Portioned EDF-based Scheduling on Multiprocessors [*]

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

ABSTRACT

This paper presents an EDF-based algorithm, called Earliest Deadline Deferrable Portion (EDDP), for efficient scheduling of recurrent real-time tasks on multiprocessor systems. The design of EDDP is based on the portioned scheduling technique which classifies each task into a fixed task or a migratable task. A fixed task is scheduled on the dedicated processor without migrations. A migratable task is meanwhile permitted to migrate between the particular two processors. In order to curb the cost of task migrations, EDDP makes at most $M-1$ migratable tasks on M processors. The scheduling analysis derives the condition for a given task set to be schedulable. It is also proven that no tasks ever miss deadlines, if the system utilization does not exceed 65%. Beyond the theoretical analysis, the effectiveness of EDDP is evaluated through simulation studies. Simulation results show that EDDP achieves high system utilization with a small number of preemptions, compared with the traditional EDF-based algorithms.

Categories and Subject Descriptors

D.4.1 [**Operating Systems**]: Process Management—*Scheduling, Multiprocessing/multiprogramming/multitasking*; D.4.7 [**Operating Systems**]: Organization and Design—*Real-time systems and embedded systems*

General Terms

Algorithms, Theory

Keywords

Scheduling Algorithm, Real-Time Systems, Multiprocessor Systems

[*]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.

1. INTRODUCTION

Ever since the Earliest Deadline First (EDF) algorithm [14] turned out to be no longer optimal on multiprocessors [10], the real-time computing community have developed alternative scheduling algorithms. The primary focus is at improving the worst-case system utilization with guaranteeing all tasks to meet deadlines. Unfortunately, most of the developed algorithms suffer from trade-off between theoretical schedulability and practical overhead: achieving high system utilization leads to complex computations. Solutions are still widely discussed.

For the scheduling of recurrent real-time tasks on multiprocessors, there have been two approaches: 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. In partitioned scheduling, on the other hand, each task is assigned to a single processor, on which each of its jobs will be executed, and processors are scheduled independently. Thus, a task is executed on a dedicated processor and never migrates among processors. The main advantage of partitioned scheduling is that they reduce a multiprocessor scheduling problem to a set of uniprocessor ones.

Dhall *et al.* showed that global EDF may cause a deadline to be missed if the total utilization of a task set is slightly greater than 1 [10]. Letting M be the number of processors, the worst-case system utilization is therefore no greater than $1/M$. The partitioned scheduling approaches are often more efficient for multiprocessors. Lopez *et al.* showed that EDF with the FF (or BF) heuristic can successfully schedule any task set with a total utilization at most $(\beta M+1)/(\beta+1)$, where $\beta = \lfloor 1/\alpha \rfloor$ and α is the maximum utilization of every individual task [15]. Letting $\alpha = 1$ and $\beta = 1$, the total utilization becomes $(M+1)/2$. That is, the worst-case system utilization becomes 50% for $M \rightarrow \infty$.

This paper presents an EDF-based algorithm for efficient scheduling of recurrent real-time tasks on multiprocessor systems. The design of the presented algorithm is based on the portioned scheduling technique [12, 13] which classifies each task into a fixed task or a migratable task. The main advantage of the presented algorithm is that the achievable worst-case system utilization is 65%, while the previous portioned scheduling algorithms, and most of scheduling algorithms, can never achieve a worst-case system utilization over 50%. Thus, we believe that the algorithm can be a new alternative for the scheduling of recurrent real-time tasks on multiprocessor systems.

The rest of this paper is organized as follows. In the next section, the system model and the terminology are defined. Section 4 presents the scheduling algorithm and analyzes the schedulability. Section 5 evaluates the effectiveness of the algorithm, compared to the traditional algorithms. This paper is concluded in Section 6.

2. RELATED WORK

Baruah *et al.* developed the Pfair scheduling technique that achieves the optimal scheduling of periodic real-time tasks on multiprocessors [5, 4]. In Pfair scheduling, each task is divided into quantum-size pieces so-called subtasks which have pseudo deadlines. A series of the Pfair algorithms is then designed so that all subtasks meet their pseudo deadlines. The PD^2 algorithm [1] is known to be the most efficient in the optimal Pfair algorithms. However, the scheduling overhead is often criticized, since they necessarily generate many preemptions due to quantum-based scheduling. The practical implementation of Pfair scheduling has been therefore considered recently [18, 6].

The LLREF algorithm [7] is another optimal multiprocessor scheduling algorithm. Unlike the Pfair algorithms, LLREF does not rely on the quantum-based scheduling model but on the original T-L plane abstraction. In T-L plane abstraction, the scheduling decision is made locally within every interval of job releases to restrain and bound the number of preemptions. The algorithm has been extended so that it moreover reduces the number of preemptions [11], but it still generates many preemptions compared to non-optimal scheduling algorithms.

The EKG algorithm [2] considers trade-off between system utilization and the number of preemptions. EKG classifies each task into a fixed task or a migratable task. A fixed task is scheduled on the dedicated processor. A migratable task is meanwhile permitted to migrate between the particular two processors. The main advantage of this approach is that the number of migratable tasks is limited to $M-1$ on M processors, while the migrations of $M-1$ tasks improve system utilization. Thus, the migration overhead is relaxed compared to the other optimal algorithms. The algorithm trades an achievable system utilization with the cost of preemptions by adjusting the parameter k, where $2 \leq k \leq M$. For $k < M$, the achievable utilization is $k/(k+1)$. For $k = M$, on the other hand, it is 100%, thereby EKG performs optimally.

In general, optimal scheduling algorithms lead to many preemptions. There are also efficient algorithms that aim for practical use with a small number of preemptions. The EDF-US[x] algorithm [17] simply assigns the highest priority to the tasks with utilizations greater than x. The rest of the tasks are then scheduled by EDF. Baker showed that $x = 1/2$ is the best configuration for this algorithm [3], and any task set can be successfully scheduled by EDF-US[1/2], if the total utilization does not exceed $(M+1)/2$ on M processors.

The EDZL algorithm [8] dynamically assigns the highest priority to the tasks which reach zero laxity. EDZL is more attractive than EDF-US[x] in that it is at least as effective as EDF, while EDF-US[x] is not. Piao *et al.* showed that any task sets can be successfully scheduled by EDZL if the total utilization does not exceed $(M+1)/2$ on M processors [16]. In recent years, more strict analysis of EDZL has been presented [9, 19].

The Ehd2-SIP algorithm [12] takes a similar approach to EKG in such a way that each task is classified into a fixed task or a migratable task, but the approach is simplified for practical use. While EKG utilizes the full capacity of every processor on which a migratable task is executed, Ehd2-SIP does not fully utilize the processor to reduce the computation complexity. Thus, the scheduling of migratable tasks is more straightforward than EKG. Such a scheduling scheme was defined *portioned scheduling* in [12]. Although the achievable total utilization was proven to be no greater than $(M+1)/2$, the simulation results showed that Ehd2-SIP is competitive with EKG setting $k = 2$ in terms of achievable system utilization.

