JCSE Journal of Computing Science and Engineering

Regular Paper

Journal of Computing Science and Engineering,
Vol. 14, No. 3, September 2020, pp. 121-130

Real-Time Scheduling for Periodic Tasks on Uniform Multiprocessors

Sang-Gil Lee and Cheol-Hoon Lee*

System Software Lab., Department of Computer Science and Engineering, Chungnam National University, Daejeon, Korea
sk0137@cnu.ac.kr, clee@cnu.ac.kr

Abstract

The problem of scheduling a set of periodic tasks on a uniform multiprocessor system is considered in the present study. Each processor in a uniform multiprocessor system is characterized by its speed or computation capacity, i.e., execution of a job on a processor with speed s for t time units completes $s \times t$ units of execution. In the commonly-known partitioned scheduling, each task is assigned to a processor and all of its jobs are required to be executed on that processor. However, partitioning of periodic tasks requires solving the bin-packing problem, which is known to be intractable (NP-hard in the strong sense). This paper presents a global scheduling algorithm that transforms a given periodic task system into another using a "task-splitting" (as opposed to the "set-splitting") technique. Each transformed periodic task system is guaranteed to be scheduled successfully on any uniform multiprocessor using a partitioned scheduling algorithm. The earliest deadline first (EDF) algorithm is chosen for scheduling tasks on each processor. It is proven that the proposed algorithm results in the theoretical-maximum utilization bound on any uniform multiprocessor platform if the platform is "reasonably powerful". Therefore, the proposed algorithm is optimal in the sense of maximizing achievable utilization. Since the task-splitting technique will incur context switches during runtime, we have also considered the ways of reducing the number of context switches, and suggest a method which can significantly reduce the number of context switches in the schedules generated by the proposed algorithm.

Category: Real-Time Systems

Keywords: Real-time scheduling; Real-time tasks; Task-splitting

I. INTRODUCTION

In hard real-time systems, there are certain basic units of work, known as *jobs*, which must be executed promptly. In multiprocessor systems, there are multiple processors on which the real-time jobs may be executed: all the processors operate at the same speed in a homogeneous multiprocessor, while different processors may operate at different speeds in an inhomogeneous multiprocessor.

A hard real-time system is specified by an instance of jobs with hard real-time requirements and the computing platform upon which the jobs are to be executed. A real-time **job** $j = (a, e, d)$ is characterized by three parameters, *arrival time a, execution requirement e*, and *deadline d*, with the interpretation that this job must receive e time units of execution over the interval $[a, d)$. A real-time ***instance*** J is a finite or infinite collection of jobs: $J = \{j_1, j_2, \cdots,\}$ Let $\tau = \{T_1, T_2, \cdots, T_n\}$ denote a **periodic task system**. Each periodic task T_i is completely characterized by a 4-tuple (a_i, e_i, d_i, p_i) with the *offset* a_i (which denotes

Open Access http://dx.doi.org/10.5626/JCSE.2020.14.3.121 http://jcse.kiise.org

This is an Open Access article distributed under the terms of the Creative Commons Attribution Non-Commercial License (http://creativecommons.org/licenses/by-nc/4.0/) which permits unrestricted non-commercial use, distribution, and reproduction in any medium, provided the original work is properly cited.

Received 07 July 2020; **Revised** 10 August 2020; **Accepted** 13 August 2020

*Corresponding Author

Copyright © 2020. The Korean Institute of Information Scientists and Engineers

pISSN: 1976-4677 eISSN: 2093-8020

the instant at which the first job generated by this task becomes available), the *execution requirement* e_i, the *relative deadline* d_i, and the *period* p_i. That is, $T_i = (a_i, e_i, d_i, p_i)$ generates an infinite succession of jobs, each with execution requirement e_i, at each instant $(a_i + k \cdot p_i)$ for all the integers $k \geq 0$, and the job generated at the instant $(a_i + k \cdot p_i)$ has a deadline at the instant $(a_i + k \cdot p_i + d_i)$. Unless stated otherwise, we assume that $d_i = p_i$. Periodic task systems are called *synchronous* if the offsets of all the tasks are identical (usually considered zero), else *asynchronous*. While the execution time may be any non-negative number, periods are assumed here to be non-negative integers. We define the *utilization* $U(T_i)$ by task T_i to be the ratio of its execution requirement to its period: $U(T_i) \stackrel{\text{def}}{=} e_i/p_i$. Without loss of generality, we assume that the tasks in τ are indexed according to non-increasing utilization: $U(T_i) \geq U(T_{i+1})$ for all, i, $1 \leq i < n$. For any periodic task system τ, $U_{\text{sum}}(\tau)$ will denote the cumulative utilization of all tasks in $\tau (U_{sum}(\tau) \stackrel{\text{def}}{=} \sum_{i=1}^{n} U(T_i))$ and $U_{\max}(\tau)$ will denote the largest utilization of any task in $\tau (U_{max}(\tau) \stackrel{\text{def}}{=} \sum_{i=1}^{n} U(T_i))$. We assume that each job is independent in the sense that it does not interact in any manner (accessing shared data, exchanging messages, etc.) with other jobs of the same or another task.

In this paper, we have considered the problem of scheduling periodic task systems upon a *uniform* multiprocessor platform comprising of m processors. Each processor is characterized by a single parameter denoting its speed or computation capacity. Therefore, a job that executes on a processor of computation capacity s for t time units completes $s \times t$ units of execution (Note that homogeneous multiprocessors are a special case of uniform multiprocessors, in which the computation capacities of all the processors are equal). We use the notation $\pi = [s_1, s_2, \cdots, s_m]$ to represent the uniform multiprocessor platform with m processors in which the processors have computation capacities $s_1, s_2, \cdots, s_m$ respectively. Without loss of generality, we assume that these speeds are indexed in a non-increasing order: $s_j \geq s_{j+1}$ for all j, $1 \leq j < m$. For any such uniform multiprocessor platform π, $S_{\text{sum}}(\pi)$ will denote the aggregate computation capacities such that $S_{\text{sum}}(\pi) \stackrel{\text{def}}{=} \sum_{i=1}^{j} s_i$.

