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 \(\alpha\) is a maximum utilization of individual tasks \([18]\). Let \(\alpha = 1\) and \(m \to \infty\), 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 \([11]\) 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 \([12]\) 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 assigned to any individual processors, to strictly dominate the previous algorithms based on classical partitioned scheduling.
• 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.

EDF [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 $\delta$ 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.

Those algorithms mentioned above have concerns as follows. First, since tasks may migrate across processors, even though they can be assigned to individual processors, we are not sure that they are truly more effective than classical partitioned scheduling approaches. Then, such tasks may repeat migrations in and out of the same processor within the same period, which is likely to cause cache performance to degrade. The number of context switches is also problematic due to repetition of migrations. In addition, optional techniques for EDF, such as synchronization protocols [5] and aperiodic servers [20], may not be easily applicable, because 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
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.

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 \( P_1 \) or \( P_4 \), it is assigned to \( P_1 \) 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 [9] that will be used in the remainder of this section. The demand bound function \( dbf(T, L) \), given by Equation (1), computes the maximum amount of time, so-called processor demand, consumed by all jobs of a task \( T \) that have both release times and dead-

---

\(^1\)In fact, any partitioning methods are available under EDF-WM.
Given by Equation (4) and $H$ describes the details of the algorithm.

$s$ is the number of processors across which $T_k$ is split. Hence, after $d_k^* = d_k/s$ where $s$ is the number of processors across which $T_k$ is split. Hereafter, $d_k^* = d_k/s$ and $c_k' = c_k/s$, since they all have the same value. With such $d_k^*$, we determine $c_k'$ as the minimum of Equation (7) for $\forall L$ (1), (8) for $\forall L > 0$.

\[
\sum_{T \in \mathcal{T}} dbf(T_i, L) + dbf(T_k, L) \leq L \tag{6}
\]

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.

\[
dbf(T_k, L) = \max\left(0, \frac{L - d_{k,x}^*}{p_k}\right) + 1 c_{k,x}^* \tag{5}
\]

We now explain how to compute the values of $d_{k,x}^*$ and $c_{k,x}^*$. Let $\tau$ 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 $\forall L > 0$.

\[
\sum_{T \in \mathcal{T}} dbf(T_i, L) + dbf(T_k, L) \leq L \tag{6}
\]

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. Hence, $d_{k,x}^* = d_k/s$ for all $x$ is unified by $d_{k,x}^* = d_k/s$, since they all have the same value. With such $d_{k,x}^*$, we determine $c_{k,x}^*$ as the minimum of Equation (7) for $\forall L$ ($d_{k,x}^* \leq L$) aligned with deadlines of jobs. Note that those values of $L < d_{k,x}^*$ can be ignored, because the processor demand of $T_k$ is always zero.

\[
c_{k,x}^* = \frac{L - \sum_{T \in \mathcal{T}} dbf(T_i, L)}{\left\lfloor \frac{L - d_{k,x}^*}{p_k} \right\rfloor + 1} \tag{7}
\]

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 $d_{k,x}^*$. The other is $\sum_{T \in \mathcal{T}} dbf(T_i, L)$ 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 $d_{k,x}^* = d_k/s$. 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 $p_x$ denotes a set of $s$ processors under consideration, across which $T_k$ is split.

\[
\sum_{P_x \in \mathcal{P}} c_{k,x}^* \geq c_k \tag{8}
\]

- $T_k$ 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.
Note that when\( L \) by Equation (7), we recalculate Equation (9). Since \( \sum_{i \in \tau} c_{i,x} = c_{k,x} \), we split \( c_{k,x} \) monotonically increasing with respect to \( c \). The algorithm returns failure only when \( s \) exceeds \( m \) (line 2).

\[
L_a = \max \left( \frac{\sum_{i \in \tau_x} (p_i - d_i)u_i + (p_k - d'_{k,x})u'_{k,x}}{1 - \sum_{i \in \tau_x} u_i - u'_{k,x}} D \right)
\]  

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

\[
u'_{k,x} < 1 - \sum_{i \in \tau_x} u_i \Leftrightarrow c'_{k,x} < \left( 1 - \sum_{i \in \tau_x} u_i \right) p_k
\]  

Based on the discussion above, we first assume \( L_a = \mathcal{H} \) and \( c'_{k,x} = (1 - \sum_{i \in \tau_x} u_i)p_k \). 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 \( L_a \) 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, \texttt{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 \( P_k \). If \( \sum_{i \in \tau_x} c'_{k,x} \) does not reach \( c_k \), the same procedure is repeated with \( s = s + 1 \). Otherwise, \( T_k \) is assigned to each of \( P_k \in \pi_k \). Note that \( c'_{k,x} \) needs to be adjusted at line 13, since \( P_k \) is the last processor that is not necessarily filled to capacity by the last portion of \( T_k \). The algorithm returns failure only when \( s \) exceeds \( m \) (line 2).

