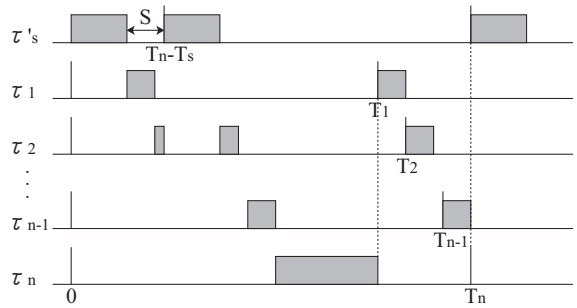

Figure 8. Case in which τ_s'' is executed twice within T_1 and T_n

tasks, denoted by $\tau_1, \tau_2, ..., \tau_n$ for simplicity of description, are also assigned to P_x. Note that the first portion of a split task is not a concern, because the RMDP scheduler treats it in the same manner as the non-split tasks. The goal of the analysis is to derive a formula leading to the schedulable condition of P_x for any x.

According to the RM analysis [15], the minimal schedulable utilization for n tasks occurs for the case in which all the tasks are released at the same time with the following relations: $T_1 < T_n < 2T_1$, $C_i = T_{i+1} - T_i$ $(1 \le i \le n-1)$ and $C_n = 2T_1 - T_n$. Thus, the minimal schedulable utilization for non-split n tasks in the RMDP algorithm also occurs for this condition, those tasks are scheduled according to the RM algorithm. Now therefore, the worst-case phasing of τ_s'' is a concern. For a fixed value of U_s'', the remaining utilization of P_x is obviously minimized when τ_s'' submits as many jobs as possible. That occurs for the case in which an arbitrary job of τ_s'' is deferred as much as possible and its following jobs are executed without any preemptions as soon as they are released. Such worst-case phasing is classified into three cases, shown in Figure 8, Figure 9 and Figure 10. The difference between the three phases is that τ_s'' is executed (i) twice within T_1 and T_n in the phase 1, (ii) three times within T_1 and T_n in the phase 2, and (iii) twice with in T_1 and three times within T_n in the phase 3. The analysis does not need to consider the case in which τ_s'' is executed more than three times within T_1 or T_n, since the condition of the worst-case phasing includes $T_1 < T_n < 2T_1$.

For simplicity of description, the fraction of T_s and T_1 is defined $R_s = T_1/T_s$ and that of the periods of two consecutive tasks τ_i and τ_{i+1} is defined $R_i = T_{i+1}/T_i$ henceforth. Notice that $\forall k, R_1R_2\cdots R_{k-1} = T_k/T_1$. Also it is necessary to have the minimum slack S in the figures to obtain the schedulable utilization. Since τ_s'' has the highest priority on the processor, it is only deferred by τ_s' scheduled on the neighbor processor. Therefore, the latest finish time of τ_s'' is $r_{s,k} + C_s' + C_s'' = r_{s,k} + C_s = r_{s,k+1} - (T_s - C_s)$. As a result, the minimal slack can be expressed by $S = T_s - C_s$.

5.1 Case of Phase 1

The analysis begins with the simplest case in which τ_s'' is executed twice within T_1 and T_n as shown in Figure 8. Under this relation, the execution times of the n tasks are denoted as follows.

$$\begin{aligned} C_i &= T_{i+1} - T_i \ \ (1 \le i \le n-1) \\ C_n &= T_1 - 2C_s'' - \textstyle\sum_{j=1}^{n-1} C_j = 2T_1 - 2C_s'' - T_n \end{aligned}$$

Hence, the resulting utilization is written as Equation (1).

$$\begin{aligned} U &= \frac{C_s''}{T_s} + \frac{C_1}{T_1} + \frac{C_2}{T_2} + \cdots + \frac{C_n}{T_n} \\ &= U_s'' + \sum_{i=1}^{n-1} \frac{T_{i+1} - T_i}{T_i} + \frac{2T_1 - 2C_k'' - T_n}{T_n} \\ &= U_s'' + \sum_{i=1}^{n-1} \frac{T_{i+1}}{T_i} + 2\left(1 - \frac{C_s''}{T_1}\right)\frac{T_1}{T_n} - n \\ &= U_s'' + \sum_{i=1}^{n-1} R_i + \frac{2\left(1 - \frac{U_s''}{R_s}\right)}{R_1R_2\cdots R_{n-1}} - n \end{aligned} \quad (1)$$

In order to minimize U over R_i, the above expression is partially differentiated with respect to R_i.

$$\frac{\partial U}{\partial R_i} = 1 - \frac{2\left(1 - \frac{U_s''}{R_s}\right)}{R_i^2(\prod_{j\ne i}^{n-1} R_j)}$$

Now, U is minimized when each R_i satisfies the following equation where $P = R_1R_2\cdots R_{n-1}$.

$$R_iP = 2\left(1 - \frac{U_s''}{R_s}\right) \ (1 \le i \le n-1)$$

That is, U is minimized when all the R_i have the same value.

$$R_1 = R_2 = \cdots = R_{n-1} = \left\{2\left(1 - \frac{U_s''}{R_s}\right)\right\}^{1/n}$$