From the viewpoint of balance between schedulability and complexity, Ehd2-SIP and EKG with a small parameter k are attractive. This paper integrates the notions of Ehd2-SIP and EKG. The developed algorithm succeeds the design simplicity of Ehd2-SIP for practical use, and at the same time partially imitates the approach of EKG to improve the system utilization bound. The resulting advantage is that the implementation cost is not far beyond the traditional partitioned scheduling algorithms while the worst-case system utilization is no less than 65%.

3. SYSTEM MODEL

The system is composed of M processors: $P_1, P_2, ..., P_M$. A periodic task set $\Gamma = \{\tau_1, \tau_2, ..., \tau_N\}$ is given to the system. Each task τ_i is defined by tupple $(C_i,\ T_i)$ where C_i is its worst-case execution time and T_i is its period. Note that $C_i \leq T_i$ is always satisfied. The processor utilization of τ_i is defined by $U_i = C_i/T_i$. Every task generates a sequence of jobs periodically. The kth job of τ_i is denoted by $\tau_{i,k}$ that is released at time $r_{i,k}$ and its deadline is equal to the release time of the next job, i.e. $d_{i,k} = r_{i,k+1} = r_{i,k} + T_i$. The start time and the finish time of $\tau_{i,k}$ are denoted by $s_{i,k}$ and $f_{i,k}$ respectively. All the tasks are preemptive and independent. Therefore, no tasks make critical sections and synchronize with other tasks. Any jobs 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.

This research has the other assumptions as follows. The system is a memory-shared multiprocessor. Each processor shares the code and data. The costs of preempting and migrating tasks are not considered, since those costs depend on the processor performance. From the viewpoint of scheduling algorithms, it is more variant to focus on how often context switches occur, in order to discuss the runtime overhead of the system. Hence this paper takes the number of context switches as a performance metric of the scheduling overhead. This research also ignores the behavior of a cache system, because it is dominated by the processor specification and architecture. In fact, a cache system affects the analysis of the worst-case execution time rather than scheduling. Such an analysis is not in the range of this paper.

4. THE EDDP ALGORITHM

This section presents the Earliest Deadline Deferrable Portion (EDDP) algorithm. The design of EDDP is based on the portioned scheduling technique which is composed of the task assigning phase and the task scheduling phase. This

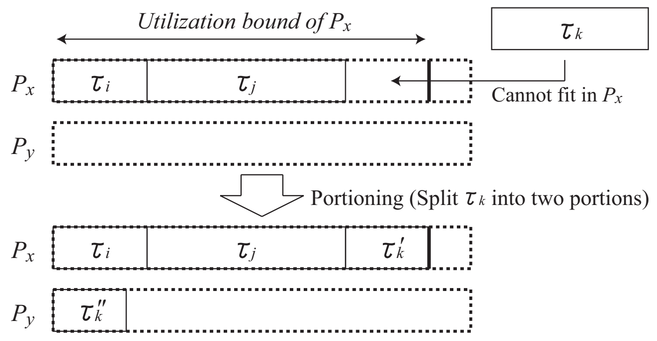

Figure 1: Portioning example

section begins with a review of the portioned scheduling technique. Then, the task assigning algorithm and the task scheduling algorithm are presented. Finally, the scheduling analysis of EDDP is given.

4.1 Portioned Scheduling Technique

The portioned scheduling technique [12] is composed of the two phases: task assigning phase and task scheduling phase. In the task assigning phase, each task is assigned to a particular processor, as long as the task does not cause the total utilization of the processor to exceed the analyzed theoretical bound. Such a task is classified into a fixed task. If the total utilization exceeds the bound, the utilization of the task is split into two portions. Such a task is classified into a migratable task. One is assigned to the processor to which the tasks are being assigned and the other is assigned to the next processor to which the rest of the tasks will be assigned.

Notice that the body (program code) of the task is not really divided into two blocks, but its utilization is shared on the two processors. In other words, the processor capacity that the split task will use for execution is reserved on the two processors. From the scheduling point of view, each split portion is deemed as a pseudo task to serve the execution of the original task on each processor.

Figure 1 shows an example of task splitting, called *portioning* in this paper. The width of the box in which the task name is indicated is the processor utilization of the task. This example presumes that a task τ_k causes the total utilization of P_x to exceed its utilization bound, if it is assigned to P_x. Thus, τ_k is split into 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_y. Hence, the portioning approach obviously improves the total utilization of P_x.

The execution times of τ'_k and τ''_k are denoted by C'_k and C''_k respectively. Letting U^*_x be the utilization bound of P_x, C'_k and C''_k are calculated as: $C'_k = T_k(U^*_x - U_i - U_j)$ and $C''_k = C_k - C'_k$. This means that τ_k consumes C'_k time units on P_x and C''_k time units on P_y within every T_k. In fact, τ'_k and τ''_k are scheduled as individual periodic tasks, but they must be scheduled exclusively on P_x and P_y, since a task τ_k cannot be executed in parallel.

4.2 Task Assigning Phase

This section describes how tasks are assigned and split to processors in EDDP. Let U^* denote the utilization bound for EDDP, which is analyzed in Section 4.4. Then, τ_i is defined to be a *heavy task*, if Condition (1) holds. Otherwise, it is defined to be a *light task*.

$$U_i > U^* \tag{1}$$

The EDDP assigning algorithm has two steps: the first step assigns the heavy tasks to the dedicated processors and the second step assigns the light tasks to the rest of the processors. Figure 2 shows the pseudo code of the task assigning algorithm. The algorithm assumes that the given task set includes h heavy tasks and $N - h$ light tasks, where the heavy tasks are indexed $1 \sim h$ and the light tasks are indexed $h+1 \sim N$. The set of the light tasks is sorted so that the period of τ_i is smaller than or equal to that of τ_{i+1}. Basically, the algorithm assigns the tasks to the processors sequentially, which means that it always assigns τ_{i+1} after τ_i. If a task causes the total utilization of a processor to exceed its utilization bound, it splits the task into two portions. Then it continues to assign the rest of the tasks from the next processor. A set of the tasks assigned to a processor P_x is denoted by Π_x.

The procedure of the algorithm is as follows. If the number of the heavy tasks is smaller than the number of the processors, the heavy tasks are first assigned to dedicated processors $P_1 \sim P_h$ (line 2∼5). Note that each processor P_x $(1 \leq x \leq h)$ has only one heavy task. Then, the light tasks are assigned to the rest of the processors. Each light task τ_i is assigned to a processor P_x as long as the total utilization of P_x is less than or equal to its utilization bound denoted by U^*_x (line 8∼9). If the total utilization of P_x exceeds the utilization bound, τ_i is going to be split into two portions as follows. At first, the execution times of the first portion and the second portion of the task being split are calculated based on the remaining schedulable utilization of P_x, denoted by $U^*_x - U(\Pi)$ (line 11). Then, τ_i is split into $\tau'_i(C'_i, T_i)$ and $\tau''_i(C''_i, T_i)$ based on the remaining schedulable utilization (line 12). τ'_i is assigned to P_x and τ''_i is assigned to P_{x+1} (line 13). Finally, the utilization bound of the next processor is calculated using the formula that is described in Section 4.4 (line 16). If all the tasks can be assigned to the processors, the algorithm succeeds.

