Semi-Partitioned Scheduling of Sporadic Task Systems on Multiprocessors [*]

Shinpei Kato[†], Nobuyuki Yamasaki[‡], and Yutaka Ishikawa[†]
[†]Department of Computer Science, The University of Tokyo, Tokyo, Japan
[‡]Department of Information and Computer Science, Keio University, Yokohama, Japan
shinpei@il.is.s.u-tokyo.ac.jp, yamasaki@ny.ics.keio.ac.jp, ishikawa@is.s.u-tokyo.ac.jp

Abstract

This paper presents a new algorithm for scheduling of sporadic task systems with arbitrary deadlines on identical multiprocessor platforms. The algorithm is based on the concept of semi-partitioned scheduling, in which most tasks are fixed to specific processors, while a few tasks migrate across processors. Particularly, we design the algorithm so that tasks are qualified to migrate only if a task set cannot be partitioned any more, and such migratory tasks migrate from one processor to another processor only once in each period. The scheduling policy is then subject to Earliest Deadline First. Simulation results show that the algorithm delivers competitive scheduling performance to the state-of-the-art, with a smaller number of context switches.

1 Introduction

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

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

The partitioned scheduling class has such an advantage that reduces a problem of multiprocessor scheduling into a set of uniprocessor one after tasks are partitioned. In addition, it does not incur runtime overhead as much as global scheduling, since tasks never migrate across processors. However, there is a disadvantage in schedulability bounds. In fact, any partitioned scheduling approaches may cause deadlines to be missed on m processors, if the total processor utilization exceeds $(\beta m + 1)/(\beta + 1)$, where $\beta = \lfloor 1/\alpha \rfloor$ and α is a maximum utilization of individual tasks [18]. Let $\alpha = 1$ and $m \rightarrow \infty$, then the worst-case processor utilization is bounded by 50%.

The global scheduling class is attractive in the worst-case schedulability. In this class, Pfair [7] and LLREF [10] are known to be optimal algorithms. Any task sets are scheduled successfully by those algorithms, if the processor utilization is less than or equal to 100%. However, the number of migrations and context switches is often criticized. This scheduling class also provides concise algorithms, such as EDZL [11] and EDCL [14], which perform with less context switches than the optimal ones, but the absolute worst-case processor utilization is still 50%.

Recent work [1, 4, 2, 3, 13, 15] have made available a new class, called *semi-partitioned scheduling* in this paper, for the purpose of finding a balance point between partitioned scheduling and global scheduling. In this scheduling class, most tasks are fixed to specific processors as partitioned scheduling to reduce the number of migrations, while a few tasks may migrate across processors to improve available processor utilization as much as possible.

This paper presents a new algorithm for semi-partitioned scheduling of sporadic task systems with arbitrary deadlines on identical multiprocessor platforms. We primarily aim at delivering competitive scheduling performance to the state-of-the-art, with a smaller number of context switches, in terms of the generic-case schedulability rather than the absolute worst-case schedulability. The algorithm also brings several benefits as summarized below.

- The algorithm allows tasks to migrate across processors only if they cannot be assgined to any invididual processors, to strictly dominate the previous algorithms based on classical partitioned scheduling.

[*]This work is supported by the fund of Research Fellowships of the Japan Society for the Promotion of Science for Young Scientists.

- The algorithm allows migratory tasks to migrate from one processor to another processor only once in each period, to bound the number of context switches to be smaller than the complementary algorithms of the same sort based on semi-partitioned scheduling. This property also helps to keep runtime processor performance as much as possible, particularly with respect to local caches.

- The algorithm conforms the scheduling policy to Earliest Deadline First (EDF) [17], to make available the prior analytical results of EDF.

- The algorithm is available for all categories of periodic and sporadic task systems with implicit, constrained, and arbitrary deadlines.

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

2 Prior Work

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

EKG [4] is designed to guarantee all tasks to meet deadlines for implicit-deadline periodic task systems. Unlike EDF-fm, migratory tasks are executed in certain time slots, while fixed tasks are scheduled according to EDF. The achievable processor utilization is traded with the number of preemptions and migrations, by a parameter k. The configuration of $k = m$ on m processors leads EKG to be optimal, with more preemptions and migrations.

In the later work [2], EKG is extended for sporadic task systems. The extended algorithm is also parametric with respect to the length δ of the time slots reserved for migratory tasks. The authors claim that $1 \leq \delta \leq 4$ seems reasonable. EDF-SS [3] is a further extension of the algorithm for arbitrary-deadline systems. It is shown by simulations that EDF-SS offers a significant improvement on schedulability over EDF-FFD [6], the best performer among partitioned scheduling algorithms. We are not aware of any other algorithms, designed based on semi-partitioned scheduling, that are effective to arbitrary-deadline systems.

EDDHP [13] and its extension, EDDP [15], are designed in consideration of reducing context switches. The resultant scheduling is based on priorities, and no time slots are reserved for migratory tasks. It is shown by simulations that they also outperform partitioned scheduling algorithms. The worst-case processor utilization is then bounded by 65% for implicit-deadline systems.

We have several concerns for the previous algorithms mentioned above. First, tasks migrate across processors, even though they can be assigned to individual processors. Hence, we are not sure that those algorithms are truly more effective than classical partitioned scheduling approaches. Then, such tasks may migrate in and out of the same processor many times within the same period, which is likely to cause the cache hit ratio to decline. The number of context switches is also problematic due to repetition of migrations. In addition, optional techniques for EDF, such as SRP [5] and TBS [20], may not be easily available, since the scheduling policy is more or less modified from EDF. In this paper, we address those concerns.

3 System Model

The system contains m identical processors P_1, P_2, ..., P_m, and a set of n sporadic tasks $\tau = \{T_1, T_2, ..., T_n\}$. Each sporadic task T_i is characterized by a tuple (c_i, d_i, p_i), where c_i is a worst-case computation time, d_i is a relative deadline, and p_i is a minimum inter-arrival time that is also referred to as a period. The utilization of T_i is then denoted by $u_i = c_i/p_i$. For any T_i, $c_i \leq d_i$ and $c_i \leq p_i$ are satisfied. In this paper, we consider such arbitrary-deadline systems that allow T_i to have any value of d_i. Note that the presented algorithm is also effective to constrained-deadline systems that meet $d_i \leq p_i$ as well as classical implicit-deadline systems that meet $d_i = p_i$.