Online scheduling algorithms make scheduling decisions at each time instant based upon the characteristics of the jobs that have arrived thus far, without any knowledge of jobs that may arrive in the future. Several online uniprocessor scheduling algorithms, such as the earliest-deadline-first (EDF) scheduling algorithm [1-4] and the least-laxity-first (LLF) algorithm [5, 6] are known to be optimal in the sense that if a set of jobs can be scheduled such that all jobs will be completed by their deadlines, so can these algorithms. However, for multiprocessor systems, no online scheduling algorithm can be optimal: this was shown for the simplest (homogeneous) multiprocessor model by Dertuozos and Mok [7], and Hong and Leung [8], and the technique in [7, 8] can be directly extended to the more general uniform machine model. An important advance in the study of online scheduling upon multiprocessors was made by Phillips et al. [9], who explored the use of *resource-augmentation* techniques for online scheduling of real-time jobs. They showed that if a real-time instance is feasible on m identical processors, then the same instance will be scheduled to meet all the deadlines by EDF on m processors in which the individual processors are $(2-\frac{1}{m})$ times as fast as in the original system. Given the specifications of a uniform multiprocessor platform π_0, Funk et al. [7] generalized the techniques in [9] to obtain a condition upon the specifications of any other uniform multiprocessor platform π_1 such that, if π_1 satisfies this condition, then any hard real-time task system feasible on π_0 will meet all the deadlines when scheduled on π_1 using EDF.

In this paper, we have considered the problem of scheduling periodic task systems on uniform multiprocessor platforms which permit job preemptions and migrations (i.e., a job executing on a processor may be interrupted and resumed later on the same or a different processor with no cost or penalty) and disallowing a job to be executed on more than one processor at any time.

It has been proven by Leung and Whitehead [11] that the partitioned and global approaches to static-priority scheduling on homogeneous multiprocessors are *incomparable*, implying that neither of the approached static-priority multiprocessor scheduling is strictly better than the other. In [12, 13], the rate-monotonic (RM) scheduling of periodic task systems on homogeneous multiprocessor platforms was studied. A *utilization bound* was derived such that any periodic task system with aggregate utilization no larger than this bound is guaranteed to be successfully scheduled by RM on a homogeneous multiprocessor platform. Baruah et al. [14] generalized the result for EDF scheduling on uniform multiprocessor platforms. The authors of [15] proved that the problem of optimally scheduling periodic tasks on a homogeneous multiprocessor could be solved in polynomial time using the PFair scheduling algorithm. Under PFair, each periodic task is executed at an (approximately) uniform rate (corresponding to its utilization factor) by breaking it into a series of quantum-length *subtasks*.

In this paper, we present a global scheduling algorithm that transforms a given periodic task system into another using a "task-splitting" technique. Each transformed periodic task system is guaranteed to be successfully scheduled on any uniform multiprocessor using a partitioned scheduling algorithm. The EDF algorithm is of our choice for scheduling tasks on each processor. It is proven that, for any uniform multiprocessor platform π, the proposed algorithm achieves the utilization bound of $S_{\text{sum}}(\pi)$, which is theoretically maximum, if the platform is "reasonably powerful". Therefore, the proposed algorithm

is *optimal* in terms of achievable utilization. Since task-splitting will incur context switches during runtime, it is important to reduce the number of such context switches. Consequently, we have considered the way to reduce the number of context switches, and suggest a method which can greatly reduce the number of context switches on the schedules generated by the algorithm.

For an n independent non-real-time task system, Gonzalez and Sahni [16] suggested an $O(n)$ time algorithm to obtain an optimal finish time preemptive schedule on m uniform processors. This algorithm can be applied to a real-time task system by splitting every task into subtasks with the common deadline of unit time. This would lead to an unacceptably large number of migrations—as many as $(n \times m)$ migrations every time unit—and is hence not a realistic approach to real-time scheduling of periodic task systems. As can be seen later, most $(m-1)$ of the tasks are split in our algorithm thereby resulting in much fewer migrations.

The rest of this paper is organized as follows. In Section II, we review some results concerning EDF scheduling on multiprocessor platforms. Subsequently, the task-splitting technique is presented. In Section III, we present a global scheduling algorithm that transforms a given periodic task system into another using the task-splitting technique. It is proven that each transformed periodic task system is guaranteed to be successfully scheduled upon any uniform multiprocessor using EDF if the utilization of the system is not greater than $S_{\text{sum}}(\pi)$. In Section IV, we suggest a method that can significantly reduce the number of context switches on the schedules generated by the proposed algorithm. The paper concludes with Section V.

II. BACKGROUND

We first briefly review results on EDF scheduling on multiprocessor platforms. Then, we describe the task-splitting technique which is essential for our proposed approach.

Under EDF scheduling, jobs are assigned priorities that are inversely proportional to their deadlines, i.e., the earlier the deadline, the higher the priority. EDF is known to be optimal on uniprocessors; that is, if any periodic task system can be correctly scheduled on a given preemptive uniprocessor by *any* scheduling algorithm, the EDF will correctly schedule this task system on that processor. Unfortunately, EDF is *not* optimal on multiprocessors considering the same logic. There are, nevertheless, significant advantages to use EDF for scheduling jobs on multiprocessors, if possible. Accordingly, the EDF scheduling of periodic task systems on multiprocessor platforms has recently attracted considerable attention (e.g., [17], [18], [19], [10]).

For multiprocessor scheduling using the partitioned approach, it has been shown that the utilization bound cannot exceed $\left(\frac{m+1}{2}\right)$ on m identical processors using *fixed-priority* scheduling; if the largest utilization $U_{\max}(\tau)$ by any task in τ is known; hence, a somewhat better bound of $(\frac{\beta m+1}{\beta+1})$ was proven by Lopez et al. [17], where $\beta = \lfloor 1/U_{\max}(\tau) \rfloor$. Goossens et al. [18] proved that the periodic task system τ is scheduled to meet all the deadlines by EDF on m identical processors, provided $U_{\text{sum}}(\tau) \le m-(m-1)U_{\max}(\tau)$. Using this result, Baruah [19] proposed a fixed-priority scheduling algorithm to be used for the global scheduling of periodic task systems with a schedulable utilization equal to $(m+1)/2$ on m identical multiprocessors. The author also proved that no fixed-priority scheduling algorithm can have a schedulable utilization greater than $(m+1)/2$ upon m identical processors. Given the specifications of a uniform multiprocessor platform π_0, Funk et al. [7] derived a sufficient condition upon the specifications of any other uniform multiprocessor platform π_1 such that, if π_1 satisfies this condition, then any hard real-time task system feasible on π_0 will meet all the deadlines when scheduled on π_1 using EDF. As a corollary to their results, we can obtain the result of [9] concerning EDF-scheduling on homogeneous multiprocessors: if a set of jobs is feasible on a homogeneous m-processor platform, then the same set of jobs will be scheduled to meet all the deadlines by EDF on a homogeneous m-processor platform in which the individual processors are $(2-\frac{1}{m})$ times as fast as in the original system.