Although the task assigning algorithm of EDDP is similar to that of Ehd2-SIP, it differs in that the tasks are defined to be heavy or light, and the heavy tasks are assigned to dedicated processors. The separation of heavy tasks and light tasks bring two advantages over Ehd2-SIP. First, it improves the worst-case system utilization, as described in Section 4.4. Second, it makes the scheduling easier, because the processors on which only one heavy task is executed need no schedulers. In this point, EDDP partially imitates EKG. However, the scheduling algorithm of EDDP is quite different from EKG, as described in the next section.

4.3 Task Scheduling Phase

This section describes the task scheduling algorithm of EDDP. Notice that the processors executing heavy tasks need no schedulers, since each of them contains only one heavy task. On the other hand, the processors executing light tasks need schedulers. Like the traditional partitioned scheduling algorithms, the schedulers have the same scheduling policy, but each of them is not completely independent, because two processors next to each other may share a migratable task.

Assumption:
i indicates the index of the tasks.
x indicates the index of the processors.
h denotes the number of the heavy tasks.
U^*_x indicates the utilization bound of P_x.
Task set Γ is sorted so that $\{\tau_i \mid 1 \le i \le h\}$ are heavy tasks and $\{\tau_j \mid h+1 \le j \le N\}$ are light tasks.
The light tasks are also sorted so that $T_j \le T_{j+1}$.

```
1.  $\forall \Pi_x = \emptyset$ ;
2.  if $h > M$
3.     return FAILURE;
4.  for $1 \le i \le h$
5.     $\Pi_i = \{\tau_i\}$ ;
6.  $x = h + 1$ and $U^*_x = 1.0$ ;
7.  for $h + 1 \le i \le n$
8.     if $U(\Pi_x) + U_i \le U^*_x$
9.        $\Pi_x = \Pi_x \cup \tau_i$ ;
10.    else if $x < M$
11.       $C'_i = \{U^*_x - U(\Pi_x)\}T_i$ and $C''_i = C_i - C'_i$ ;
12.       split $\tau_i$ into $\tau'_i(C'_i, T_i)$ and $\tau''_i(C''_i, T_i)$ ;
13.       $\Pi_x = \Pi_x \cup \tau'_i$ and $\Pi_{x+1} = \{\tau''_i\}$ ;
14.       $x = x + 1$ ;
15.       if $i + 1 \le N$
16.          $U^*_x = 1.0 - \frac{C''_i(T_i + \min\{C'_i, C''_i\} - C''_i)}{T_i T_{i+1}}$ ;
17.    else
18.       return FAILURE;
19. return SUCCESS;
```

Figure 2: EDDP assigning algorithm

Assume that τ''_i, τ'_k, and $\{\tau_j \mid i < j < k\}$ $(i < k)$ are assigned to P_x. More specifically, τ_i is split into P_{x-1} and P_x, and its second portion τ''_i is assigned to P_x. In addition, τ_k is also split into P_x and P_{x+1}, and its first portion τ'_k is assigned to P_x. Then, the task set is scheduled as follows.

1. If the task with the earliest deadline is τ''_i but τ'_i is currently executed on P_{x-1}, then the task with the second earliest deadline is dispatched, since τ'_i and τ''_i, which form a same job, cannot be executed in parallel.
2. Otherwise the earliest deadline task is dispatched.

In other words, the tasks are scheduled by EDF except that τ''_i has the earliest deadline but its corresponding first portion τ'_i also has the earliest deadline and is in execution on the neighbor processor P_{x-1}. In this exceptional case, the scheduler defers execution of τ''_i until τ'_i completes so that τ'_i and τ''_i are executed exclusively.

Figure 3 depicts an example of EDDP scheduling in which τ_i is a migratable task between P_{x-1} and P_x. Suppose that τ_i is released at time r_i with deadline d_i that is the earliest on P_x but is not on P_{x-1}, and there are two active tasks, τ_j and τ_k, on P_x with deadlines of d_j and d_k respectively, both of which are later than d_i. In such a case, τ''_i starts execution on P_x at time r_i. Assume that τ'_i receives the earliest deadline on P_{x-1} at time t_1. Then, τ'_i is dispatched on P_{x-1} and τ''_i is preempted even if it has the earliest deadline on P_x. Instead, τ_j with the second earliest deadline is dispatched on P_x. Hence, τ''_i is deferred, though it has the earliest deadline. Assume that τ'_i is preempted by a task released with an earlier deadline at time t_2 and is later resumed at time t_3. Then, τ''_i can be executed during a time

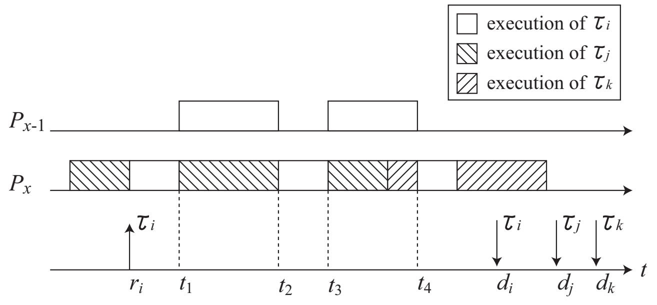

Figure 3: EDDP scheduling example

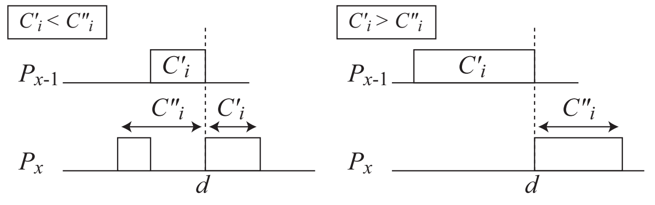

Figure 4: The latest completion for EDDP

interval $[t_2, t_3)$. When τ'_i completes at time t_4, meaning that τ_i consumes C'_i time units, τ''_i is resumed.

The above scheduling policy has a potential problem with respect to deadline assignments. Consider a job of τ''_i with a deadline d. If a set of the tasks assigned to P_x, whose total utilization is less than or equal to 1, is scheduled by EDF without taking into account exclusive scheduling of τ'_i and τ''_i, the job of τ''_i never misses the deadline d, though the job may begin at time $d - C''_i$ and complete at time d in the latest case. However, applying the above scheduling policy for the exclusive scheduling of τ'_i and τ''_i, the completion time of the job may be delayed until time $d + \min\{C'_i, C''_i\}$, if the job of τ'_i is scheduled during $[d - C'_i, d)$ on P_{x-1}.

Figure 4 depicts such a problematic scheduling. Due to the execution of the corresponding job of τ'_i, the job of τ''_i may be blocked for at most C'_i time units if $C'_i < C''_i$. On the other hand, if $C'_i \ge C''_i$, the job of τ''_i may be blocked for at most C''_i time units.