By substituting the value of each R_i to Equation (1), the utilization bound U_b is obtained as follows. Here, let $K = 2(1 - U_s''/R_s)$ due to limitation of space.

$$\begin{aligned} U_b &= U_s'' + (n-1)K^{1/n} + \frac{K}{K^{n-1/n}} - n \\ &= U_s'' + nK^{1/n} - K^{1/n} + K^{1/n} - n \\ &= U_s'' + n\left\{\left(\frac{2(1 - U_s'')}{R_s}\right)^{1/n} - 1\right\} \end{aligned}$$

The above expression implies that U_b is moreover minimized by reducing the value of R_s. According to Figure 8, the condition below must be satisfied.

$$\sum_{i=1}^{n} C_i = T_1 - 2C_s'' \quad \ge \quad S = T_s - C_s$$

Dividing by T_s, the range of R_s is acquired as follows.

$$\begin{aligned} R_s - 2U_s'' &\ge 1 - U_s \\ R_s &\ge 2U_s'' - U_s + 1 \end{aligned}$$

The condition of $T_s \le T_1$ leads to $R_s \ge 1$. Hence, U_b is minimized when $R_s = \max\{1, 2U_s'' - U_s + 1\}$. Finally, U_b is described by Equation (2) where $R_s = \max\{1, 2U_s'' - U_s + 1\}$.

$$U_b = U_s'' + n\left[\left\{2\left(1 - \frac{U_s''}{R_s}\right)\right\}^{1/n} - 1\right] \quad (2)$$

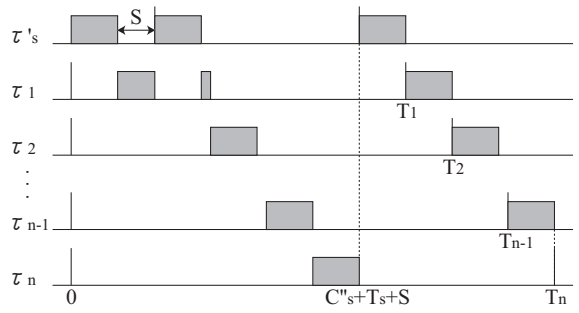

Figure 9. Case in which τ_s'' is executed three times within T_1 and T_n

Taking the limit as $n \to \infty$, the worst case is derived.

$$\lim_{n\to\infty} U_b = U_s'' + \ln\left\{2\left(1 - \frac{U_s''}{R_s}\right)\right\} \quad (3)$$

Equation (3) is a monotonically-increasing function with respect to R_s. Since $T_s \le T_1$, R_s is never less than 1. Hence, Equation (3) is minimized to Equation (4) with $R_s = 1$.

$$\hat{U}_b = U_s'' + \ln\{2(1 - U_s'')\} \quad (4)$$

Consequently, the absolute minimum value of the utilization bound becomes $\hat{U}_b = 0.5$ with $U_s'' = 0.5$ and $U_s = 1$. Note that this bound is given only for the situation in which $T_s = T_1 = T_2 = \cdots = T_n$ and $C_1 = C_2 = \cdots = C_n = 0$.

5.2 Case of Phase 2

The analysis moves on the case in which τ_s'' is executed three times within T_1 and T_n as shown in Figure 9. The execution times of the n tasks are defined as follows.

$$\begin{aligned} C_i &= T_{i+1} - T_i \ (1 \le i \le n-1) \\ C_n &= T_1 - 3C_s'' - \textstyle\sum_{j=1}^{n-1} C_j = 2T_1 - 3C_s'' - T_n \end{aligned}$$

Hence, the resulting utilization is written as Equation (5).

$$\begin{aligned} U &= U_s'' + \sum_{i=1}^{n-1} \frac{T_{i+1}}{T_i} + \left(2 - \frac{3C_s''}{T_1}\right)\frac{T_1}{T_n} - n \\ &= U_s'' + \sum_{i=1}^{n-1} R_i + \frac{2 - \frac{3U_s''}{R_s}}{R_1 R_2 \cdots R_{n-1}} - n \end{aligned} \quad (5)$$

The value of R_i that minimizes Equation (5) can be calculated by the same step in Section 5.1. That is, the utilization bound is obtained by the following expression.

$$U_b = U_s'' + n\left\{\left(2 - \frac{3U_s''}{R_s}\right)^{1/n} - 1\right\}$$

Notice that the above expression is a monotonically-increasing function with respect to R_s. Hence, it is minimized by reducing the value of R_s. In order to find the minimum value of R_s, the value of T_1 must be explored first. Figure 9 leads to T_1 as follows.

$$T_1 = T_s + 2C_s'' + S = 2T_s + 2C_s'' - C_s$$

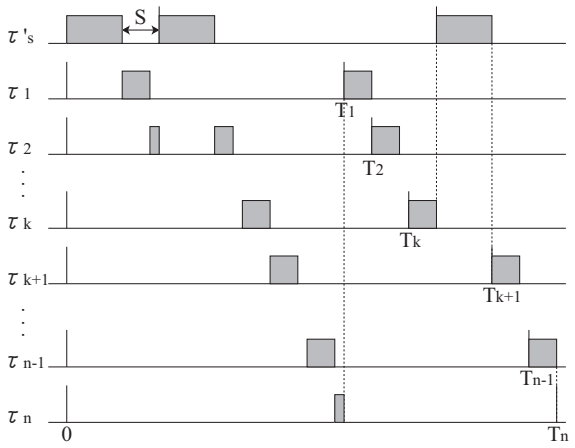

Figure 10. Case in which τ_s'' is executed twice within T_1 and three times within T_n