The above results deal with utilization-based conditions for determining whether a given system τ of periodic tasks is successfully scheduled on any specified multiprocessor system using the EDF scheduling algorithm. In this paper, we assume that jobs are scheduled according to the EDF$^+$ policy which is the same as EDF, except that, among jobs whose deadlines are the same, the one with the latest arrival time has the highest priority (LIFO policy); in case both deadline and arrival times are equal, the job with the lowest index has the highest priority. This EDF$^+$ *priority ordering* is essential in our approach because it provides a total priority order. For a given periodic task system, we can get a much better utilization bound under the EDF$^+$ scheduling policy by transforming it into another periodic task system. A periodic task system τ_1 can be transformed into τ_2 without compromising with the feasibility as formalized by the following definition:

DEFINITION 1 (transformable). *For a given multiprocessor platform π, if a periodic task system τ_1 is* **transformable** *into τ_2 (denoted as $\tau_1 \xrightarrow{\pi} \tau_2$) and if τ_2 can be correctly scheduled by EDF$^+$ on π, then τ_1 will also be correctly scheduled on π.*

For example, consider a multiprocessor platform $\pi = [1, 1]$ and the following two-periodic task systems:

$$\tau_1 = \{T_1, T_2, T_2\}$$
$$\equiv \{(0, 1.6, 2, 2), (0, 0.6, 1, 1), (0, 1, 2, 2)\} \text{ and}$$
$$\tau_2 = \{T_1', T_2', T_3', T_4'\}$$
$$\equiv \{(0, 1.6, 2, 2), (0, 0.6, 1, 1), (0, 0.2, 0.2, 1),$$
$$(0.7, 0.3, 0.3, 1)\}.$$

If we assign T_1' and T_3' to one processor and the other two tasks (T_2' and T_4') to another processor, and run EDF$^+$ on each processor, then τ_2 can be feasibly scheduled as shown in Fig. 1(a). Comparing τ_1 and τ_2, we see that $T_1 \equiv T_1'$ and $T_2 \equiv T_2'$. Only T_3 is split into T_3' and T_4'. Note that T_3' and T_4' do not execute simultaneously at any time instant, and they all execute for $2 (= e_3)$ time units over each interval $[2 \cdot k,\ 2 \cdot (k+1)) (= [a_3 + k \cdot p_3,\ a_3 + k \cdot p_3 + d_3))$ for all integer $k \geq 0$. Thus, by mapping T_1, T_2, and T_3 into T_1', T_2' and $T_3' + T_4'$, respectively, τ_1 can also be feasibly scheduled on π as shown in Fig. 1(b). Therefore, $\tau_1 \xrightarrow{\pi} \tau_2$.

Intuitively, as shown in the above example, we can transform periodic task systems by "splitting" some task(s) as formalized by the following definition:

DEFINITION 2 (splitting). *A task $T_i = (a_i, e_i, d_i, p_i)$ is said to be* "**split**" *into j tasks $T_{i_k} = (a_{i_k}, e_{i_k}, d_{i_k}, p_{i_k})$, $1 \leq k \leq j$, if the following properties satisfy for all k:*

P1: $\boldsymbol{p_{i_k} = 1}$,

P2: $\boldsymbol{d_{i_k} = e_{i_k} / s_{i_k}}$

P3: $\boldsymbol{a_{i_k} + d_{i_k} \leq a_{i_{k+1}} \leq 1 - d_{i_{k+1}}}$, ***and***

P4: $\boldsymbol{\sum_{k=1}^{j} U(T_{i_k}) \equiv \sum_{k=1}^{j} e_{i_k} = U(T_i)}$

where each task T_{i_k} is assigned to the processor s_{i_k}.

LEMMA 1 ($\tau_1 \xrightarrow{\pi} \tau_2$). *For any periodic task system τ_1, let τ_2 be the resulting periodic task system by splitting some task(s) in τ_1. Then $\tau_1 \xrightarrow{\pi} \tau_2$ for any multiprocessor platform π.*

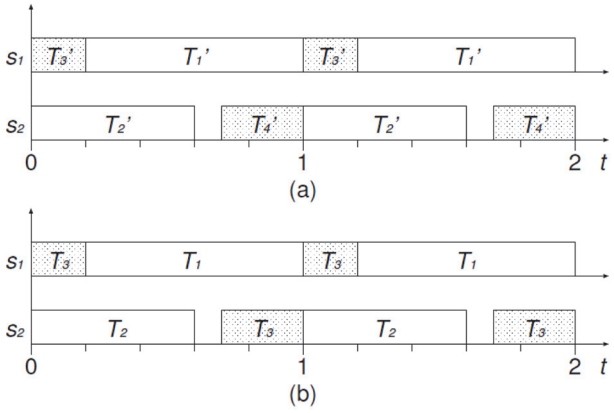

Fig. 1. Example schedules: (a) EDF schedule of τ_2, and (b) the corresponding schedule for τ_1 by mapping T_1, T_2, and T_3 into T'_1, T'_2 and $T'_3 + T'_4$, respectively.

Proof. For a given multiprocessor platform π, let us suppose that τ_2 is feasibly scheduled on π by EDF$^+$ Let task $T_i = (a_i, e_i, d_i, p_i)$ in τ_1 be split into j tasks $T_{i_k} = (a_{i_k}, e_{i_k}, d_{i_k}, p_{i_k})$, $1 \leq k \leq j$, where each split task T_{i_k} is assigned to the processor s_{i_k}, respectively. Then, each task T_{i_k} executes for exactly d_{i_k} time units to complete $e_{i_k} (= s_{i_k} \times d_{i_k})$ units of execution. Therefore, by P3, no two of the j tasks T_{i_k}'s split from the task T_i execute simultaneously at any instant in time on the schedule, and they execute a total of e_i time units over each interval $[a_i + l \cdot p_i,\ a_i + l \cdot p_i + d_i)$ for all integer $l \geq 0$. Thus, by mapping T_i into the j tasks T_{i_k}'s, T_i can also be feasibly scheduled on π. Therefore, $\tau_1 \xrightarrow{\pi} \tau_2$.