Each task T_i generates an infinite sequence of jobs, each of which has a worst-case computation time equal to c_i. A job of T_i released at time t has a deadline at time $t + d_i$. Any inter-arrival intervals of successive jobs of T_i are separated by at least length of p_i.

All tasks are independent and preemptive. An individual job is not allowed to execute in parallel. When the deadline of a task is greater than its period, it is possible that a job of the task may be released before the preceding job of the task has completed. In this case, two jobs of the same task are allowed to execute in parallel.

4 New Algorithm

We present a new algorithm, called **EDF with Window-constraint Migration (EDF-WM)**, based on the concept of semi-partitioned scheduling. Given the migration and preemption costs, EDF-WM allows a task to migrate, only if

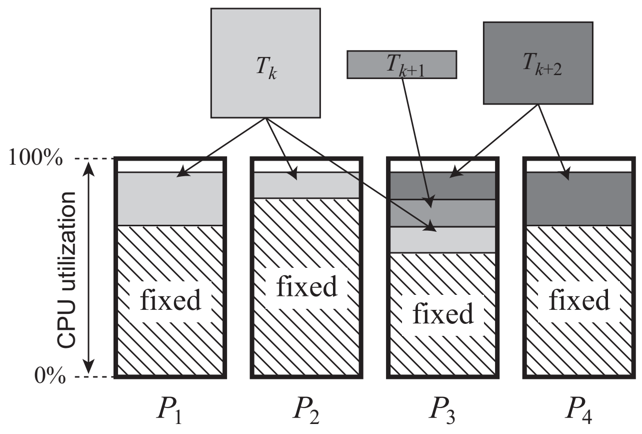


Figure 1. Semi-partitioning.

no individual processors have remaining capacity enough to accept the full share of the task, in such a manner that it is never migrated back to the same processor within the same period, once it is migrated from one processor to another processor. The same approach is introduced in [16] but is based on the fixed-priority policy, while we consider the algorithm with its basis on EDF in this paper.

4.1 Basic Approach

The approach considered here is a superset of traditional partitioned scheduling. Each task is assigned to an individual processor according to a first-fit heuristic[1], as long as it can be. A task is then decided to be migratory, only when no individual processors have remaining capacity enough to accept the full share of the task. In terms of utilization share, the task is *split* into more than one processor. As a result, EDF-WM strictly dominates the traditional partitioned scheduling approaches. To the best of our knowledge, no previous algorithms based on semi-partitioned scheduling strictly dominate them.

Figure 1 shows an example of semi-partitioning on four processors. This example assumes that tasks with smaller index than k are already assigned (fixed) to processors. We then consider a case in which a task T_k cannot be assigned to any individual processors. In traditional partitioning, such T_k is not schedulable. In semi-partitioning, on the other hand, T_k is split across more than one processor, for instance three processors P_1, P_2, and P_3.

A task is split in such a way that a processor is filled to capacity by the portion of the task assigned to the processor. However, only the last processor to which the portion is assigned may not be filled to capacity, because the size of the portion is not necessarily equal to the remaining capacity of the processor. Thus, in the example, no tasks will be assigned to P_1 and P_2, while some tasks may be later assigned to P_3. Note that the bound of processor utilization to be filled to depends on algorithms.

[1]In fact, any partitioning methods are available under EDF-WM.

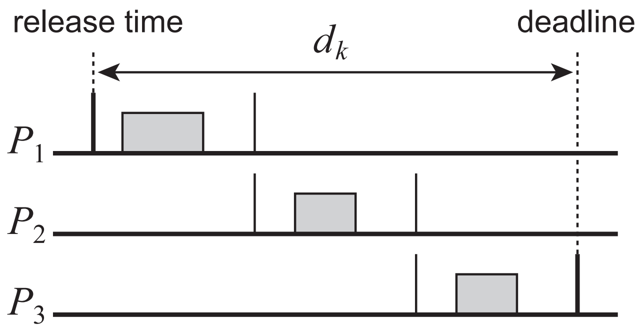


Figure 2. EDF-WM scheduling.

The remaining tasks are also assigned to processors in the same manner. Since the utilization of T_{k+1} is small enough to be fixed to P_3 or P_4, it is assigned to P_3 according to a first-fit heuristic. T_{k+2} is then split across P_3 and P_4. As a result, a processor may include more than one migratory task, such as P_3 in the example. Note that a task is not necessarily split across continuous processors, though it is done in the example for simplicity of explanation.

In the scheduling phase, a migratory task must be executed exclusively among processors, since an individual job is not allowed to execute in parallel. EDF-WM therefore splits the deadline of each migratory task into the same number of windows as the processors across which the task is qualified to migrate.

Figure 2 shows an example of scheduling a migratory task T_k that is split across the three processors, as the previous example. The deadline is split into three windows. The task is presumed to be released at the beginning of a window and to have a deadline at the end of a window. The task is then scheduled according to EDF, based on such a pseudo-deadline, on each processor. Since the task is never executed out of the window, it is guaranteed that multiple processors never execute the task simultaneously.

An issue of concern here is how to determine the length of a window and the amount of time which a migratory task is allowed to consume within a window. We will describe the details of the algorithm in Section 4.3.

4.2 Demand Bound Function

Before the algorithm description of EDF-WM, we introduce the demand bound function [8] that will be used in the remainder of this section. The demand bound function $dbf(T_i, L)$, given by Equation (1), computes the maximum amount of time, so-called *processor demand*, consumed by all jobs of a task T_i that have both release times and dead-

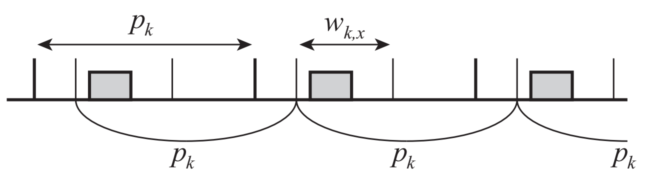


Figure 3. Each window of a migratory task is separated by at least length of the period.

lines within an interval of length L.

$$dbf(T_i, L) = \max\left(0, \left\lfloor \frac{L - d_i}{p_i} \right\rfloor + 1\right) c_i \tag{1}$$