Dividing by T_s, the value of R_s is acquired as follows.

$$R_s = 2U_s'' - U_s + 2$$

Hence, U_b is now expressed by Equation (6).

$$\begin{aligned} U_b &= U_s'' + n\left\{\left(2 - \frac{3U_s''}{2U_s'' - U_s + 2}\right)^{1/n} - 1\right\} \\ &= U_s'' + n\left\{\left(\frac{U_s'' - 2U_s + 4}{2U_s'' - U_s + 2}\right)^{1/n} - 1\right\} \end{aligned} \quad (6)$$

Taking the limit as $n \to \infty$, the worst case is appeared.

$$\lim_{n\to\infty} U_b = U_s'' + \ln\left(\frac{U_s'' - 2U_s + 4}{2U_s'' - U_s + 2}\right) \quad (7)$$

In order to find the value of U_s'' minimizing U_b, Equation (7) is differentiated with respect to U_s''.

$$\frac{\partial U_b}{\partial U_s''} = \frac{2U_s''^2 + (10 - 5U_s)U_s'' + 2U_s^2 - 5U_s + 2}{(U_s'' - 2U_s + 4)(2U_s'' - U_s + 2)}$$

Finally, the value of U_s'' that minimizes U_b is obtained by Equation (8).

$$\hat{U}_s'' = \frac{5U_s - 10 + \sqrt{9U_s^2 - 60U_s + 84}}{4} \quad (8)$$

Since $U_s'' \ge 0$, the absolute minimum value of the utilization bound becomes $\hat{U}_b \simeq 0.652$ with $U_s'' \simeq 0.186$ and $U_s = 1$. Since the previous case has a lower utilization bound of 50%, this case is not the worst case.

5.3 Case of Phase 3

The final case the analysis considers is a combination of the previous two cases. Since τ_s'' is executed twice within T_1 while is executed three times within T_n, the execution times of the n tasks are denoted as follows.

$$\begin{aligned} C_i &= T_{i+1} - T_i \ (1 \le i \le k-1) \\ C_k &= T_{k+1} - C_s'' - T_k \\ C_i &= T_{i+1} - T_i \ (k+1 \le i \le n-1) \\ C_n &= T_1 - 2C_s'' - \textstyle\sum_{j=1}^{n-1} C_j = 2T_1 - C_s'' - T_n \end{aligned}$$

Then, the resulting utilization is written as Equation (9).

$$\begin{aligned} U &= U_s'' + \sum_{i=1}^{k-1} \frac{T_{i+1} - T_i}{T_i} + \frac{C_k}{T_k} + \sum_{i=k+1}^{n-1} \frac{T_{i+1} - T_i}{T_i} + \\ &\quad \frac{2T_1 - C_s'' - T_n}{T_n} \\ &= U_s'' + \sum_{i=1}^{k-1} \frac{T_{i+1}}{T_i} + U_k + \sum_{i=k+1}^{n-1} \frac{T_{i+1}}{T_i} + \left(2 - \frac{C_s''}{T_1}\right)\frac{T_1}{T_n} - \\ &\quad (n-1) \\ &= U_s'' + \sum_{i=1}^{k-1} R_i + U_k + \sum_{i=k+1}^{n-1} R_i + \frac{2 - \frac{U_s''}{R_s}}{R_1 R_2 \cdots R_{n-1}} - \\ &\quad (n-1) \end{aligned} \quad (9)$$

Unfortunately, it is too complicated to explore the value of R_i that minimizes Equation (9) with respect to two variables of n and k. Therefore, let the value of k be fixed first. U_k is described by $U_k = R_k - C_s''/T_k - 1$ due to $C_k = T_{k+1} - C_s'' - T_k$, Equation (9) can be transformed as follows.

$$U = U_s'' + \sum_{i=1}^{n-1} R_i - \frac{C_s''}{T_k} + \frac{2 - \frac{U_s''}{R_s}}{R_1 R_2 R_3 \cdots R_{n-1}} - n$$

The above expression implies that Equation (9) is obviously minimized for $k = 1$ due to $T_1 \le T_2 \le \cdots \le T_n$. Therefore, Equation (9) is reduced as follows.

$$U = U_s'' + U_1 + \sum_{i=2}^{n-1} R_i + \frac{2R_s - U_s''}{R_s R_1 R_2 R_3 \cdots R_{n-1}} - (n-1)$$

In order to minimize U, the above expression is differentiated with respect to R_i as follows, where $2 \le i \le n-1$.

$$\frac{\partial U}{\partial R_i} = 1 - \frac{2R_s - U_s''}{R_s R_1 R_i^2 \prod_{j \ne i}^{n-1} R_j}$$

That is, U is minimized when all the R_i have the same value.

$$R_2 = R_3 = \cdots = R_{n-1} = \left(\frac{2R_s - U_s''}{R_s R_1}\right)^{1/(n-1)}$$