Taking into account the blocking time of τ''_i, EDDP assigns job deadlines as follows.

- For a job of τ''_i with a deadline d, its deadline is virtually transformed to $d - \min\{C'_i, C''_i\}$.
- For a job of $\{\tau_j \mid i < j < k\}$ and τ'_k with a deadline d, its deadline is remained d.

In other words, τ''_i is deemed to have a relative deadline $T_i - \min\{C'_i, C''_i\}$. Once a job of τ''_i has the earliest deadline on P_x, no other jobs have earlier deadlines until the job of τ''_i completes, since the period (relative deadline) of τ''_i is guaranteed to be the smallest in Π_x by the characteristic of the task assigning algorithm presented in the previous section. Therefore, letting $\bar{d} = d - \min\{C'_i, C''_i\}$, if a job of τ''_i is guaranteed to complete by time $\bar{d}$ in EDF scheduling, it is also guaranteed to complete by time $\bar{d} + \min\{C'_i, C''_i\} = d$ in EDDP scheduling. Since the other jobs are never blocked, they are guaranteed to complete by their deadlines in EDDP

Assumption:
τ_i is a migratable task between P_{x-1} and P_x.
τ_k is a migratable 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 $\tau_i$ is released
 4.       $e_i = C''_i$ and $t_i = t$.
 5.     if $\tau_k$ 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 $\tau_i$ is currently running on $P_x$
12.     $e_i = e_i - (t - t_i)$.
13.   else if $\tau_k$ is currently running on $P_x$
14.     $e_k = e_k - (t - t_k)$.
15.   /* Selection of the next task. */
16.   if $\tau_i$ has earliest deadline
17.     if $\tau_i$ is in execution on $P_{x-1}$
18.       execute $\tau_j$ with second earliest deadline.
19.     else
20.       execute $\tau_i$.
21.       let second_end invoke at $t + e_i$.
22.       $t_i = t$.
23.   else if $\tau_k$ has earliest deadline
24.     if $\tau_k$ is in execution on $P_{x+1}$
25.       invoke schedule_$P_{x+1}$ on $P_{x+1}$.
26.     execute $\tau_k$.
27.     let first_end invoke at $t + e_k$.
28.     $t_k = t$.
```

Figure 5: EDDP scheduling algorithm

scheduling, if they are guaranteed to complete by their deadlines in EDF scheduling. The condition for EDDP to guarantee all tasks to meet deadlines is presented in detail in the next section.

Figure 5 shows the pseudo code of the EDDP scheduler. Every time any tasks are released on P_x, a scheduling function *schedule_P_x* is invoked (line 1$\sim$7). Let t be such a time. At this time, if the migratable tasks τ_i and τ_k are released, the scheduler needs to reset the variables t_i, t_k, e_i and e_k to track their remaining execution times. Since τ_i and τ_k are not allowed to consume more than C''_i and C'_k time units respectively on P_x, they must be preempted by using timers.when they exhaust C''_i and C'_k time units. Thus, the information of the remaining execution times is required by the scheduler.

The scheduling function proceeds as follows. First of all, if the currently-running task is a migratable task, the scheduler saves its remaining execution time (line 11$\sim$13). Then, it selects a task to execute (after line 15). If τ_i has the earliest deadline (line 16), the scheduler examines whether τ_i is currently in execution on P_{x-1} (line 17), since τ''_i must wait

```
 1. function job_end
 2.   remove the caller task from a ready set.
 3.   call schedule_$P_x$.

 4. function first_end
 5.   remove $\tau_r$ from a ready set.
 6.   call schedule_$P_x$.
 7.   if $\tau_k$ is ready on $P_{x+1}$
 8.     invoke schedule_$P_{x+1}$ on $P_{x+1}$.