It has been shown [8] that an EDF-feasibility of arbitrarily-deadline sporadic task systems can be tested by the demand bound function: all tasks are guaranteed to meet deadlines by EDF on single processors, if and only if the following condition holds for $\forall L > 0$.

$$\sum_{T_i \in \tau} dbf(T_i, L) \le L \tag{2}$$

In fact, we only need to verify the condition above for those values of L that are aligned with deadlines of jobs. According to [9], the length of check points is also reduced to a finite number of L^* given by Equation (3), where L_a is given by Equation (4) and $\mathcal{H}$ is a hyperperiod of τ.

$$L^* = \min(L_a, \mathcal{H}) \tag{3}$$

$$L_a = \max\left\{ \frac{\sum_{T_i \in \tau}(p_i - d_i)u_i}{1 - \sum_{T_i \in \tau} u_i}, \max(d_i \mid T_i \in \tau) \right\} \tag{4}$$

4.3 Algorithm Description: Semi-Partitioning

The scheduling policy of EDF-WM is strictly subject to EDF, once tasks are assigned to processors successfully, apart from that the relative deadline of each migratory task is transformed to the length of a window. All we have to do in the semi-partitioning phase is thus to assign the window and the processor demand to each migratory task in such a way that the processor utilization is maximized without timing violations of the already-assigned tasks. The following describes the details of the algorithm.

Let T_k be a migratory task whose share is assigned to a processor P_x. We denote the length of a window assigned to T_k on P_x by $w_{k,x}$. Here the value of $w_{k,x}$ is fixed, once it is determined. Thus, each window of T_k is separated by at least length of p_k on any P_x, as shown in Figure 3. This means that we can regard T_k as an arbitrary-deadline sporadic task with a (relative) deadline $w_{k,x}$ and a period p_k. Henceforth, such a pseudo-deadline of T_k is denoted by $d'_{k,x} = w_{k,x}$. The amount of time that T_k is allowed to consume in $d'_{k,x}$ is also denoted by $c'_{k,x}$.

On splitting a task, we need to determine $d'_{k,x}$ and $c'_{k,x}$ for each processor P_x so that all tasks are guaranteed schedulable. To this end, EDF-WM makes use of the demand bound function. For a fixed (non-migratory) task, the processor demand is obviously given by Equation (1). For a migratory task T_k, on the other hand, we need to replace d_k and c_k with $d'_{k,x}$ and $c'_{k,x}$ for each P_x. The processor demand of T_k consumed on P_x is thus given by Equation (5), assuming that $d'_{k,x}$ and $c'_{k,x}$ are known.

$$dbf(T_k, L) = \max\left(0, \left\lfloor \frac{L - d'_{k,x}}{p_k} \right\rfloor + 1\right) c'_{k,x} \tag{5}$$

We now explain how to compute the values of $d'_{k,x}$ and $c'_{k,x}$. Let τ_x be a set of tasks that are already assigned to P_x. Then, $d'_{k,x}$ and $c'_{k,x}$ must be such values that satisfy Inequation (6) for $\forall L > 0$.

$$\sum_{T_i \in \tau_x} dbf(T_i, L) + dbf(T_k, L) \le L \tag{6}$$

In order to calculate one of $c'_{k,x}$ and $d'_{k,x}$ back from Inequation (6), the other of them must be known. Due to this dilemma, EDF-WM gives $d'_{k,x}$ such that $d'_{k,x} = d_k/s$, where s is the number of processors across which T_k is split. Hereafter, $d'_{k,x} = d_k/s$ for $\forall x$ is unified by $d'_k = d_k/s$, since they all have the same value. With such d'_k, we determine $c'_{k,x}$ as the minimum of Equation (7) for $\forall L$ ($d'_k \le L \le L^*$) aligned with deadlines of jobs. Note that those values of $L < d'_k$ can be ignored, because the processor demand of T_k is always zero.

$$c'_{k,x} = \frac{L - \sum_{T_i \in \tau_x} dbf(T_i, L)}{\left\lfloor \frac{L - d'_k}{p_k} \right\rfloor + 1} \tag{7}$$

To improve schedulability, it is better to split T_k across processors so that $c'_{k,x}$ is maximized. The value of $c'_{k,x}$ is dominated by two factors. One is s that affects d'_k. The other is $\sum_{T_i \in \tau_x} dbf(T_i, L)$ that affects the available processor demand. We therefore make the following policy to split T_k.

- According to Equation (7), $c'_{k,x}$ is monotonically increasing with respect to $d'_k = d_k/s$. Hence, the value of s should be small. In consideration of this, we first assume $s = 2$. We then increment s until Inequation (8) is met, where π_k denotes a set of s processors under consideration, across which T_k is split.

$$\sum_{P_x \in \pi_k} c'_{k,x} \ge c_k \tag{8}$$

- T_k is then split across such s processors that provide greater values of $c'_{k,x}$.
- If s reaches m but Inequation (8) is not met yet, T_k cannot be successfully split across the processors, which means that T_k is not schedulable with the algorithm.

```
1.  $s = 2$;
2.  if $s > m$ then return $FAIL$; end if
3.  $d'_k = d_k/s$;
4.  for $x = 1$ to $m$ do
5.      $c'_{k,x} = calc\_exec\_time(T_k, P_x)$;
6.  end for
7.  let $c'_{k,z}$ be the $s$th greatest in $\{c'_{k,x} \mid 1 \leq x \leq m\}$;
8.  $\pi_k = \{P_x \mid c'_{k,x} \geq c'_{k,z}\}$;
9.  if $\sum_{P_x \in \pi_k} c'_{k,x} < c_k$ then
10.     $s = s + 1$;
11.     goto 2.;
12. else
13.     $c'_{k,z} = c'_{k,z} - (\sum_{P_x \in \pi_k} c'_{k,x} - c_k)$;
14.     for each $P_x \in \pi_k$ do
15.         $\tau_x = \tau_x \cup \{T_k\}$;
16.     end for
17. end if
18. return $SUCCESS$;
```

Figure 4. Pseudo-code of $split\_task(T_k)$**.**

We here need to calculate the length of L^*. Equation (4) requires $u'_{k,x} = c'_{k,x}/p_k$, while we will obtain $c'_{k,x}$ afterwards. In fact, Equation (4) can be transformed to Equation (9) for τ_x and T_k, where $D = \max(d_i \mid T_i \in \tau_x)$.