As shown in the above example, a periodic task system that cannot be feasibly scheduled by EDF can be feasibly scheduled after being transformed into another task system. Therefore, a much better utilization bound for EDF can be obtained using the task-splitting technique. In the rest of this paper, we assume that input periodic task systems are synchronous and every task has its relative deadline equal to its period. We also assume that the uniform multiprocessor platform is "*reasonably powerful*" in the sense that the i-th fastest processor can solely execute the i-th *heaviest* task (*i.e.,* the task with the i-th largest utilization) without missing any deadline. In other words, the processors are assumed to satisfy the following condition:

Condition 1: $s_i \geq U(T_i), \ \forall i,\ 1 \leq i \leq m$.

If periodic task systems satisfy this condition, we can get the utilization bound of $S_{\text{sum}}(\pi)$, which is the theoretical maximum, for any uniform multiprocessor platform π. The complete algorithm is described in the following section.

III. THE ALGORITHM

The optimality of EDF on uniprocessors suggest a correlation between bin-packing and partitioned scheduling since a periodic task system τ can be partitioned onto a uniform multiprocessor platform π if and only if the tasks can be divided into disjoint subsets $\tau_1, \tau_2, \cdots, \tau_m$ such that $\sum_{T_j \in \tau_i} U(T_j) \leq s_i$. Task partitioning on homogeneous multiprocessors corresponds to the bin-packing problem with all the bins of the same size. Johnson [20] proved that this problem is NP-complete in the strong sense. In this section, however, we show that any periodic task system with the total utilization no greater than the theoretical maximum (i.e., $S_{\text{sum}}(\pi)$) is transformable into a periodic task system which can be partitioned onto a uniform multiprocessor platform π.

To completely describe our proposed algorithm for scheduling periodic task systems on a uniform multiprocessor platform, we must specify two separate algorithms: (1) the **pre-assignment** algorithm that determines to which processor each task should be assigned, without splitting, and (2) the **task-splitting** algorithm that splits each remaining task and assigns the split tasks to the processors. We consider first-fit-decreasing (FFD) as the task pre-assignment heuristic which sorts the tasks and assigns them to processors in a weakly-decreasing order of utilization. A task can be assigned to the processor s_i if its utilization is not greater than $(s_i - U_i)$, where U_i is the total utilization of tasks already assigned to the processor; in this case, we say the task *fits* on the processor. Algorithm 1 shows the FFD task pre-assignment algorithm in which each task is assigned to the fastest processor upon which it will fit. The variable **gap**(j) denotes the remaining capacity available on the process or s_i and **rem** denotes the set of tasks that cannot be assigned to any processor. According to the FFD task pre-assignment algorithm, each task T_i is assigned to the processor s_j, where j is the smallest-indexed processor with **gap**(j) at least as large as $U(T_i)$, the utilization of T_i. If FFD attempts to assign the task T_i to a processor and all the processors' gaps are smaller than $U(T_i)$, then T_i goes into the set **rem**. Each of these unassigned tasks is split and assigned to multiple processors by the task-splitting algorithm described later. The run-time computational complexity of FFD task pre-assignment is $\mathcal{O}(n\log n)$ (for sorting the tasks in non-increasing order of utilization) + $\mathcal{O}(m\log m)$ (for sorting the processors in non-increasing order of capacities) + $\mathcal{O}(n \times m)$ (for doing the actual assignment of tasks to processors), for an overall computational complexity of $\mathcal{O}(n \cdot (\log n + m))$ assuming that the number of processors does not exceed the number of tasks.

Algorithm 1. The FFD pre-assignment algorithm

FFD task pre-assignment (τ, π)

Let $\tau = \{T_1, T_2, \cdots, T_n\}$ denote the tasks,
with $U(T_i) \geq U(T_{i+1})$ for all i
Let $\pi = [s_1, s_2, \cdots, s_m]$ denote the processors,
with $s_j \geq s_{j+1}$ for all j

1: for $j \leftarrow 1$ to m do $gap(j) \coloneqq s_j$
2: $rem \coloneqq \emptyset$

3: for $i \leftarrow 1$ to n do
4: Let j_0 denote the smallest index such that
$gap(j_0) \geq U(T_i)$
5: If no such j_0 exists, then $rem \coloneqq rem \cap T_i$; break
6: Assign T_i to processor j_0
7: $gap(j_0) \coloneqq gap(j_0) - U(T_i)$
8: od

If every task is assigned by the FFD pre-assignment algorithm (i.e., if the resulting **rem** is empty), there is no need to split any task. In this case, the periodic task system can be feasibly scheduled by EDF without any transformation. When the resulting **rem** is not empty, each task in rem should be split and assigned to multiple processors. However, there is a limit on the number of tasks in **rem** (i.e., ||**rem**||) as shown in the following lemma:

Lemma 2. *If* $U_{\text{sum}}(\tau) \leq S_{\text{sum}}(\tau)$, then ||**rem**|| < m.

Proof. Suppose that ||**rem**|| ≥ m. According to the FFD pre-assignment algorithm, **gap**(j) < $U(T_i)$, $1 \leq j \leq m$, $T_i \in$ **rem**. That is, $\sum_{T_i \in rem} U(T_i) > \sum_{1 \leq j \leq m}$**gap**($j$). Let U_a be the sum of the utilizations of all the tasks assigned by the FFD pre-assignment algorithm. Then,

$$\begin{aligned} U_{\text{sum}}(\tau) &= U_a + \sum_{T_i \in \text{rem}} U(T_i) \\ &> U_a + \sum_{1 \leq j \leq m} \text{gap}(j) \\ &= S_{\text{sum}}(\pi) \end{aligned}$$

This is a contradiction. Thus, the lemma follows.

Lemma 3. *If* $U_{\text{sum}}(\tau) \leq S_{\text{sum}}(\pi)$, *then*

$$U(T_i) \leq \frac{S_{\text{sum}}(\pi)}{m+1}, \quad \forall i, T_i \in rem.$$

Proof. Let U_a be the sum of the utilizations of all the tasks assigned by FFD. Also, let $u_{\max}$ be $\max_{T_i \in rem} U(T_i)$. Then, we get

$$U_{\text{sum}}(\tau) \geq U_a + u_{\max}. \tag{1}$$

According to **Condition 1**, m heaviest tasks are guaranteed to be assigned. Thus,

$$U_a \geq m \times u_{\max}. \tag{2}$$

From Eqs. (1) and (2), we get

$$u_{\max} \leq \frac{U_{\text{sum}}(\tau)}{m+1} \leq \frac{S_{\text{sum}}(\pi)}{m+1}.$$

Therefore, the lemma follows.