 9. function second_end
10.   remove $\tau_i$ from a ready set.
11.   call schedule_$P_x$.
```

Figure 6: The job finishing functions

for τ'_i. If τ_i is in execution on P_{x-1}, the scheduler executes a task with the second earliest deadline (line 18). Otherwise, τ_i can be executed (line 20). Because τ_i cannot overrun C''_i time units on P_x, the scheduler sets a timer so that it will invoke a function *second_end* at time $t + e_i$ to preempt τ_i on P_x (line 21). Notice that the *second_end* function may not be invoked at time $t + e_i$, since the time is updated if τ_i is preempted once and later dispatched again within the same period. So, it is just a time at which τ_i may exhaust C''_i time units. The scheduler must save the current time as the last dispatched time of τ_i (line 22) to track its remaining execution time. Meanwhile, if τ_k has the earliest deadline (line 23), the scheduler examines whether τ_k is in execution on P_{x+1}. If τ_k is in execution on P_{x+1}, the scheduler invokes *schedule_P_{x+1}* to reschedule P_{x+1}, since τ''_k must wait for τ'_k. Then, it executes τ_k (line 26). A timer is also set so that a function *first_end* will be invoked at time $t + e_k$ to preempt τ_k on P_x (line 27). Finally, it saves the current time as the last dispatched time of τ_k (line 28).

Since the scheduler needs to invoke a scheduling function on the neighbor processor, the processors must support software interruptions. The scheduler can easily know if migratable tasks are currently running on the neighbor processor, because the processors share code and data.

The job finishing functions are shown in Figure 6. Tasks except for the migratable ones call the *job_end* function, when jobs of them complete (line 1$\sim$3). The *first_end* and *second_end* functions are called only through the timers. Those functions call the scheduling function to reschedule the tasks (line 3, 6 and 11). Only when τ_k consumes C'_k on P_x but has not consumed C''_k on P_{x+1} yet, the scheduler invokes the scheduler operating on P_{x+1} to dispatch and execute τ_k on P_{x+1}, since τ_k has the earliest deadline on P_{x+1} but has been deferred.

4.4 Schedulability Analysis

This section derives the schedulable condition for EDDP. No deadline misses occur on a processor containing a heavy task, since 100% of processor time is allocated to every heavy task. Hence, the analysis focuses on the schedulable conditions of processors where the light tasks are assigned. Like the previous section, let $\Pi_x = \{\tau''_i, \{\tau_j \mid i < j < k\}, \tau'_k\}$ $(i < k)$ be a set of the light tasks assigned to a processor P_x. The goal of this section is to derive the schedulable condition and the utilization bound of P_x.

Suppose that a job of some fixed task, i.e. $\tau_s \in \Pi_x \setminus \tau_i''$, misses its deadline at time d. Let t $(< d)$ be the latest time instant at which the processor is either idle or is executing a job whose deadline is after d. The total amount of processor time consumed by τ_i'' in the interval of $(t, d]$ is at most equal to W'' expressed by Equation (2).

$$W'' = C_i'' + \left\lfloor \frac{d + \min\{C_i', C_i''\} - t - C_i''}{T_i} \right\rfloor C_i'' \qquad (2)$$

Figure 7 shows the case in which τ_i'' consumes this amount of time. At time t, a deferred job of τ_i'' starts with the laxity of zero and finishes at $t + C_i''$, which is its deadline, without being preempted. Since d must be after or at the pseudo deadline of the last job of τ_i'' before time d in Figure 7, τ_i'' consumes at most $\lfloor\{(d + C_i') - (t + C_i'')\}/T_i\rfloor C_i''$ time units after $t + C_i''$. Hence, the maximal amount of processor time consumed by τ_i'' in the interval of $(t, d]$ is calculated by Equation (2). Since $\lfloor a \rfloor \le a$ for any a, W'' must satisfy the following condition.

$$\begin{aligned} W'' &\le C_i'' + \frac{d + \min\{C_i', C_i''\} - t - C_i''}{T_i} C_i'' \\ &= U_i''(d - t + T_i + \min\{C_i', C_i''\} - C_i'') \end{aligned}$$

Here, the total amount of processor time consumed by each fixed task, i.e. $\tau_j \in \Pi_x \setminus \tau_i''$, in the interval of $(t, d]$ is at most $C_j(d-t)/T_j$, since the tasks except for τ_i'' are scheduled by EDF. In order to cause the job of τ_s to miss its deadline at time d, the total amount of processor time consumed by all the jobs with deadlines before or at d needs to exceed the interval of $(t, d]$. Hence, the following inequality must be satisfied. Let $L = d - t$ due to the limitation of space.

$$L < \sum_{\tau_j \in \Pi_x \setminus \tau_i''} \frac{C_j}{T_j} L + U_i''(L + T_i + \min\{C_i', C_i''\} - C_i'')$$

If the above inequality is not satisfied, the job never misses its deadline. Dividing both sides of the above inequality and considering $T_s \le d - t$, the schedulable condition for τ_s is derived by Equation (3).

$$\sum_{\tau_j \in \Pi_x \setminus \tau_i''} U_j + U_i'' \left(1 + \frac{T_i + \min\{C_i', C_i''\} - C_i''}{T_s}\right) \le 1 \qquad (3)$$

Equation (3) implies that all the tasks must satisfy this condition if the task with the shortest period satisfies this condition. Since the light tasks are sorted in the order of increasing period, T_{i+1} is the shortest in $\{T_j \mid \tau_j \in Pi_x \setminus \tau_i''\}$. Therefore, Equation (4) is the schedulable condition for all the tasks assigned to P_x.

$$\sum_{\tau_j \in \Pi_x \setminus \tau_i''} U_j + U_i'' \left(1 + \frac{T_i + \min\{C_i', C_i''\} - C_i''}{T_{i+1}}\right) \le 1$$

$$\Leftrightarrow \sum_{\tau_j \in \Pi_x \setminus \tau_i''} U_j + U_i'' \le 1 - \frac{C_i''(T_i + \min\{C_i', C_i''\} - C_i'')}{T_i T_{i+1}} \qquad (4)$$