$$L_a = \max\left\{\frac{\sum_{T_i\in\tau_x}(p_i - d_i)u_i + (p_k - d'_k)u'_{k,x}}{1 - \sum_{T_i\in\tau_x} u_i - u'_{k,x}}, D\right\} \qquad (9)$$

Equation (9) implies that L_a is monotonically increasing with respect to $u'_{k,x}$. Here, $u'_{k,x}$ must satisfy Inequation (10), otherwise a task set assigned to P_x is not schedulable.

$$u'_{k,x} < 1 - \sum_{T_i\in\tau_x} u_i \Leftrightarrow c'_{k,x} < \left(1 - \sum_{T_i\in\tau_x} u_i\right) p_k \qquad (10)$$

Based on the discussion above, we first assume $L_a = \mathcal{H}$ and $c'_{k,x} = (1 - \sum_{T_i\in\tau_x} u_i)p_k$. If the value of $c'_{k,x}$ is reduced by Equation (7), we recalculate Equation (9). Since L_a is monotonically increasing with respect to $c'_{k,x}$, the value of L_a is also reduced. The procedure of calculating $c'_{k,x}$ is then finished, when L reaches L_a or when the renewed value of L_a is less than or equal to the current value of L.

Finally, the pseudo-code of the function, $split\_task(T_k)$, for splitting a task T_k is indicated in Figure 4. It first sets $s = 2$, and then calculates $c'_{k,x}$ for each processor P_x. If $\sum_{P_x\in\pi_k} c'_{k,x}$ does not reach c_k, the same procedure is repeated with $s = s+1$. Otherwise, T_i is assigned to each of $P_x \in \pi_k$. Note that $c'_{k,z}$ needs to be adjusted at line 13, since P_z is the last processor that is not necessarily filled to capacity by the last portion of T_i. The algorithm returns failure only when s exceeds m (line 2).

```
1.  $c'_{k,x} = (1 - \sum_{T_i\in\tau_x} u_i)p_k$; $L^* = \mathcal{H}$;
2.  for each $T_i \in \tau_x \cup \{T_k\}$ do
3.      $\alpha = \lceil (d'_k - d_i)/p_i \rceil$;
4.      while $L = \alpha p_i + d'_i < L^*$ do
5.          $c = \{L - dbf(\tau_x, L)\}/\{\lfloor (L - d'_k)/p_k \rfloor + 1\}$;
6.          if $c < c'_{k,x}$ then
7.              $c'_{k,x} = c$;
8.              $L^*$ = Equation (9), where $u'_{k,x} = c'_{k,x}/p_k$;
9.          end if
10.         $\alpha = \alpha + 1$;
11.     end while
12. end for
22. return $c'_{k,x}$;
```

Figure 5. Pseudo-code of $calc\_exec\_time(T_k, P_x)$**.**

Due to limitation of space, we illustrate the function, $calc\_exec\_time(T_k, P_x)$, assosiated with calculating $c'_{k,x}$ (at line 5) in Figure 5. In the pseudo-code, α is a natural number, and $dbf(\tau_x, L)$ represents $\sum_{T_i\in\tau_x} dbf(T_i, L)$. For fixed tasks, let us define $d'_i = d_i$ for unifying the description of deadlines, since migratory tasks are considered to have deadlines $d'_i = d_i/s$, as we defined before. Then, it finds out the minimum of $c'_{k,x}$ for all $L = \alpha p_i + d'_i < L^*$. The value of α set at line 3 guarantees $L \geq d'_k$. The value of L^* is then renewed every time $c'_{k,x}$ is renewed (line 6 to 9).

The time complexity for calculating each $c'_{k,x}$ by the $calc\_exec\_time$ function is closely bounded to that of the demand bound function. Since we need to call the function m times to obtain $c'_{k,x}$ for all m processors (line 4-6 in Figure 4), and this procedure may be repeated up to $m - 1$ times ($s = 2$ to m in Figure 4), the resultant time complexity may be high, as compared to a simple schedulability test based on the demand bound function. However, we see this time complexity problem only when a task cannot be assigned to any individual processor. So the time complexity is traded with schedulability improvements.

Task sorting problem. Most partitioning algorithms [6] sort a task set before they assign tasks to processors, to improve achievable processor utilization. We consider that EDF-WM is also likely to perform better, if a task set is sorted in non-increasing order of deadline. The reason is given as follows. The value of $c'_{k,x}/d'_{k,x}$ is greater as the value of $d'_{k,x}$ is smaller, since the available processor demand within a time interval of any length is fixed. It is clear that the greater $c'_{k,x}/d'_{k,x} = sc'_{k,x}/d_k$ is, the greater $u'_{k,x} = c'_{k,x}/p_k$ is. We know that T_k is split when most tasks are partitioned, so the value of d_k is likely small by sorting a task set in non-increasing order of deadline, which leads to a greater value of $u'_{k,x}$. Thus, task sorting is effective for EDF-WM to improve achievable processor utilization.

```
function $schedule(P_x)$:
1.  if $T_c$ exists then $e_c = e_c - (t - t_{last})$; end if
2.  $t_{last} = t$;
3.  select $T_k$ such that $\overline{d}_k = \min\{\overline{d}_i \mid \overline{a}_i \leq t, T_i \in \gamma_x\}$;
4.  if $T_k \neq T_c$ then
5.      if $T_k$ is a migratory and
        $y = \min\{z \mid P_z \in \pi_k, z > x\}$ exists then
6.          $set\_timer(migrate(T_k, P_x, P_y), t + e_k)$;
7.      end if
8.      if $T_c$ is a migratory task then
9.          $delete\_timer(T_c)$;
10.     end if
11.     switch to $T_k$ from $T_c$;
12. end if

function $migrate(T_k, P_x, P_y)$:
13. $\gamma_x = \gamma_x \setminus \{T_k\}$; $\gamma_y = \gamma_y \cup \{T_k\}$;
14. $\overline{a}_k = \overline{d}_k$; $\overline{d}_k = \overline{a}_k + d'_k$; $e_k = c'_{k,y}$;
15. $set\_timer(schedule(P_y), \overline{a}_k)$;

function $complete(T_k, P_x)$:
16. $\gamma_x = \gamma_x \setminus \{T_k\}$;
17. $schedule(P_x)$;

function $release(T_k, P_x)$:
18. $\gamma_x = \gamma_x \cup \{T_k\}$; $\overline{a}_k = t$;
19. if $T_k$ is a migratory task then
20.     $\overline{d}_k = t + d'_k$; $e_k = c'_{k,x}$; else $\overline{d}_k = t + d_k$; $e_k = c_k$;
21. end if
22. $schedule(P_x)$;
```

Figure 6. Pseudo-code of EDF-WM scheduler.