Now, the minimum value of U is described as follows.

$$\begin{aligned} U &= U_s'' + U_1 + (n-2)\left(\frac{2R_s - U_s''}{R_s R_1}\right)^{1/(n-1)} + \\ &\quad \frac{2R_s - U_s''}{R_s R_1 \left(\frac{2R_s - U_s''}{R_s R_1}\right)^{1-1/(n-1)}} - (n-1) \\ &= U_s'' + U_1 + (n-1)\left\{\left(\frac{2R_s - U_s''}{R_s R_1}\right)^{1/(n-1)} - 1\right\} \end{aligned}$$

$C_k = T_{k+1} - C_s'' - T_k$ and $k = 1$ lead to $R_1 = U_1 + U_s''/R_s + 1$. Also, the value of R_s that minimizes the above expression is $\max\{1, 2U_s'' - U_s + 1\}$ as in Section 5.1. Finally, the utilization bound U_b is described by Equation (10) where $R_s = \max\{1, 2U_s'' - U_s + 1\}$ and $m = n - 1$.

$$U_b = U_s'' + U_1 + m\left[\left\{\frac{2R_s - U_s''}{R_s(U_1 + 1) + U_s''}\right\}^{1/m} - 1\right] \quad (10)$$

Taking the limit as $n \to \infty$, the worst case is appeared.

$$\lim_{n\to\infty} U_b = U_s'' + U_1 + \ln\left\{\frac{2R_s - U_s''}{R_s(U_1 + 1) + U_s''}\right\} \quad (11)$$

By the same token as Section 5.1, the minimum of Equation (11) is expressed by Equation (12).

$$\hat{U}_b = U_s'' + U_1 + \ln\left(\frac{2 - U_s''}{U_s'' + 1 + U_1}\right) \quad (12)$$

Seeking the values of U_s'' and U_1 that minimizes $\hat{U}_b$, the absolute minimum bound is obtained $\hat{U}_b \simeq 0.5$ with $U_s'' = 1/2$ and $U_1 = 0$. This bound occurs only for the situation in which $T_s = T_1 = T_2 = \cdots = T_k$, $T_{k+1} = T_{k+2} = \cdots = T_n = T_s + C_s''$ and $C_1 = C_2 = \cdots = C_n = 0$. Comparing the analyzed three cases, the worst-case utilization bound of each processor is finally derived 50%. That is, the utilization bound of the entire system is also 50%.

5.4 General Case

The tasks are sorted so that $T_i \le T_{i+1}$, hence T_1 is a known value when τ_s is split. In other words, R_s is a known value when τ_s is split. Therefore, the worst-case is not necessarily presumed, but the utilization bound can be calculated by either Equation (2), Equation (6), or (10).

The analysis now proceeds to consider the general case. Equation (5) can be rewritten as follows.

$$U = U_s'' + \sum_{i=1}^{n-1} \frac{T_{i+1}}{T_i} + \left(2 - \frac{C_s''}{T_1}\right)\frac{T_1}{T_n} - n - \frac{2C_s''}{T_n} \quad (13)$$

Equation (9) can be also rewritten as follows.

$$U = U_s'' + \sum_{i=1}^{n-1} \frac{T_{i+1}}{T_i} + \left(2 - \frac{C_s''}{T_1}\right)\frac{T_1}{T_n} - n - \frac{\frac{T_n}{T_k}C_s''}{T_n} \quad (14)$$

Notice that T_n/T_k in Equation (14) never exceeds 2, because τ_s'' must be executed twice within T_k and three times within T_n. Hence, Equation (5) is always smaller than Equation (9) with respect to the same set of $\{T_s, T_1, T_2, ..., T_n\}$. This fact implies that the utilization bound for the case in which τ_s'' is executed L times within T_1 and T_n is always smaller than that for the case in which τ_s'' is executed $F < L$ times within T_1 and is executed L times within T_n. Therefore, the analysis needs to concern only the case in which τ_s'' is executed L times within T_n for the general case. Referring to Figure 9, L can be described by the following expression.

$$L = 1 + \left\lceil \frac{T_n - C_s'' - S}{T_s} \right\rceil = 1 + \left\lceil \frac{T_n - T_s + C_s'}{T_s} \right\rceil$$

Finally, the utilization utilization for the general case is obtained by Equation (15).

$$U_b = U_s'' + n\left\{\left(2 - \frac{LU_s''}{R_s}\right)^{1/n} - 1\right\} \quad (15)$$

The value of Equation (15) is 50% with $L = 2$, $U_s = 1$ and $U_s'' = 0.5$. Hence, the worst case is contained. Letting

$U''_s = 0$, the analysis also contains the case in which there is no task that has the second portion on P_x. In fact, this case leads to Equation (16) which is the well-known utilization bound of the RM algorithm [15].

$$U_b = n(2^{1/n} - 1) \quad (16)$$

6 Simulation Studies

This section evaluates the effectiveness of the RMDP algorithm in terms of the schedulability and the number of task preemptions. The utilization bound was proved in the previous section, and hence the RMDP algorithm can be compared with the traditional algorithms theoretically. However, in order to estimate the performances of the algorithms truly, the sufficient number of task sets with different properties must be submitted to the algorithms, since the utilization bounds of the algorithms actually vary depending on given task sets. The number of task preemptions are also dominated by the characteristics of the given task sets. The evaluations compare the RMDP algorithm with the traditional RM-based algorithms: RM-FF, RM-FFDU, R-BOUND-MP-NFR, RM, RM-US, and WM.