Let u' be the utilization of the $(m+1)$-th heaviest task. Then, $u_{\max} \leq u'$, since m heaviest tasks are guaranteed to be assigned. Therefore, the following corollary holds.

Corollary 1. $U(T_i) \leq s_j$, $\forall j$, $1 \leq j \leq m$, $T_i \in$ ***rem***.

Since the remaining capacity of each processor is less than the utilization of any task in ***rem***, we get the following corollary from Lemma 3.

COROLLARY 2. *If* $U_{\text{sum}}(\tau) \le S_{\text{sum}}(\pi)$, *then*

$$gap(j) < \frac{S_{\text{sum}}(\pi)}{m+1},\ \forall j,\ 1 \le j \le m.$$

LEMMA 4. *If* $\|rem\| > 0$, *then*

$$gap(j) < \frac{s_j}{2},\ \forall j,\ 1 \le j \le m.$$

Proof. Let $T_i \in \mathbf{rem}$. Then, by Corollary 1, $U(T_i) \le s_j$, $\forall j,\ 1 \le j \le m$. Therefore, for each processor s_j, there must be at least one task T_k assigned to it with $U(T_k) \ge U(T_i)$, since, if not, T_i should have been assigned before those assigned to s_j. Let U_j be the sum of the utilizations of all the tasks assigned to s_j. Then,

$$\begin{aligned} s_j &= U_i + \text{gap}(j) \\ &\ge U(T_i) + \text{gap}(j) \\ &> 2 \times \text{gap}(j). \end{aligned}$$

Therefore, the lemma follows.

We are now ready to describe the task-splitting algorithm. The algorithm sorts the tasks in *rem* and splits them in weakly-decreasing order of utilization. It also sorts the processors in non-increasing order of remaining capacities. Algorithm 2 shows the task-splitting algorithm in which each task is split into smaller ones such that each of which (except the last one) is with the utilization equal to the remaining capacity of the corresponding processor and is assigned to the processor. The variable *left*(*i*) denotes the amount of remaining utilization of T_i' after splitting. After splitting, the offset increases accordingly for the next split task (see Line 9 of the algorithm) to guarantee that no two split tasks execute simultaneously at any instance of time on the schedule. Note that, to reduce the number of context switches, the offset of the last split task is shifted right to the end of integer boundaries (see Line 13 of the algorithm). The run-time computational complexity of the algorithm is $\mathcal{O}(k \log k)$ (for sorting the tasks in *rem* in non-increasing order of utilization) + $\mathcal{O}(m \log m)$ (for sorting the processors in non-increasing order of remaining capacities) + $\mathcal{O}(k \times m)$ (for doing the actual splitting of tasks), for the overall computational complexity of $\mathcal{O}(m^2)$ since $k < m$ by Lemma 2.

Each task in *rem* is split into more than two tasks since its utilization is greater than the remaining capacity of any processor. Then, by the mechanism to make new tasks (see the inner loop of the algorithm), it is obvious that all the properties in Definition 2 are embraced. Let task T_i' be split into j tasks T_{i_k}', $1 \le k \le j$. Then, by the algorithm, they are assigned to j processors s_p', $l \le p \le l+j-1$, one-by-one, in non-increasing order of the remaining capacities. Let T_{i+1}' the next task be split into

Algorithm 2. The task-splitting algorithm

Task Splitting (rem, π)

Let $rem = \{T_1', T_2', \cdots, T_k'\}$ denote the tasks,
with $U(T_i') \ge U(T_{i+1}')$ for all i
Let $\pi = [s_1', s_2', \cdots, s_m']$ denote the processors,
with $gap(j) \ge gap(j+1)$ for all j.

1: for $i \leftarrow 1$ to k do $left(j) \coloneqq U(T'_j)$
2: $p \coloneqq 1$

3: for $i \leftarrow 1$ to k do
4: a:= 0; q:= 1
5: for $j \leftarrow p$ to m do
6: If $left(i) \ge gap(p)$, then
7: $e \coloneqq gap(p); gap(p) \coloneqq 0; d \coloneqq e/s'_p$
8: Make a new task $T'_{i_q} = (a, e, d, 1)$ and assign it to processor p; $q \coloneqq q+1$
9: $a \coloneqq a + d$; $p \coloneqq p+1$
10: $left(i) \coloneqq left(i) - e$
11: If $left(i) = 0$, then break
12: else
13: $e \coloneqq left(i); d \coloneqq e/s'_p; a \coloneqq 1-d; left(i) \coloneqq 0$
14: Make a new task $T'_{i_q} = \{a, e, d, 1\}$ and assign it to processor p.
15: $gap(p) \coloneqq gap(p) - e$; break
16: od
17: od

j' tasks $T_{(i+1)_k}'$, $1 \le k \le j'$. Then, in the same way, they are assigned to j' processors s_p', $l' \le p \le l'+j'-1$. Based on the algorithm, $l' = j$ or $l' = j+1$. Therefore, we can get the following corollary:

COROLLARY 3. *At most two split tasks can be assigned to each processor.*

According to Lemma 4, less than half of the capacity is used for the execution of split tasks on each processor. Let task T_i' be split into j tasks T_{i_k}', $1 \le k \le j$, and assigned to j processors s_p', $l \le p \le l+j-1$, respectively. If every split task is feasibly scheduled on each processor, each T_i' executes exactly during $[a_{i_k}, a_{i_k}+d_{i_k})$ as stated in the proof of Lemma 1. Then, no one executes across integer boundaries, nor do any two splits from the same task execute simultaneously at any instant of time. This is formally described and proved in the following lemma.

LEMMA 5. *Let task T_i' be split into j tasks T_i', $1 \le k \le j$, then the followings satisfy:*

$E1$: $a_{i_1} = 0$,

$E2$: $a_{i_k} + d_{i_k} = a_{i_{k+1}}$, $\forall k$, $1 \le k \le j-1$, and

$E3$: $a_{i_{j-1}} + d_{i_{j-1}} \le a_{i_j} = 1 - d_{i_j}$.