4.4 Algorithm Description: Scheduling

Figure 6 shows the pseudo-code of EDF-WM scheduler. In the pseudo-code, $\overline{a}_k$, $\overline{d}_k$, and e_k denote the absolute release time, the absolute deadline, and the remaining execution time of a task T_k respectively. γ_x is a set of ready tasks on P_x. T_c represents a current task on P_x. t is a current time, and t_{last} is a last time at which the scheduler is invoked. Due to limitation of space, we define $set\_timer(func, t)$ as a function that sets a timer to invoke the specified function $func$ at time t. We also define $delete\_timer(T_k)$ as a function that deletes a timer set previously. Most operating systems, e.g. Linux, prepare those timer functions.

All tasks assigned to P_x are scheduled in $schedule(P_x)$. The function first records the remaining execution time of the current task (line 1), and then selects such a task T_k that has the earliest deadline in a set of ready tasks with release times earlier than the current time (line 3). If T_k is a migratory task, we set a timer to preempt T_k and migrate it to the next processor at time $t+e_k$ (line 5-7), because it is not allowed to consume the amount of time beyond $c'_{k,x}$ on P_x. Note that, for a migratory task, e_k represents the remainder of $c'_{k,x}$ on each P_x. The timer set here is deleted, if T_k is preempted later before it consumes $c'_{k,x}$ (line 8-10). Finally, the context is switched from T_c to T_k, if they are different tasks (line 11).

A migratory task T_k is migrated from P_x to P_y by $migrate(T_k, P_x, P_y)$. It removes T_k from γ_x and inserts it to γ_y (line 11). It then updates $\overline{a}_k$, $\overline{d}_k$, and e_k (line 12). Since T_k will have a pseudo-release time $\overline{a}_k$ on P_y, a timer function is set to invoke $schedule(P_y)$ at time $\overline{a}_k$. Although T_k is now ready on P_y, it is never selected for execution before $\overline{a}_k$, since only the tasks with release times set earlier than the current time will be scheduled, as described at line 2.

We assume that $release(T_k, P_x)$ and $complete(T_k, P_x)$ are called when T_k is released and completes respectively. $schedule(P_x)$ is then called at the end of those functions, since we need to schedule a next task. Note that more than one $release(T_k, P_x)$ may be called at the same time, if more than one task has the same release time. In this case, we do not have to call $schedule(P_x)$ every time but call it once after all such $release(T_k, P_x)$ are executed.

Recall that Figure 6 is a pseudo-code. The algorithm implementation in practice depends on developers. One of our concerns is that EDF-WM needs to use kinds of high-resolution timers for executions of migratory tasks after all. This is a common problem more or less for such algorithms [4, 2, 3, 13, 15] that are based on semi-partitioned scheduling. Implementation issues are left open.

4.5 The Number of Context Switches

In practical real-time systems, the number of context switches should be bounded, so as to estimate runtime overhead. Given that a context is switched only when jobs are preempted or complete in EDF scheduling, the number of context switches is bounded as follows.

Let $np(T_i, L)$ and $nc(T_i, L)$ be the number of preemptions generated by a task T_i and the number of job completions of T_i respectively, within a time interval $[0, L)$.

We first consider T_i as a fixed task. Since the scheduling policy is based on EDF, T_i may generate a preemption every time its job is released, if it has the earliest deadline. Hence, we have $np(T_i, L) \leq \lceil L/p_i \rceil$. The number of job completions of T_i is then bounded to the number of its job releases, so we also have $nc(T_i, L) \leq \lceil L/p_i \rceil$. As a result, $np(T_i, L) + nc(T_i, L) \leq 2\lceil L/p_i \rceil$ is derived.

We then consider T_i as a migratory task split across more than one processor. Let P_x and P_z be the first processor and the last processor respectively: jobs of T_i are released on P_x and complete on P_z. We have $nc(T_i, L) \leq \lceil L/p_i \rceil$ for P_z, and $nc(T_i, L) = 0$ for any other processors. The number of preemptions is bounded as follows.

On the first processor P_x, T_i may generate a preemption every time its job is released, and it also generates a preemption every time it consumes $c'_{i,x}$ time units. Thus, we have $np(T_i, L) \leq 2\lceil L/p_i \rceil$. On the last processor P_z, T_i may generate a preemption every time its assigned window begins. Since the window appears at every p_i, we have $np(T_i, L) \leq \lceil L/p_i \rceil$. On any middle processor P_y if it exists, T_i may generate a preemption every time its assigned window begins, and it also generates a preemption every time it consumes $c'_{i,y}$ time units. As a result, we have $np(T_i, L) \leq 2\lceil L/p_i \rceil$. From the discussion above, we derive the following condition for any T_i.

$$np(T_i, L) + nc(T_i, L) \leq 2 \left\lceil \frac{L}{p_i} \right\rceil \tag{11}$$

Let $ncs(L, P_x)$ be the number of context switches on any processor P_x within a time interval $[0, L)$. It is then clear that $ncs(L, P_x)$ is bounded by Inequation (12).

$$ncs(L, P_x) \leq 2 \sum_{T_i \in \tau_x} \left\lceil \frac{L}{p_i} \right\rceil \tag{12}$$

Let E be the execution cost of one context switch. Finally, we can apply the above analysis to the demand bound function so that the amount of time available for the execution of tasks on a processor P_x within a time interval $[0, L)$ is reduced to $L - ncs(L, P_x) \times E$.

The number of migrations is also bounded in the same way. Let T_i be a migratory task. Since T_i is migrated only once within a period from one processor to another processor, the number of migrations for T_i within a time interval $[0, L)$ is at most $s\lceil L/p_i \rceil$, where s is the number of processors to which the share of T_i is assigned.