6.1 Experimental Setup

The simulations estimate the schedulability of an algorithm as follows. Every system utilization U_{sys} ranging from 30% to 100%, 1000 task sets with different properties, whose system utilizations are all U_{sys} equally, are generated and submitted to the algorithm. Then, the success ratio, defined by the following expression, is measured.

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

Algorithms having high success ratios are estimated to offer high schedulable utilizations. The definition of a successfully-scheduled task set depends on a scheduling algorithm. For the RMDP, RM-FF, RM-FFDU, and R-BOUND-MP-NFR algorithms, a task set is said to be successfully scheduled if all the tasks can be assigned to the processors, since those algorithms are designed so that no tasks will miss the deadline once they are successfully assigned to the processors. For the RM, RM-US and WM algorithms, on the other hand, a task set is said to be successfully scheduled if all tasks are scheduled without missing any deadlines, since the theoretical utilization bounds of those algorithms are very pessimistic. Meanwhile, the simulations estimate the number of preemptions for an algorithm by calculating its average number, defined by the following expression.

$$\frac{\text{the total number of preemptions in the scheduled task sets}}{\text{the number of scheduled task sets}}$$

Each simulation is characterized by the four parameters: M, U_{max}, U_{min} and U_{total}. M is the number of the processors. U_{max} and U_{min} are the maximum and minimum values of the processor utilization of every individual task in a given task set. U_{total} refers to the total processor utilization of the tasks. The system utilization is defined by $U_{sys} = U_{total}/M$, which ranges from 0% to 100%. Although many combinations of the parameters can be considered, the simulations attempt the following combinations due to the limitation of space. The system utilization is determined within the range of [30%, 100%]. Most of the existing algorithms can successfully schedule a task set with a system utilization below 30%, hence system utilizations below 30% are removed. As for the number of the processors, the simulations prepare the three sets: $M = 4$, $M = 8$, and $M = 16$. The target systems of this research, such as humanoid robots, would make use of multicore processors having such number of cores. In those systems, simple activities can be realized with only light tasks (tasks with low utilizations), whereas enhancing the quality of the activities requires heavy tasks (tasks with high utilizations). Thereby, the simulations prepare the two sets of U_{min} and U_{max}: $(U_{min}, U_{max}) = (0.01, 0.1)$ and $(U_{min}, U_{max}) = (0.01, 1.0)$.

6.2 Schedulability Results

Figure 11 shows the success ratios for each algorithm with respect to task sets in which the utilization of every individual task ranges from within $[0.01, 0.1]$. The R-BOUND-MP-NFR algorithm is abbreviated as R-BOUND-MP for simplicity. The schedulable utilizations of the RMDP algorithm are around $70 \sim 73\%$. The RM-FF, RM-FFDU and R-BOUND-MP algorithms are also competitive, achieving schedulable utilizations around $65 \sim 67\%$, though the RMDP algorithm slightly outperform the others. Therefore, the simulations demonstrate that the partitioned scheduling approaches have little performance difference to light task sets in which tasks have low processor utilizations. The RM and RM-US algorithms generate the same schedule for the case with $U_{max} = 0.1 < M/(3M - 2)$. It is remarkable that the RM and RM-US algorithms perform much better than the RMDP and partitioned scheduling algorithms. Since the schedulable utilizations of the RM and RM-US algorithms are dominated by the maximum utilization of every individual task, those algorithms offer excellent performance. However, remember that the timing constraints cannot be guaranteed by the RM and RM-US algorithms, if the system utilization exceeds $M/(3M - 2)$ [2]. Thus, those algorithms are not desired in safe systems. The WM algorithm achieves schedulable utilizations around 90%, which are much better than the other algorithms. Such a superiority of the WM algorithm is derived from the characteristic of Pfair scheduling, which can make an optimal schedule. Unfortunately, the WM algorithm is not implementable for practical use, since it may invoke a scheduler every quanta.

Figure 12 shows the success ratios for each algorithm with respect to task sets in which the utilization of every individual task ranges from within $[0.01, 1.0]$. Note that this ranging generates heavy tasks. Unlike the previous results in which only light tasks are generated, the RMDP algorithm clearly outperform the other algorithms except for the WM algorithm. The schedulable utilizations of the RMDP