\[
c'_{k,x} = (1 - \sum_{i \in \tau_x} u_i)p_k; L' = \mathcal{H};
\]

\[
\text{for each } T_i \in \tau_x \cup \{ T_k \} \text{ do } \alpha = [(d_i' - d_i)/p_i];
\]

\[
\text{while } L = \alpha p_i + d'_{k,x} < L' \text{ do } c = (\mathcal{L} - dbf(\tau_x, L))/((\mathcal{L} - d'_{k,x}/p_k) + 1);
\]

\[
\text{if } c < c'_{k,x} \text{ then } c'_x = c;
\]

\[
L' = \text{Equation (9), where } u'_{k,x} = c'_{k,x}/p_k;
\]

\[
\text{end if}
\]

\[
\alpha = \alpha + 1;
\]

\[
\text{end while}
\]

\[
\text{end for}
\]

\[
\text{return } c'_{k,x};
\]

Figure 5. Pseudo-code of \texttt{calc\_exec\_time}(T\_k, P\_x).

Due to limitation of space, we illustrate the function, \texttt{calc\_exec\_time}(T\_k, P\_x), associated with calculating \( c'_{k,x} \) (at line 5) in Figure 5. In the pseudo-code, \( \alpha \) is a natural number, and \( dbf(\tau_x, L) \) represents \( \sum_{i \in \tau_x} \mathcal{d}bf(T_i, L) \). 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 \( L = \alpha p_i + d'_{k,x} < L' \). The value of \( \alpha \) set at line 3 guarantees \( L \geq d'_{k,x} \). 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 \texttt{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.

\textbf{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 \( c'_{k,x}/d'_{k,x} = s c'_{k,x}/d_k \) 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 \( u'_{k,x} \). Given that the processor utilization achievable by EDF-WM depends on each \( u'_{k,x} \), task sorting also brings schedulability improvements to EDF-WM.
Due to limitation of space, we define $t$ as the timer execution time of a task $T_c$; the absolute deadline, and the remaining execution time of that task $T_k$. Let $T_c$ be a migratory task that is migrated from $P_s$ to $P_y$ by migrate($T_k, P_s, P_y$). It removes $T_k$ from $\gamma_y$ and inserts it to $\gamma_y$ (line 11). It then updates $\bar{a}_k, \bar{d}_k$, and $e_k$ (line 12). Since $T_k$ will have a pseudo-release time $\bar{a}_k$ on $P_y$, a timer function is set to invoke schedule($P_y$) at time $\bar{a}_k$. Although $T_k$ is now ready on $P_y$, it is never selected for execution before $\bar{a}_k$, 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($T_k, P_s$) and complete($T_k, P_s$) are called when $T_k$ is released and completes respectively. schedule($P_s$) is then called at the end of those functions, since we need to schedule a next task. Note that more than one release($T_k, P_s$) 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($P_s$) every time but call it once after all such release($T_k, P_s$) 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 $P_s$ and $P_y$ be the first processor and the last processor respectively: jobs of $T_i$ are released on $P_s$ and complete on $P_y$. We have $nc(T_i, L) \leq \lceil L/p_i \rceil$ for $P_s$, and $nc(T_i, L) = 0$ for any other processors. The number of preemptions is bounded as follows.
On the first processor $P_s$, $T_i$ may generate a preemption every time its job is released, and it also generates a preemption every time it consumes $c_i$ time units. Thus, we have $np(T_i, L) \leq 2\lceil L/p_i \rceil$. On the last processor $P_f$, $T_i$ 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 $P_x$, 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 $c_i$ 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 $T_i$.

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

$$ncs(L, P_x) \leq 2\sum_{T \in \mathcal{T}} \left\lceil \frac{L}{p_i} \right\rceil$$ (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 $\delta$ 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 $2\sum_{T \in \mathcal{T}} \left\lceil L/p_i \rceil$. 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 $\sum_{T \in \mathcal{T}} \left\lceil L/p_i \rceil$, 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
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 $m$, $u_{\text{min}}$, and $u_{\text{max}}$.

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 deadlines 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 deadlines, since the scheduling policy is subject 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 $m$, $u_{\text{min}}$, and $u_{\text{max}}$. This means that the value of $\delta$ 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. Given that EDF-WM(sort) is competitive to EDF-SS(DT) in terms of schedulability, its effectiveness is more obvious when the number of context switches is taken into account.
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 [8]. 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

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

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