Proof. **E1** and **E2** are obvious from the algorithm. We prove only **E3** here. The assumption that **E3** does not hold means that $\sum_{k=1}^{j} d_{i_k} > 1$. Let tasks T_{i_k}''s, $1 \le k \le j$, be assigned to processors s_p''s, $l \le p \le l+j-1$, respectively. Then,

$$1 < \sum_{k=1}^{j} d_{i_k} = \frac{e_{i_1}'}{s_l'} + \frac{e_{i_2}'}{s_{l+1}'} + \cdots + \frac{e_{i_j}'}{s_{l+j-1}'}.$$

Let $s_{\min}' = \min_{p=l}^{l+j-1} s_p'$. Then, we get

$$1 < \sum_{k=1}^{j} d_{i_k} \le \frac{\sum_{k=1}^{l} e_{i_k}'}{s_{\min}'} = \frac{U(T_i')}{s_{\min}'}. \quad (3)$$

By Corollary 1, $U(T_i') \le s_j'$, $\forall j$, $T_i' \in \text{rem}$. This contradicts Eq. (3). Therefore, the lemma follows.

Let $\mathcal{T}_i$ be the set of tasks (including split tasks) assigned to the processor s_i. According to the algorithm, the processor index (the variable p) is incremented if and only if the remaining capacity of the current processor becomes zero. Therefore, if $U_{\text{sum}}(\tau) \le S_{\text{sum}}(\pi)$, then $\sum_{T_j \in \mathcal{T}_i} U(T_j) = s_i$, $\forall i$. Obviously, if $U_{\text{sum}}(\tau) = S_{\text{sum}}(\pi)$, then $\sum_{T_j \in \mathcal{T}_i} U(T_j) = s_i$, $\forall i$. Therefore, if $U_{\text{sum}}(\tau) \le S_{\text{sum}}(\pi)$, the task set $\mathcal{T}_i$ on each processor s_i can be feasibly scheduled by **EDF**$^+$, as proven in the following theorem.

THEOREM 1. If $U_{\text{sum}}(\tau) \le S_{\text{sum}}(\pi)$, then the task set $\mathcal{T}_i$ on each processor s_i can be feasibly scheduled by **EDF**$^+$.

Proof. By Corollary 3, there are at most two split tasks in $\mathcal{T}_i$. For each split task $T_{i_k} = (a_{i_k}, e_{i_k}, d_{i_k}, 1)$, modify it to $T_{i_k}' = (0, e_{i_k}, 1, 1)$. Then, the resulting task set $\mathcal{T}_i'$ is synchronous and every task has its deadline parameter equal to its period. (Note that we assume each task of input periodic task system (τ) satisfies these properties.) Since $U_{\text{sum}}(\mathcal{T}_i') \le s_i$, EDF can feasibly schedule $\mathcal{T}_i'$. Let the schedule by EDF be EDF.$\mathcal{T}_i'$. In the schedule EDF.$\mathcal{T}_i'$, each split task $T_{i_k}' = (0, e_{i_k}, 1, 1)$ executes for e_{ik} time units during every time interval $(j, j+1]$ for all the integers $j \ge 1$, without being preempted during execution (Note that the periods of input tasks are integers). The deadline of each non-split task, say T_o, which executes in every time interval $(j, j+1]$, is an integer l ($\ge j+1$). Now, we can obtain a new schedule S' from EDF.$\mathcal{T}_i'$ by shifting the execution duration of each split task T_{i_k} to $[j+a_{i_k}, j+a_{i_k}+d_{i_k})$ during every time interval $[j, j+1)$ for all the integer $j \ge 1$. There is no change in the total amount of execution units for T_o over the time interval $[j, j+1)$ on the new schedule. Consequently, T_o still meets its deadline on the new schedule. Therefore, the schedule S' is also feasible since no task misses its deadline on it. Note that the priority-ordering policy on the schedule S' is the same as **EDF**$^+$. Since the task priorities are totally ordered under **EDF**$^+$, **EDF**$^+$ generates a unique schedule for any given task set. S' is the schedule for the task set $\mathcal{T}_i$ under **EDF**$^+$. Therefore, if $U_{\text{sum}}(\tau) \le S_{\text{sum}}(\pi)$, then the task set $\mathcal{T}_i$ on each processor s_i is feasibly scheduled by **EDF**$^+$.

The task set assigned to each processor by our approach (the FFD task pre-assignment and the task-splitting algorithms) can be feasibly scheduled on the processor by the **EDF**$^+$ algorithm. Also, by Lemma 5, it was shown that no two split tasks of the same task execute simultaneously at any instant of time. Therefore, for a given uniform multiprocessor platform π, any periodic task system τ can be feasibly scheduled on the platform *if* and *only if* the total utilization $U_{\text{sum}}(\tau)$ is not greater than the total sum of processor capacities $S_{\text{sum}}(\pi)$.

THEOREM 2. For a given uniform multiprocessor platform π, any periodic task system τ can be feasibly scheduled on the platform *if* and *only if*.

$$U_{\text{sum}}(\tau) \le S_{\text{sum}}(\pi).$$

Proof. Let τ' be the resulting periodic task system by splitting some task(s) in τ. Then, by Theorem 1 and Lemma 5, τ' can be feasibly scheduled on any uniform multiprocessor platform π if $U_{\text{sum}}(\tau) \le S_{\text{sum}}(\pi)$. Therefore, by Lemma 1, $\tau \overset{\pi}{\rightarrow} \tau'$ means that τ can also be feasibly scheduled on π. Moreover, $S_{\text{sum}}(\pi)$ is the theoretic maximum bound for utilization on π. Therefore, the theorem follows.

Theorem 2 confirms the optimality of the proposed algorithm from the perspective of achievable utilization. However, our task-splitting algorithm produces schedules with a large number of context switches. In the next section, we consider this issue and suggest a method that can significantly reduce the number of context switches on the schedules generated by the proposed algorithm.

IV. PRACTICAL CONSIDERATIONS

The proposed algorithm is proven to successfully schedule a periodic task system on any uniform multiprocessor platform, as long as $U_{\text{sum}}(\tau) \le S_{\text{sum}}(\pi)$, by using the task-splitting technique. However, task-splitting incurs a large number of context switches during runtime. We now consider ways of reducing the number of context switches and suggest a method that can significantly reduce the

number of context switches on the schedules generated by the proposed algorithm.