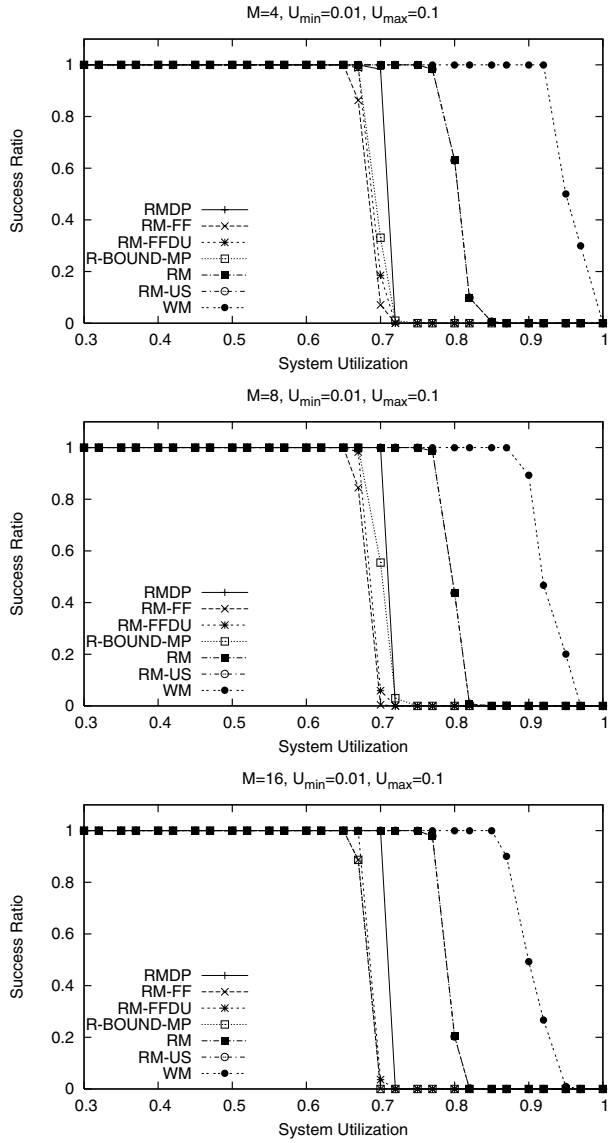

Figure 11. Success ratios ($U_{max} = 0.1$)

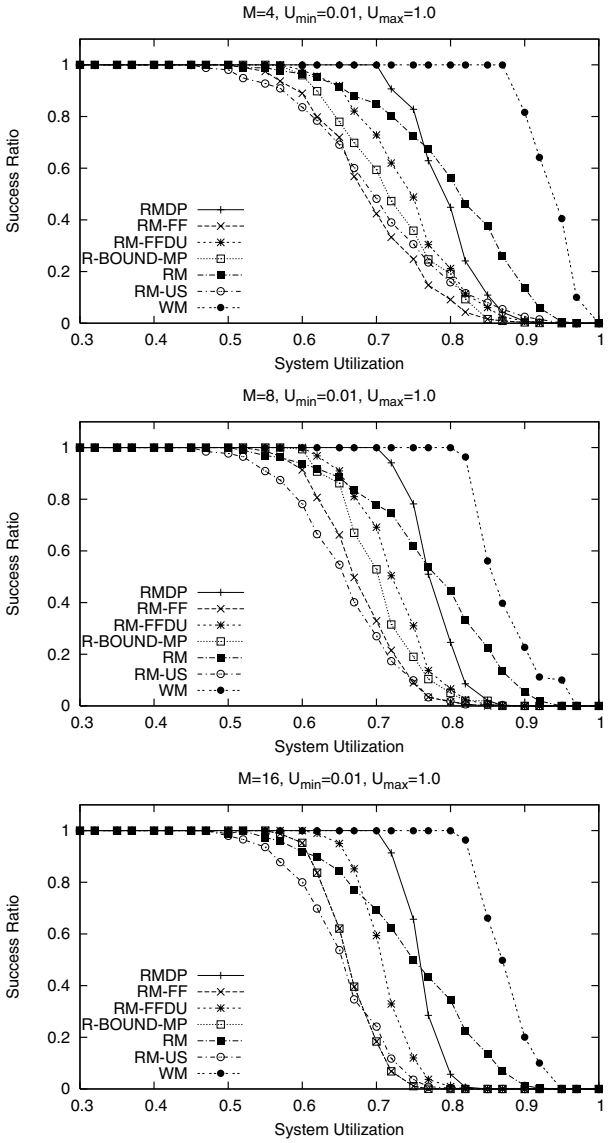

Figure 12. Success ratios ($U_{max} = 1.0$)

algorithm are around 70 $\sim$ 73% as well as the previous results. Meanwhile, the RM-FF, RM-FFDU and R-BOUND-MP algorithms have schedulable utilizations around 50 $\sim$ 60%, that is, the RMDP outperform about 10 $\sim$ 20% over the partitioned scheduling algorithms. Therefore, the simulations show that the effectiveness of portioned scheduling is likely to appear with task sets in which some tasks have high utilizations. The RM and RM-US algorithms, on the other hand, offer very low schedulable utilizations. Hence, the simulations evince that the performances of those algorithms are truly affected by heavy tasks. However, it is interesting results that the RM algorithm retains high success ratios in high system utilizations. It even trades the success ratios with the RMDP algorithm in system utilizations over 80%. As stated above, the RM algorithm does not conduct a schedulability test in the simulations, whereas the RMDP, RM-FF, RM-FFDU and R-BOUND-MP algorithms carrying out schedulability tests may reject task sets that can be in fact successfully scheduled. As a result, the four algorithms have lower success ratios than the RM algorithm in high system utilizations. Recall that the timing constraints cannot be guaranteed by the RM algorithm, if the system utilization exceeds $1/M$. It is surprising that the RM-US algorithm performs much worse than the RM algorithm. It can be considered that the RM-US algorithm disobeys the RM prioritization, thereby deadlines are more likely to be missed, though it improves the worst-case schedulability. The WM algorithm still offers good performance in the presence of heavy tasks in exchange for a great deal of computation complexity.