Notice that the left-hand side of Equation (4) is equal to the total utilization of P_x. Hence, the right-hand side of Equation (4) is the utilization bound of P_x. The task assigning algorithm presented in Section 4.2 uses the right-hand side of Equation (4) as a formula of obtaining the schedulable utilization bound at line 17 in Figure 2.

Now the analysis derives the worst-case utilization bound by minimizing the right-hand side of Equation (4). Let $\bar{U}$ denote the right-hand side of Equation (4). Since $T_i \le T_{i+1}$, $\bar{U}$ is minimized when T_{i+1} is reduced to T_i.

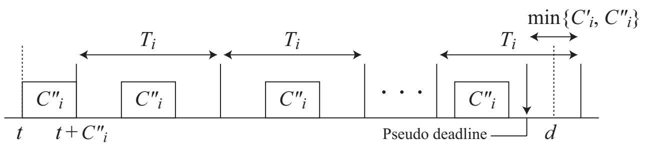

Figure 7: τ_i'' consumes the most time

$$\bar{U} \ge 1 - \frac{C_i''(T_i + \min\{C_i', C_i''\} - C_i'')}{T_i^2}$$

The analysis first considers the case of $C_i' \ge C_i''$. In this case, the relations of $U_i' \ge U_i''$ and $U_i = U_i' + U_i''$ derive $U_i'' \le U_i/2$. Hence, $\bar{U}$ is minimized as follows.

$$\begin{aligned} \bar{U} &\ge 1 - \frac{C_i''(T_i + C_i'' - C_i'')}{T_i^2} \\ &\ge 1 - \frac{U_i}{2} \end{aligned}$$

Remember that the utilization of a light task is at most U^* according to Equation (1), which means that $\bar{U}$ is minimized when $U_i = U^*$. Hence, the worst-case utilization bound for the case of $C_i' \ge C_i''$ is $U^* = 1 - U^*/2$, that is, $U^* = 2/3 \simeq 66\%$. Next, the analysis considers the case of $C_i' \le C_i''$. In this case, $\bar{U}$ is minimized as follows with taking $U_i' = U_i - U_i''$ into account.

$$\begin{aligned} \bar{U} &\ge 1 - \frac{C_i''(T_i + C_i' - C_i'')}{T_i^2} \\ &= 1 - U_i''(1 + U_i' - U_i'') \\ &= 1 - U_i''(1 + U_i - 2U_i'') \\ &= 2{U_i''}^2 - (1 + U_i)U_i'' + 1 \\ &= 2\left(U_i'' - \frac{1 + U_i}{4}\right)^2 + 1 - \frac{(1 + U_i)^2}{8} \\ &\ge 1 - \frac{(1 + U_i)^2}{8} \end{aligned}$$

By the same token as the case of $C_i' \ge C_i''$, $\bar{U}$ is minimized when $U_i = U^*$. Hence, the worst-case utilization bound for the case of $C_i' \le C_i''$ is $U^* = 1 - (1 + U^*)^2/8$, that is, $U^* = 4\sqrt{2} - 5 \simeq 65\%$. This is the absolute worst-case utilization bound of P_x for EDDP. This utilization bound is valid for any processor containing light tasks. In addition, the utilization of any processor containing a heavy task is assigned is equal to the utilization of the heavy task, which is guaranteed to be higher than $U^* = 4\sqrt{2} - 5$. In consequence, the utilization bound of the whole system for EDDP is also $4\sqrt{2} - 5 \simeq 65\%$.

5. SIMULATION

Beyond the theoretical analysis, this section simulates the following algorithms: EDDP, Ehd2-SIP, EDF-FF, EDF-BF, EDZL, and EKG. Ehd2-SIP is renamed by EDDHP in the simulations, which stands for Earliest Deadline Deferrable Highest-priority Portion. EKG offers two versions. EKG-2 takes a parameter $k = 2$ with a lower utilization bound of 66% and fewer preemptions. EKG-M takes a parameter $k = M$ with the optimal utilization bound of 100% in exchange for more preemptions. EDF and EDF-US[x] were not included in the simulations, since they are dominated by EDZL [9] The Pfair algorithms and LLREF were also not included in the simulations, since they are known to generate more preemptions than EKG-M [11].

5.1 Experimental Setup

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.

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

If an algorithm has high success ratio, it is estimated to offer high system utilization with a guarantee of timing constraints. The definition of a successfully-scheduled task set depends on an algorithm. For EDDP, EDDHP, EDF-FF, EDF-BF, EKG-2, and EKG-M, a task set is said to be successfully scheduled if all the tasks can be assigned to the processors, since they are designed so that no tasks will miss the deadline once they are successfully assigned to the processors. For EDZL, on the other hand, a task set is said to be successfully scheduled if the schedulability test, presented in [9], accepts the task set.

Simulations estimate the number of preemptions for an algorithm by calculating its average number, defined by the following expression.

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

Simulations are characterized by M, U_{max}, U_{min} and U_{tot}. M is the number of 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_{tot} is the total processor utilization of the tasks. Then, the system utilization is defined by $U_{sys} = U_{tot}/M$, ranging from 0% to 100%. The system utilization is determined within the range of $[30\%, 100\%]$, since most algorithms never cause deadline misses if the system utilization is below 30%. Due to the limitation of space, simulations attempt only the following combinations of the parameters. M has three cases: 4, 8, and 16. A set of (U_{min}, U_{max}) has then two cases: $(0.01, 0.5)$ and $(0.01, 1.0)$.

A task set Γ is generated as follows. A new periodic task is appended to Γ as long as $U(\Gamma) \leq U_{tot}$ is satisfied. For each task τ_i, its utilization U_i is computed based on a uniform distribution within the range of $[U_{min}, U_{max}]$. Only the utilization of the task generated at the very end is adjusted so that $U(\Gamma)$ becomes equal to U_{tot}. T_i is determined within the range of $[100, 3000]$ randomly, since the feedback periods of control and multimedia tasks in embedded systems are often set about $1 \sim 30$ms. Then, the execution time of the task $C_i = U_i T_i$ is calculated.

5.2 The Success Ratio