5 Evaluation

This section studies the effectiveness of EDF-WM to sporadic task systems with arbitrary deadlines, through several sets of simulations. We compare EDF-WM with EDF-SS [3] and EDF-FFD [6], because (i) we are not aware of any other algorithms, based on semi-partitioned scheduling, that are designed for arbitrary-deadline systems, and (ii) EDF-FFD is found to be the champion among partitioned scheduling algorithms [6]. EDF-SS has a parameter δ that trades schedulability with the number of context switches. Since the authors claim that $1 \leq \delta \leq 4$ seems reasonable [3], we prepare $\delta = 1$ and $\delta = 4$.

The simulation results show that EDF-WM offers competitive performance to EDF-SS, far beyond EDF-FFD, with a small number of context switches. Throughout the simulations, "EDF-WM" denotes the presented algorithm without task sorting. "EDF-WM(sort)" then denotes the one with task sorting in non-increasing order of deadline. "EDF-SS(DT/4)" and "EDF-SS(DT)" are EDF-SS with $\delta = 4$ and $\delta = 1$ respectively.

5.1 Simulation Setup

Each task set is randomly generated, with a set of parameters (u_{sys}, m, u_{min}, u_{max}), so that the total processor utilization gets equal to $u_{sys} \times m$, in which the utilization of an individual task is uniformly distributed in the range of $[u_{min}, u_{max}]$. The minimum inter-arrival of an individual task is randomly determined in the range of $[100, 3000]$. For every task T_i, once u_i and p_i are known, we compute the execution time of T_i by $c_i = u_i \times p_i$. The deadline of T_i is then arbitrarily computed in the range of $c_i < d_i < 2p_i - c_i$.

Due to limitation of space, we conduct simulations with the following setups. The total processor utilization is set every 5%. The number of processors is then set $m = 4$, $m = 8$, and $m = 16$. We limit a pair of (u_{min}, u_{max}) to $(0.1, 1.0)$, $(0.5, 1.0)$, and $(0, 1, 0.5)$.

We generate 1,000,000 task sets for every prepared set of (u_{sys}, m, u_{min}, u_{max}), to assess the schedulability of an algorithm. Throughout simulations, a task set is said to be successfully scheduled, if all tasks in the task set are successfully assigned to processors, since the tasks are guaranteed schedulable under semi-partitioned scheduling as well as partitioned scheduling, once they are successfully assigned to processors. The effectiveness of an algorithm is then estimated by *success ratio*: the ratio of the number of successfully-scheduled task sets.

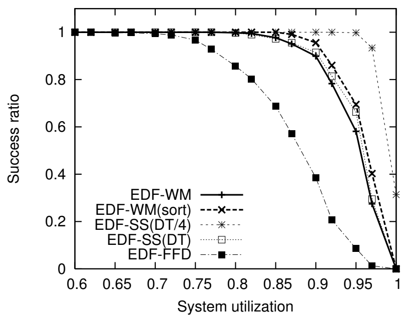
The number of context switches bounded by each algorithm, is also measured to estimate runtime overhead. As we discussed in Section 4.5, the number of context switches for EDF within a time interval of length L is bounded by $2\sum_{T_i \in \tau} \lceil L/p_i \rceil$. We refer to Inequation (12) for EDF-WM. For EDF-SS, we refer to **Theorem 1** demonstrated in [3], but we add the number of job completions, bounded by $\sum_{T_i \in \tau} \lceil L/p_i \rceil$, to the result of the theorem, because only preemptions are considered there.

5.2 Simulation Results

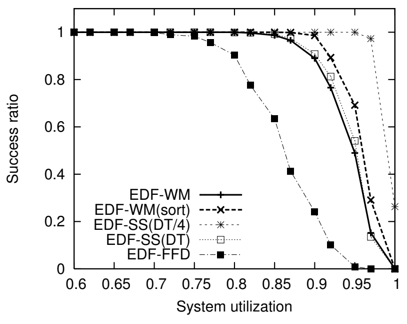
Figure 7, Figure 8, and Figure 9 show the results of the guaranteed schedulability for each algorithm, with respect to different m, u_{min}, and u_{max}. Due to limitation of space, we briefly discuss the results here.

EDF-WM and EDF-SS outperform EDF-FFD, particularly in the presence of heavy tasks with utilization greater than 0.5, as shown in Figure 7 and Figure 8, since heavy tasks are more likely to fail being assigned to individual processors than light tasks in partitioning.

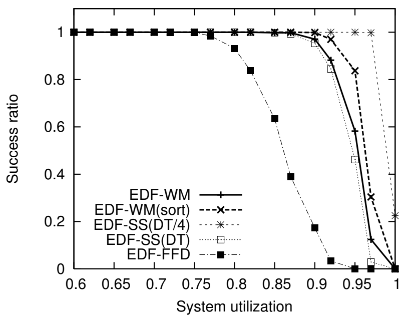
Among the four algorithms: EDF-WM, EDF-WM(sort), EDF-SS(DT/4), and EDF-SS(DT), the best performance



(a) $m = 4$

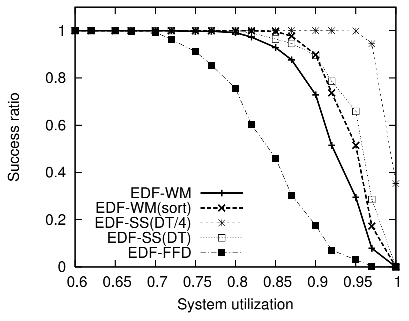


(b) $m = 8$

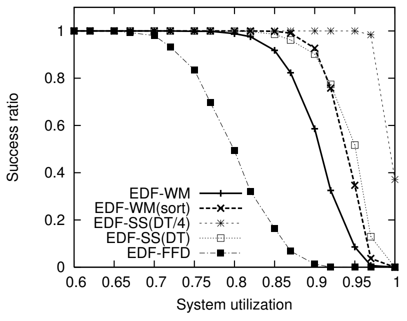


(c) $m = 16$

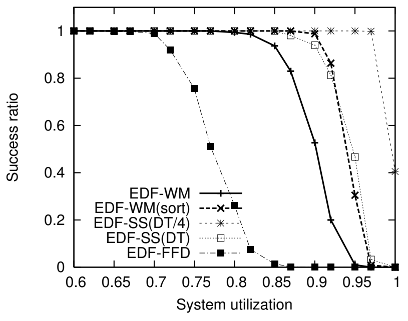
Figure 7. Guaranteed schedulability: $(u_{min}, u_{max}) = (0.1, 1.0)$**.**