By the task-splitting algorithm in Algorithm 2, a task T_i is split into j tasks T_{i_k}, $1 \le k \le j$, each with the period of 1 (Note that even though the tasks resulting from splitting always have a period of 1, input tasks have any integer period). Let these tasks be assigned to processors s_{i_k}'s. Then, each split task T_{i_k} on processor s_{i_k} executes exactly once during every time interval $[j, j+1)$, for all integer $j \ge 0$. Consequently, these j tasks T_{i_k}'s to incur j context switches on the schedule $\mathcal{S}$ generated by the proposed algorithm, for every time interval $[j, j+1)$, for all integer $j \ge 0$. However, by packing two (or more) job instances of each split task, we can reduce the number of context switches. Let $g = (g.s, g.e, g.t, g.p)$ be a job instance of a split task $T_{g.t}$ which executes for the time interval $[g.s, g.e)$ on processor $s_{g.p}$. Then, two job instances g_i and g_j, $g_i.s < g_j.s$, of the same task on the same processor (*i.e.*, $g_i.t = g_j.t$ and $g_i.p = g_j.p$) can be packed into a single job instance $g_k = (g_i.s,\ g_i.e + g_i.e - g_j.s,\ g_i.t,\ g_i.p)$ if the following conditions are met:

C1: Every job instance of the same or different tasks during the time interval $[g_i.e,\ g_j.s)$ should still meet its deadline after the packing.

C2: Each job instance g_l delayed by the packing should not overlap in time with any job instance of the same task $T_{g_l.t}$ on different processors.

C3: Any job instance of the same task $T_{g_i.t}$ on different processors should not overlap in time with the packed job instance.

In this paper, we propose a packing algorithm which is very simple, but significantly reduces the number of context switches. The packing algorithm called *Pack-Forward* is shown in Algorithm 3, where $\mathcal{S}$ is the schedule generated by our scheduling algorithm and $\mathcal{G}$ is the list of job instances of all split tasks. The algorithm sorts the job instances in $\mathcal{G}$ in increasing order of start times. From the first job instance, it tries to pack two job instances of the same task on the same processor into a single job instance. The function *packable*($\mathcal{S}$, g_i, g_j) is a function that returns YES if g_i and g_j satisfy the above three conditions on the schedule $\mathcal{S}$, else returns NO. If the two job instances are packable (see Line 7 in Algorithm 3), they are packed into a single job instance and adjust the schedule $\mathcal{S}$ using the function *pack*(). For each periodic task system $\tau = \{T_1, T_2, \cdots, T_n\}$, let $\tau.\mathcal{H}$ be the hyperperiod of τ, *i.e.*, $\tau.\mathcal{H} = LCM(p_1, p_2, \cdots, p_n)$, where LCM stands for "Least Common Multiple". We can only consider the job instances for the time interval $[0, \tau.\mathcal{H})$. For example, consider the example schedule in Fig. 1. In this example, $\tau.\mathcal{H} = 2$. The resulting schedule $\mathcal{S}'$ after packing with the algorithm *PackForward* is shown in Fig. 2(b). Note that g_1, $\cdots$, g_4 in Fig. 2(a) are the job instances of the split task T_3. In this example, g_3 and g_4 are packed with g_1 and g_2, respectively. Comparing the two schedules, we can see that the number of context switches is reduced from 8 to 5 during each hyperperiod. Note that every job on the packed schedule $\mathcal{S}'$ still follows the EDF$^+$ priority ordering.

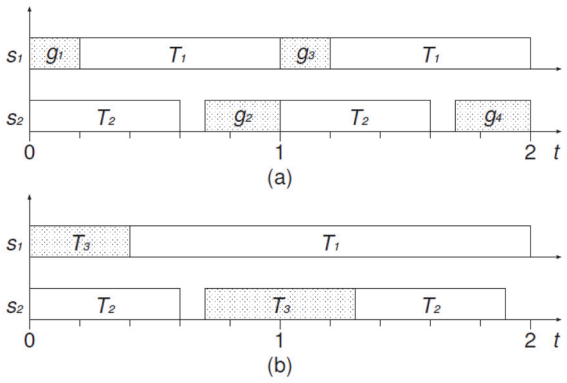

Fig. 2. Example schedules: (a) the schedule $\mathcal{S}$ generated by our scheduling algorithm for the example periodic task system in Fig. 1 and (b) the resulting packed schedule $\mathcal{S}'$.

Algorithm 3. A simple packing algorithm

PackForward $(\mathcal{S}, \mathcal{G})$

Let $\mathcal{S}$ be the schedule generated by the proposed algorithm
Let $\mathcal{G} = \{g_1, g_2, \cdots, g_k\}$ denote the job instances of split tasks, with $g_i.s \le g_{i+1}.s$ for all i

1: for $j \leftarrow 1$ to k to $used(j) :=$ NO

2: for $i \leftarrow 1$ to k do
3: If $used(j) =$ YES, continue
4: for $j \leftarrow i+1$ to k do
5: If $g_i.p \ne g_j.p$ or $g_i.t \ne g_j.t$, continue
6: Else if $packable(\mathcal{S}, g_i, g_j) \ne$ YES, break
7: Else $\mathcal{S} := pack(\mathcal{S}, g_i, g_j)$; $used(j) :=$ YES
8: od
9: od

For each periodic task system $\tau = \{T_1, T_2, \cdots, T_n\}$, let $\tau.\mathcal{C} = \sum_{i=1}^{n}(\tau.\mathcal{H}/p_i)$. Then, no algorithm can produce a feasible schedule with fewer context switches than $\tau.\mathcal{C}$ during each hyperperiod. Thus, in this paper, $\tau.\mathcal{C}$ is used as the theoretical lower bound on the optimal number of context switches for each periodic task system τ. For the above example, $\tau.\mathcal{C} = 4$. note that the optimal number of context switches for this example periodic task system is 5. Let the *estimation error* be "optimal number of context switches" $-\tau.\mathcal{C}$. For this example, the estimation error is 1. Let $\mathcal{S}_N$ be the total number of job instances of split

tasks in schedule $\mathcal{S}$. In this paper, we have used the term "packing efficiency" as a performance metric that is defined as follows:

$$\text{Packing efficiency} = \frac{\mathcal{S}_{\mathcal{N}} - \mathcal{S}'_{\mathcal{N}}}{\mathcal{S}_{\mathcal{N}}},$$

where $\mathcal{S}$ and $\mathcal{S}'$ are the schedules before and after packing, respectively.