Figure 8, Figure 9, and Figure 10 depict the success ratio for each algorithm with respect to task sets in which the utilization of every individual task ranges within $[0.01, 0.5]$. Hereinafter, the term *schedulable utilization* refers to the maximum system utilization where the success ratio is maintained 100%.

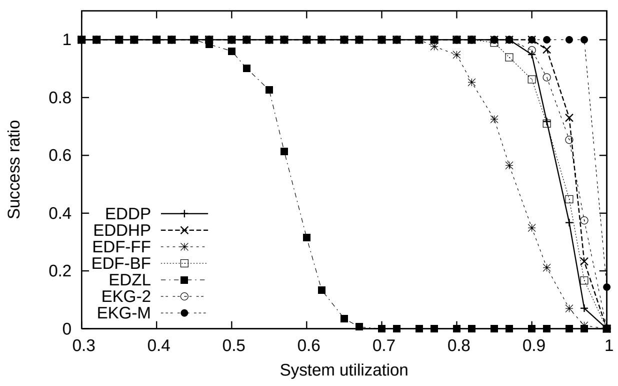

Figure 8: Success ratio: $U_{max} = 0.5$, $M = 4$

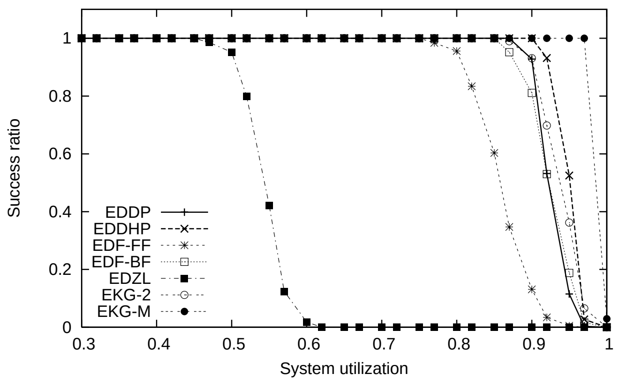

Figure 9: Success ratio: $U_{max} = 0.5$, $M = 8$

The results indicate that EDDP, EDDHP, EDF-BF, and EKG-2 are competitive. EDDHP slightly outperforms the rest of the algorithms: EDDHP achieves the schedulable utilization of 90%, while the rest of the algorithms achieves that of $85 \sim 87\%$. It is remarkable that EDDHP performs better than EDDP, though the worst-case utilization bound for EDDP is 65% that is higher than 50% achievable by EDDHP. This fact implies that the approach of assigning the highest priority to the second portion of a migratable task, which is conducted in EDDHP, may provide better schedulability than the approach of assigning the virtual deadline, which is conducted in EDDP.

Note that the success ratio for EKG-M drops below 100% before the system utilization reaches 100%, though EKG is supposed to become optimal for $k = M$. Since the execution time of a task cannot be split less than the minimum time unit of 1 in the simulations, each processor inevitably remains a little room unless its remaining utilization in splitting τ_i is an integer multiple of $1/T_i$. As a result, a few tasks may fail to be assigned to any processor, if the system utilization is very close to 100%.

EDF-FF performs relatively well, but its schedulable utilization is around 77% that is 8% behind EDF-BF. Since the BF heuristic packs items more efficiently than the FF heuristic, the performance of EDF-BF is also better than EDF-FF. EDZL has lower schedulable utilizations, which are around 50%. Since the schedulability analysis of EDZL was claimed

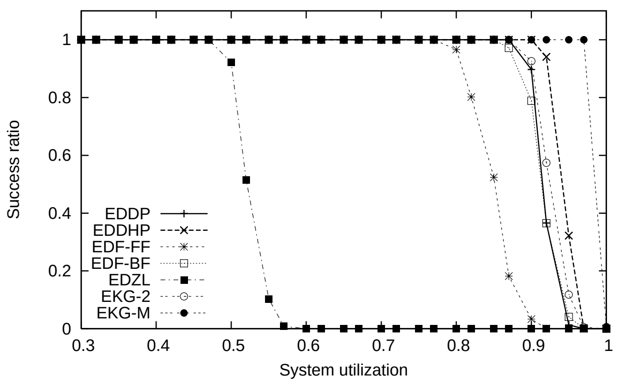

Figure 10: Success ratio: $U_{max} = 0.5$, $M = 16$

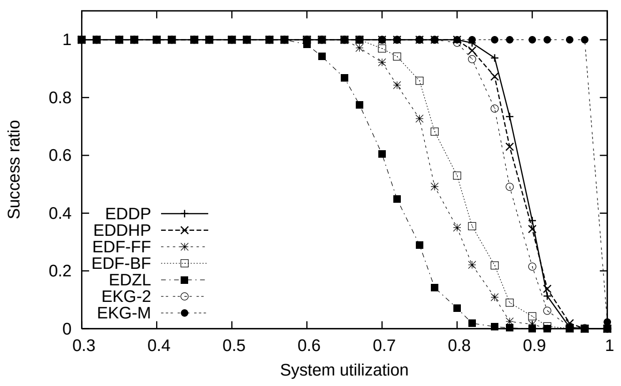

Figure 12: Success ratio: $U_{max} = 1.0$, $M = 8$

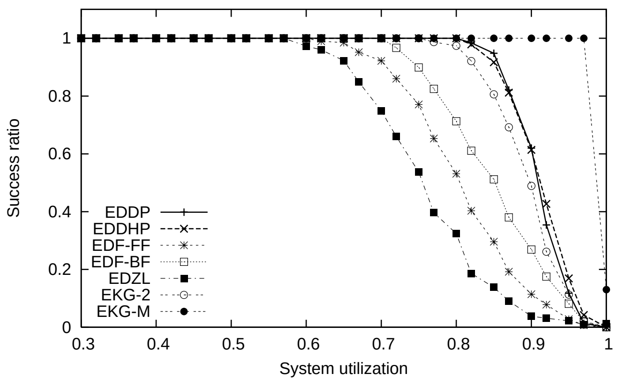

Figure 11: Success ratio: $U_{max} = 1.0$, $M = 4$

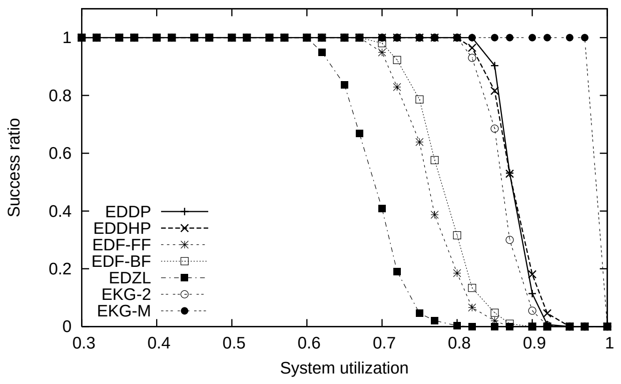

Figure 13: Success ratio: $U_{max} = 1.0$, $M = 16$

to have room for improvement in [9], the potential schedulability of EDZL may be better than the simulated results. However, no better schedulability test has been presented at the moment.

Figure 11, Figure 12, and Figure 13 depict the success ratios for each algorithm with respect to task sets in which the utilization of every individual task ranges within $[0.01, 1.0]$. EDDP, EDDHP, and EKG-2 are still competitive. However, contrary to the previous case, EDDP slightly outperforms EDDHP and EKG-2 in most cases. In general, the schedulability of an algorithm is likely to drop under the presence of heavy tasks, since the condition of the system leads to the worst case. Thus, EDDP performs better than EDDHP due to its higher worst-case utilization bound of 65%.

While the above three algorithms are competitive, EDF-BF declines the maximum schedulable utilization to 67%. The schedulable utilization for EDF-FF is also declined to 60%. This inferiority of EDF-BF and EDF-FF is due to the the presence of heavy tasks. Even though the total remaining utilization of all the processors is sufficient, a heavy task may fail to be assigned to any processor, since the acceptance of a task does not depend on the total remaining utilization but on the remaining utilization of every individual processor.

EDZL, on the other hand, improves a schedulable utilization up to 60% that is about 15% higher than the previous case. The schedulable utilization for EDZL is dominated by the opportunity that the number of the tasks which can have the zero laxity at the same time becomes greater than the number of the processors. The case in which there are heavy tasks leads the total number of the tasks to be smaller than the case in which there are only light tasks, which results in less opportunity that the number of the tasks have can have the zero laxity at the same time is greater than the number of the processors. As a result, EDZL performs well in the presence of heavy tasks.

5.3 The Average Number of Preemptions

For counting preemptions, the interval of the simulations is set the smaller of the least common multiple of the task periods in the given task set and 2^{32}. For each algorithm, the average number of preemptions in the 1000 task sets is calculated every system utilization. Then, the number of preemptions for each algorithm relative to that for EDDP is calculated only for system utilizations at which both EDDP and the measured algorithm have success ratios of 100%.

Figure 14, Figure 15, and Figure 16 depict the average number of preemptions for each algorithm with respect to task sets in which the utilization of every individual task ranges within $[0.01, 0.5]$. EDDP has slightly fewer preemptions than EDDHP. Since EDDP makes several processors which have only one heavy task, no preemptions occur on those processors. As a result, the number of preemptions for EDDP is totally reduced compared to EDDHP. EDF-FF

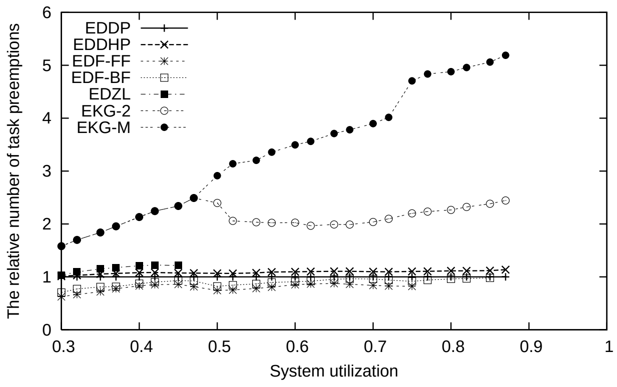

Figure 14: Task preemptions: $U_{max} = 0.5$, $M = 4$

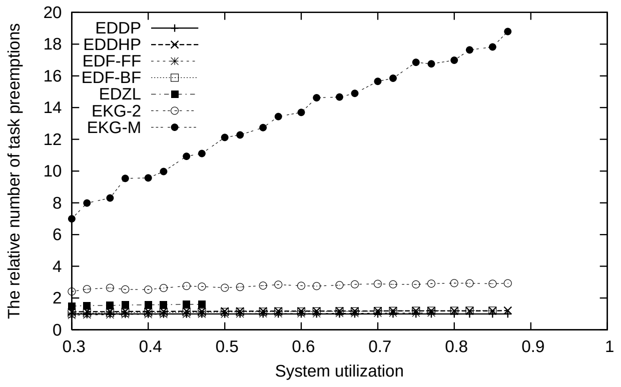

Figure 16: Task preemptions: $U_{max} = 0.5$, $M = 16$

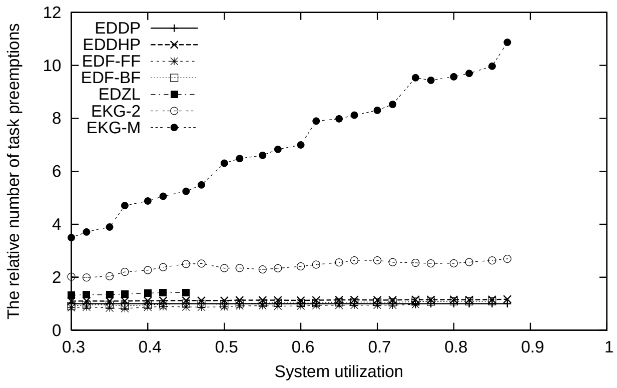

Figure 15: Task preemptions: $U_{max} = 0.5$, $M = 8$

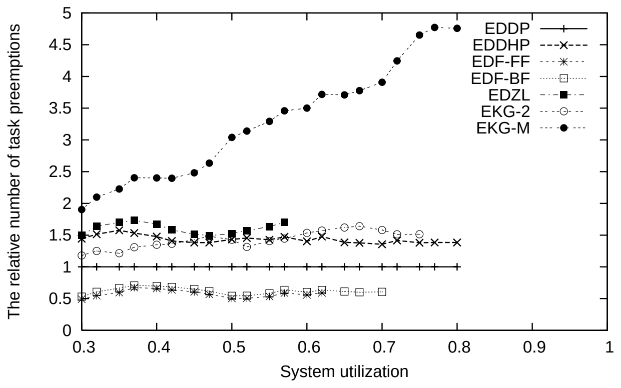

Figure 17: Task preemptions: $U_{max} = 1.0$, $M = 4$