(a) $m = 4$



(b) $m = 8$



(c) $m = 16$

Figure 8. Guaranteed schedulability: $(u_{min}, u_{max}) = (0.5, 1.0)$**.**

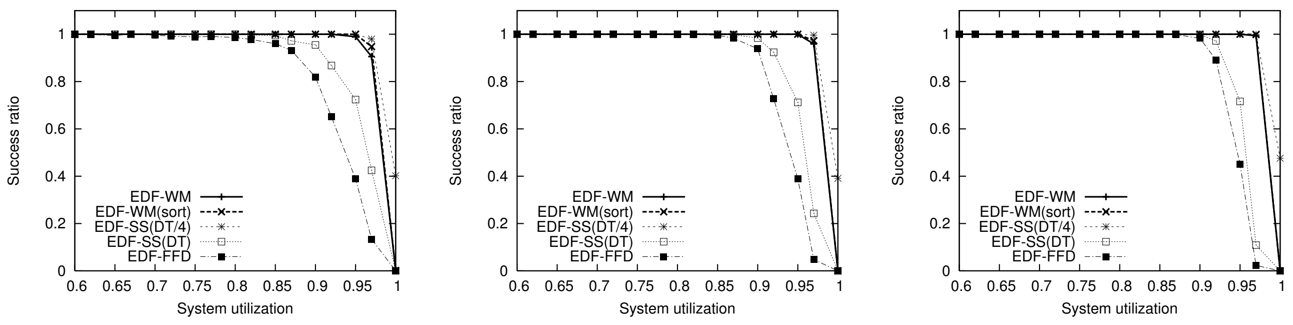
is achieved by EDF-SS(DT/4). EDF-WM(sort) and EDF-SS(DT) are then competitive to each other. According to the results, task sets are often scheduled successfully by those algorithms, even though the system utilization is around 90%. EDF-WM is inferior to them but is still much better than EDF-FFD. Notice that EDF-WM does not sort a task set in advance, while all other ones do. Particularly for task sets containing only light tasks, task sorting does not provide much impact any more, as shown in Figure 9. Given that we may not always have a chance to sort a task set every time a new task is generated at runtime in dynamic systems, we see the effectiveness of EDF-WM.

Figure 10, Figure 11, and Figure 12 show the results of the number of context switches relative to EDF-FFD for each algorithm, with respect to different m, u_{min}, and u_{max}.

Clearly, EDF-SS generates more context switches than EDF-WM and EDF-FFD, occasionally more than two hundreds times as many as them. In fact, the number of context switches for EDF-SS is dependent on the length of periods and deadlines, since the size of slots reserved for migratory tasks is aligned with the minimum of the periods and deadlines. Therefore, the number would be smaller as the range of periods and periods is smaller, whereas it would be more increased as the range is greater. EDF-WM is however not much affected by the range of periods and periods, since the scheduling policy conforms to EDF.

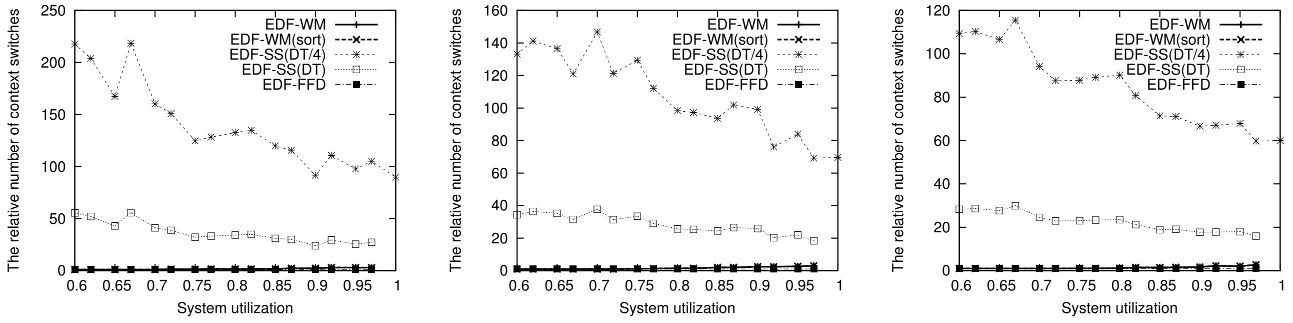
In most cases, EDF-SS(DT/4) causes about four times as many context switches as EDF-SS(DT), regardless of a set of m, u_{min}, and u_{max}. This means that the value of δ reflects the number of context switches in EDF-SS. Meanwhile, the number of context switches for EDF-WM is bounded to at most three times as much as EDF-FFD.

We also observe that the relative number of context switches for EDF-SS is often greater as the system utilization is lower. Remember that EDF-SS is willing to split tasks across processors, even though they can be assigned to individual processors, while no tasks are split by EDF-WM, as long as a task set is partitioned successfully. As a result, the number of context switches for EDF-WM is equal to that for EDF-FFD, when the system utilization is not much high, meanwhile that for EDF-SS is relatively greater. EDF-SS gradually approaches EDF-WM and EDF-FFD as the system utilization approaches 100%, however, even for $\delta = 1$, the number is still five to twenty times as much as them. Recall that EDF-WM(sort) are competitive to EDF-SS(DT) in terms of schedulability, its effectiveness is more obvious when the number of context switches is taken into account.



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

Figure 9. Guaranteed schedulability: $(u_{min}, u_{max}) = (0.1, 0.5)$**.**



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

Figure 10. The worst-case number of context switches relative to EDF-FFD: $(u_{min}, u_{max}) = (0.1, 1.0)$**.**

6 Conclusion

In this paper, we presented a new algorithm for semi-partitioned scheduling of sporadic task systems with arbitrary deadlines on identical multiprocessor platforms. The new algorithm strictly dominates the classical partitioned scheduling approaches, since tasks are qualified to migrate only if they cannot be assigned to any individual processors. The numbers of context switches and migrations are also bounded, since tasks migrate only once in each period. Furthermore, most optional techniques for EDF may be available on the new algorithm with minimum efforts, since the scheduling policy is subject to EDF.