6.3 Preemption Results

For calculation of task preemptions, the simulation intervals are set the smaller of the least common multiple of the task periods in the given task set and 2^{32}. Then, for each algorithm, the average number of task preemptions in the 1000 task sets is calculated every system utilization.

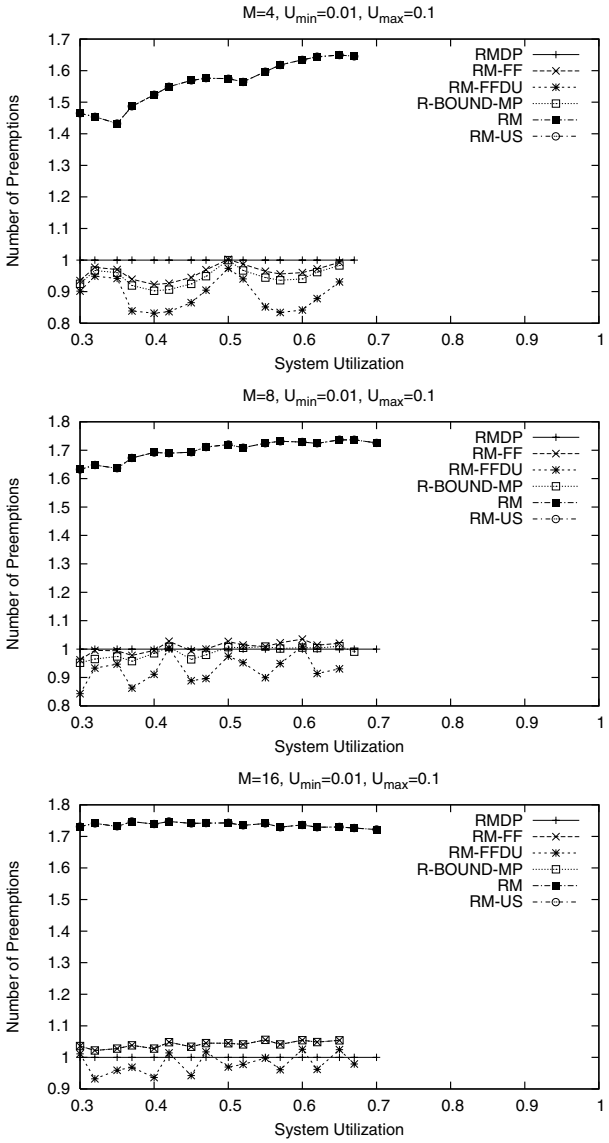

Figure 13. Task preemptions ($U_{max} = 0.1$)

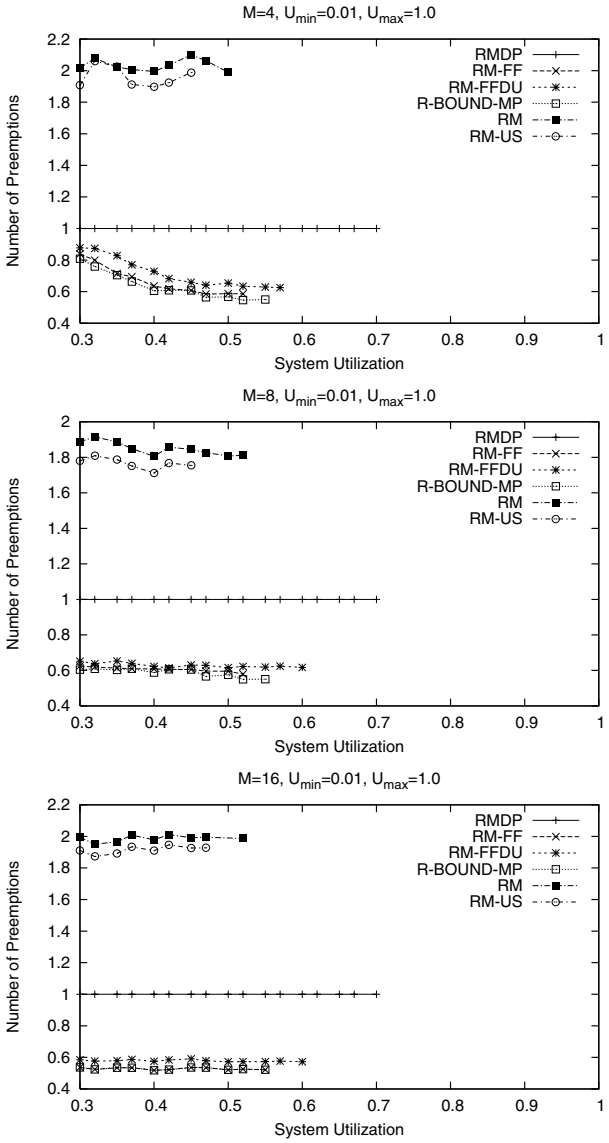

Figure 14. Task preemptions ($U_{max} = 1.0$)

Then, the number of task preemptions for each algorithm relative to that for the RMDP algorithm is calculated only only for the case in which both the target algorithm and the RMDP algorithm have a schedulable utilization of 100%. The WM algorithm consistently generate more than fifty times as many preemptions as the RMDP algorithm, hence it is not included in the results.