and EDF-BF have around $0.65 \sim 0.85$ times as many preemptions as EDDP. The primary reason why the numbers of preemptions for EDF-FF and EDF-BF are smaller than those for the others is that those algorithms simply schedule the tasks by EDF on each processor without any interference from other processors, while the other algorithms have more or less interferences among the processors.

EDZL has slightly more preemptions, which is about 1.5 times, than EDDP. A global scheduling method has only one scheduler which handles all the tasks, thereby the ordering of the task priority is likely to be changed. Since a preemption occurs every time the ordering of the task priority is changed, those algorithms cause more preemptions.

EKG causes more preemptions than the other algorithms regardless of the parameter k. More specifically, EKG-2 causes around $2.0 \sim 2.5$ times as many preemptions as EDDP, though the schedulable utilization is competitive to EDDP. EKG-M causes far too many preemptions. EKG preempts a migratable task twice during every interval of job arrivals on the related processors, which inevitably increase preemptions.

Figure 17, Figure 18, and Figure 19 depict the average number of task preemptions for each algorithm with respect to task sets in which the utilization of every individual task ranges within $[0.01, 1.0]$. In this case, EDDP has about $1.3 \sim 2.0$ times as many preemptions as EDF-FF and EDF-BF. According to EDDP, the second portion of a migratable task is preempted when its corresponding first portion portion is dispatched or preempted on the neighbor processor, while such an extra preemption never occurs for EDF-FF and EDF-BF. Since the execution time of each task tends to be long due to $U_{max} = 1.0$, the first and second portions of a migratable task are easily overlapped. As a result, EDDP causes more extra preemptions. However, EDDP suppresses the number of preemptions to about $0.5 \sim 0.7$ times as many as EKG-2 that takes a similar splitting task approach. EKG-M causes at most about 20 times as many preemptions as EDDP in exchange for its optimality. Meanwhile, EDZL causes about 1.5 times as many preemptions as EDDP.

6. CONCLUSION

This paper presented the EDDP algorithm for efficient scheduling of recurrent real-time tasks on multiprocessors. The algorithm was designed based on the portioned scheduling technique. The schedulable condition for all tasks to meet deadlines was derived by scheduling analysis. The worst-case system utilization with guarantee of timing constraints was then proven to be 65%. To the best of our knowledge, any non-optimal algorithms can achieve higher worst-case system utilization.

Beyond the theoretical analysis, the effectiveness of EDDP was validated by several sets of simulations in terms of the achievable system utilization and the average number of

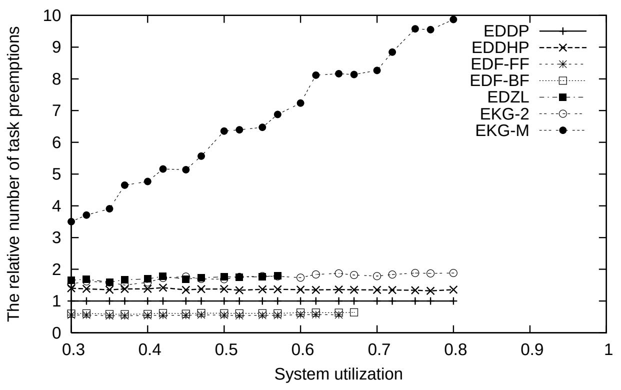

Figure 18: Task preemptions: $U_{max} = 1.0$, $M = 8$

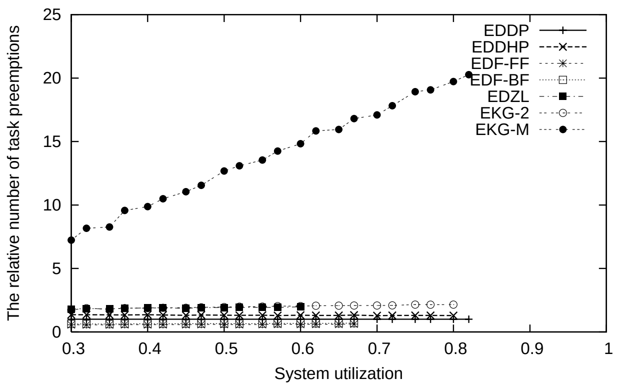

Figure 19: Task preemptions: $U_{max} = 1.0$, $M = 16$

preemptions. The simulation results showed that EDDP achieved higher system utilizations with a smaller number of preemptions, compared to EDZL, EDDHP and EKG-2. Although EKG with large k outperformed EDDP in terms of achievable system utilization, it caused more preemptions. EDF-FF and EDF-BF offered less preemptions, however their achievable system utilizations were much lower than EDDP. In consequence, this paper believes that EDDP can be a new alternative for the scheduling of recurrent real-time tasks on multiprocessors.

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

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

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

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

[6] J. Calandrino, H. Leontyev, A. Block, U. Devi, and J. Anderson. LITMUS^{RT}: A Testbed for Empirically Comparing Real-Time Multiprocessor Schedulers. In *Proc. of the IEEE Real-Time Systems Symposium*, pages 111–123, 2006.

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

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

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

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

[11] K. Funaoka, S. Kato, and N. Yamasaki. Work-Conserving Optimal Real-Time Scheduling on Multiprocessors. In *Proc. of the Euromicro Conference on Real-Time Systems*, pages 13–22, 2008.

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

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

[14] 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.

[15] J. Lopez, J. Diaz, and D. Garcia. Utlization Bounds for EDF Scheduling on Real-Time Multiprocessor Systems. *Real-Time Systems*, 28:39–68, 2004.

[16] X. Piao, S. Han, H. Kim, M. Park, Y. Cho, and S. Cho. Predictability of Earliest Deadline Zero Laxity Algorithm for Multiprocessor Real-Time Systems. In *Proc. of the IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing*, pages 359–364, 2006.

[17] A. Srinivasan and S. Baruah. Deadline-based Scheduling of Periodic Task Systems on Multiprocessors. *Information Processing Letters*, 84(2):93–98, 2002.

[18] A. Srinivasan, P. Holman, J. Anderson, and S. Baruah. The Case for Fair Multiprocessor Scheduling. In *Proc. of the IEEE International Parallel and Distributed Processing Symposium*, pages 22–26, 2003.

[19] H. Wei, Y. Chao, S. Lin, K. Lin, and W. Shih. Current Results on EDZL Scheduling for Multiprocessor Real-Time Systems. In *Proc. of the IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, pages 120–130, 2007.