According to the simulation results, the new algorithm offers competitive schedulability to the best known algorithm designed for arbitrary-deadline systems, with a smaller number of context switches.

In future work, we will consider resource augmentation [19] bounds for the new algorithm by applying the techniques presented in [12]. We will also consider windows of different lengths for the purpose of improving schedulability, though the length of windows is evenly split in the presented approach. The evaluation of the algorithm with more variety of setups, as the ones in [6], is left open.

References

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

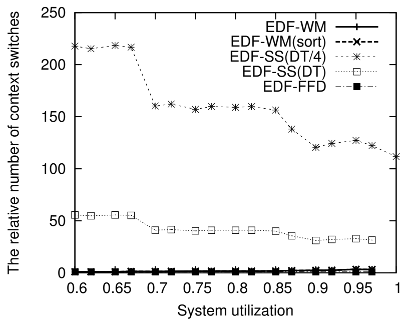
[2] B. Andersson and K. Bletsas. Sporadic Multiprocessor Scheduling with Few Preemptions. In *Proc. of the Euromicro Conference on Real-Time Systems*, pages 243–252, 2008.

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

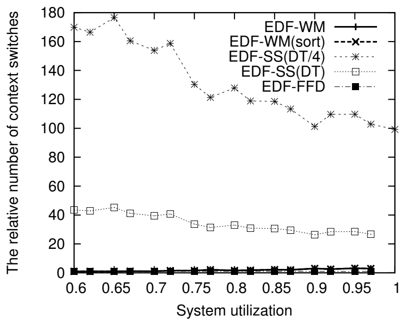
[4] B. Andersson and E. Tovar. Multiprocessor Scheduling with Few Preemptions. In *Proc. of the IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, pages 322–334, 2006.

[5] T.P. Baker. Stack-based Scheduling of Real-Time Processes. *Real-Time Systems*, 3:67–99, 1991.

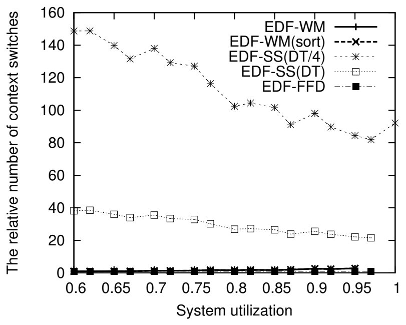
[6] T.P. Baker. Comparison of Empirical Success Rates of Global vs. Partitioned Fixed-Priority and EDF Scheduling for Hard Real Time. Technical report, Department of Computer Science, Florida State University, 2005.



(a) $m = 4$

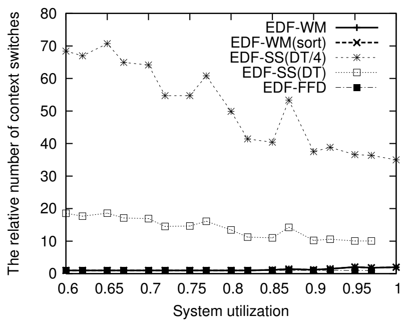


(b) $m = 8$

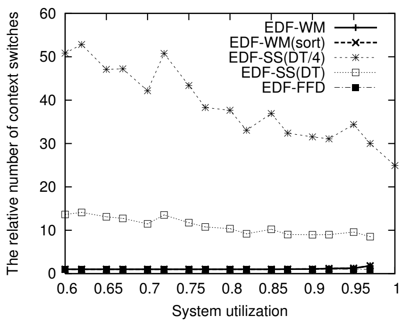


(c) $m = 16$

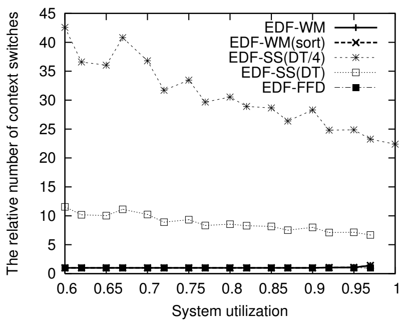
Figure 11. The worst-case number of context switches relative to EDF-FFD: $(u_{min}, u_{max}) = (0.5, 1.0)$**.**



(a) $m = 4$



(b) $m = 8$



(c) $m = 16$

Figure 12. The worst-case number of context switches relative to EDF-FFD: $(u_{min}, u_{max}) = (0.1, 0.5)$**.**

[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 and A. Mok. Preemptively Scheduling Hard-Real-Time Sporadic Tasks on One Processor. In *Proc. of the IEEE Real-Time Systems Symposium*, pages 182–190, 1990.

[9] G.C. Buttazzo. *HARD REAL-TIME COMPUTING SYSTEMS: Predictable Scheduling Algorithms and Applications, Second Edition*. Springer, 2005.

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

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

[12] N. Fisher. The Multiprocessor Real-Time Scheduling of General Task Systems. PhD thesis, Department of Computer Science, The University of North Carolina at Chapel Hill, 2007.

[13] 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.

[14] S. Kato and N. Yamasaki. Global EDF-based Scheduling with Efficient Priority Promotion. In *Proc. of the IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, pages 197–206, 2008.

[15] S. Kato and N. Yamasaki. Portioned EDF-based Scheduling on Multiprocessors. In *Proc. of the ACM International Conference on Embedded Software*, pages 139–148, 2008.

[16] S. Kato and N. Yamasaki. Semi-Partitioned Fixed-Priority Scheduling on Multiprocessors. In *Proc. of the IEEE Real-Time and Embedded Technology and Applications Symposium*, 2009.

[17] 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.

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

[19] C.A. Phillips, C. Stein, E. Torng, and J. Wein. Optimal Time-Critical Scheduling via Resource Augmentation. In *Proc. of the Annual ACM Symposium on Theory of Computing*, pages 140–149, 1997.

[20] M. Spuri and G.C. Buttazo. Scheduling Aperiodic Tasks in Dynamic Priority Systems. *Journal of Real-Time Systems*, 10:179–210, 1996.