Figure 13 shows the numbers of task preemptions for each algorithm relative to that for the RMDP algorithm with respect to task sets in which the utilization of every individual task ranges from within $[0.01, 0.5]$. The RMDP algorithm causes slightly more preemptions than the RM-FF, RM-FFDU and R-BOUND-MP algorithms, but they are almost competitive. The additional preemptions occur in the RMDP algorithm for the case in which the first portions of split tasks are dispatched when their corresponding second portions are in execution. Since the utilizations of the tasks are relatively low in the discussed simulations, the first and the second portions of split tasks have less chance to be overlapped. As a result, the number of preemptions in the RMDP algorithm is not increased very much. The RM and RM-US algorithms generate about $1.5 \sim 1.7$ times as many preemptions as the RMDP algorithm. In those algorithms, one global scheduler manages all the tasks, and hence the priority order of the tasks are more likely to be changed compared to the RMDP and partitioned scheduling algorithms. As a result, the RM and RM-US algorithms grow the numbers of preemptions.

Figure 14 shows the numbers of task preemptions for each algorithm relative to that for the RMDP algorithm with respect to task sets in which the utilization of every individual task ranges from within $[0.01, 1.0]$. The presence of heavy tasks expands the performance difference between the RMDP algorithm and the partitioned scheduling algorithms. In the RMDP algorithm, preemptions occur every time the first portions of split tasks are dispatched while their corresponding second portions are in execution. Since the tasks are likely to have high utilizations, such preemp-

tions are likely to happen, and hence the numbers of preemptions in the RMDP algorithm are boosted up, compared to the partitioned scheduling algorithms in which such preemptions never occur. The resulting numbers of preemptions in the RMDP algorithm are 2.0 $\sim$ 2.5 times as many as those in the partitioned scheduling algorithms, though they are much fewer than the RM and RM-US algorithms.

The impact of the performance difference in the number of preemptions to the system depends on the ratio of the processor time consumed by the task executions and the processor time consumed by the task preemptions. For example, the RMDP algorithm achieves schedulable utilizations about 10% higher than the RM-FFDU algorithm, while it incurs about 1.8 times as many preemptions. Therefore, the RM-FFDU algorithm may be better than the RMDP algorithm, if the scheduler consumes more than about 11% of the overall system time for the task preemptions. The system designer should take this fact into account. As long as this paper simulated, the RMDP algorithm seems better than the RM-FFDU algorithm, because the scheduler hardly consumes 11% of the system time for the task preemptions, considering the specifications of the current processors.

7 Conclusion

This paper presented the RMDP algorithm that combines the portioned scheduling technique and the Rate Monotonic algorithm. The theoretical description gave that the RMDP algorithm is well implementable, since it incurs only a little implementation in addition to the partitioned RM algorithm. The schedulability analysis derived that the worst-case utilization bound of the RMDP algorithm is 50%. The simulation studies demonstrated that the RMDP algorithms successfully scheduled the task sets with higher system utilizations than the traditional RM-based algorithms, without generating many preemptions. Besides, it is guaranteed by the theory that the degree of task migrations for the RMDP algorithm is suppressed so that at most $M-1$ tasks occur migrations and each of them migrates between the restrictive two processors. In consequence, this paper believes that the RMDP algorithm can be a new choice for scheduling recurrent real-time tasks on multiprocessor platforms.

References

[1] J. Anderson and A. Srinivasan. Early-Release Fair Scheduling. In *Proc. of the Euromicro Conference on Real-Time Systems*, pages 35–43, 2000.

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

[3] 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.

[4] 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.

[5] T.P. Baker. An Analysis of EDF Schedulability on a Multiprocessor. *IEEE TRANSACTIONS ON PARALLEL AND DISTRIBUTED SYSTEMS*, 16:760–768, 2005.

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

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

[8] 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.

[9] 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.

[10] M. Cirinei and T.P. Baker. EDZL Scheduling Analysis. In *Proceedings of the Euromicro Conference on Real-Time Systems*, pages 9–18, 2007.

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

[12] 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.

[13] T. Kuo and A. Mok. Load Adjustment in Adaptive Real-Time Systems. In *Proceedings of the IEEE Real-Time Systems Symposium*, pages 160–171, 1991.

[14] 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.

[15] 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.

[16] 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.

[17] 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.

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

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

[20] K. Olukotun, B.A. Nayfe, L. Hammond, K. Wilson, and K. Chang. The Case for a Single-Chip Multiprocessor. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 2–11, 1996.

[21] S. Ramamurthy. Scheduling Periodic Hard Real-Time Tasks with Arbitrary Deadlines on Multiprocessors. In *Proceedings of the IEEE Real-Time Systems Symposium*, pages 59–68, 2002.

[22] N. Yamasaki. Responsive Multithreaded Processor for Distributed Real-Time Systems. *Journal of Robotics and Mechatronics*, 17(2):130–141, 2005.