To evaluate the performance of the proposed scheduling (along with the packing algorithm) concerning the number of context switches generated, we have performed some simulations. Periodic real-time task sets are generated randomly such that the periods and utilization of tasks are uniformly distributed in the ranges [1, 100] and [0.0, 1.0], respectively. The computing capacities of all the processors are equal to 1, *i.e.*, $s_i = 1$, $1 \le i \le m$. Finally, the computation requirements of the periodic tasks are chosen such that $U_{\text{sum}}(\tau) = m$. We have performed simulations while varying the number of processors. Fig. 3 shows the packing efficiency of the suggested packing algorithm with 16, 32, 64, and 128 tasks, where each data is the average value after generating 100 random task sets. Fig. 3 shows a very high packing efficiency when the number of processors is small (higher than 80% with 4 or fewer processors). However, as the number of processors increases, the efficiency deteriorates. This is because, as the number of processors increases, two job instances are less likely packable due to the conditions $C2$ and $C3$. However, in all our simulations, the packing efficiency is higher than 50% over the entire ranges of the number of processors and tasks. Fig. 4 shows the context switch overhead normalized concerning the theoretical lower bounds. As can be seen in Fig. 4, the proposed algorithm generates much fewer context switches as the number of processors (tasks) decreases (increases). Especially, a large number of processors show a much larger overhead. This is because, as the number of processors increases, it is more likely that any two job instances are not packable due to the conditions $C2$ and $C3$ and the estimation error becomes much larger.

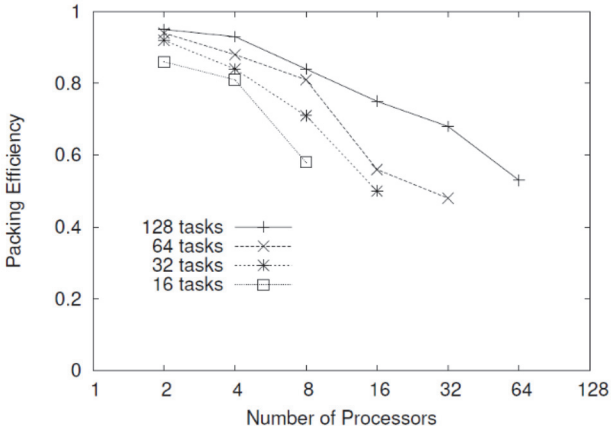

Fig. 3. Packing efficiency.

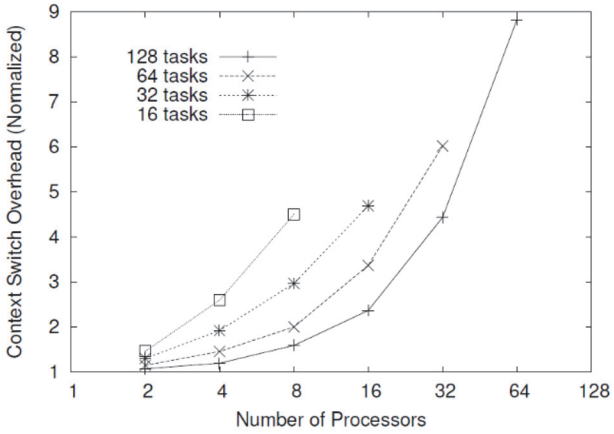

Fig. 4. Normalized context switch overhead.

V. CONCLUSIONS

We have presented a new global scheduling algorithm for scheduling periodic task systems on uniform multiprocessor platforms. This problem has been considered to solve the (variable-size) bin-packing problem, which is known to be intractable. However, we have proven that this is not the case in the context of jobs, which can be preempted at any time and resumed later, by presenting a task-splitting technique. It has been proven that the proposed algorithm successfully schedules any periodic task system on reasonably powerful uniform multiprocessor platforms if $U_{\text{sum}}(\tau) \le S_{\text{sum}}(\pi)$ by applying the EDF$^+$ scheduling algorithm on each processor. The proposed algorithm is optimal in the sense of maximizing achievable utilization.

We have also considered the practical issue of reducing the number of context switches and proposed a very simple method that can significantly reduce the number of context switches on the schedules generated by the proposed algorithm. Nevertheless, the resulting schedule may incur a large number of preemptions. It would be interesting to investigate algorithms for solving the periodic scheduling problem while minimizing the number of preemptions. This is an issue for our future investigation.

One may also argue that, due to semantic reasons, a task cannot be split arbitrarily into two tasks. The proposed task-splitting approach will still work, albeit sub-optimally, if we allow each processor to have a small slack (unused cycles).

ACKNOWLEDGMENTS

This work was supported by the research fund of Chungnam National University.

REFERENCES

1. M. Dertouzos, "Control robotics: the procedural control of physical processes," in *Proceedings of the 6th IFIP Congress*, Stockholm, Sweden, 1974, pp. 807-813.
2. C. L. Liu and J. W. Layland, "Scheduling algorithms for multiprogramming in a hard-real-time environment," *Journal of the ACM*, vol. 20, no. 1, pp. 46-61, 1973.
3. J. H. Anderson, V. Bud, and U. C. Devi, "An EDF-based scheduling algorithm for multiprocessor soft real-time systems," in *Proceedings of the 17th Euromicro Conference on Real-Time Systems (ECRTS)*, Balearic Islands, Spain, 2005, pp. 199-208.
4. F. Zhang and A. Burns, "Schedulability analysis for real-time systems with EDF scheduling," *IEEE Transactions on Computers*, vol. 58, no. 9, pp. 1250-1258, 2009.
5. A. K. L. Mok, "Fundamental design problems of distributed systems for the hard-real-time environment," Ph.D. dissertation, Massachusetts Institute of Technology, Cambridge, MA, 1983.
6. J. Lee, A. Easwaran, and I. Shin, "LLF schedulability analysis on multiprocessor platforms," in *Proceedings of 2010 31st IEEE Real-Time Systems Symposium*, San Diego, CA, 2010, pp. 25-36.
7. M. L. Dertouzos and A. K. Mok, "Multiprocessor online scheduling of hard-real-time tasks," *IEEE Transactions on Software Engineering*, vol. 15, no. 12, pp. 1497-1506, 1989.
8. K. S. Hong and J. T. Leung, "On-line scheduling of real-time tasks," in *Proceedings of the 9th IEEE Real-Time Systems Symposium*, Huntsville, AL, 1988, pp. 244-250.
9. C. A. Phillips, C. Stein, E. Torng, and J. Wein, "Optimal time-critical scheduling via resource augmentation," in *Proceedings of the 29th Annual ACM Symposium on Theory of Computing*, El Paso, TX, 1997, pp. 140-149.
10. S. Funk, J. Goossens, and S. Baruah, "On-line scheduling on uniform multiprocessors," in *Proceedings 22nd IEEE Real-Time Systems Symposium (RTSS 2001) (Cat. No. 01PR1420)*, London, UK, 2001, pp. 183-192.
11. J. Y. T. Leung and J. Whitehead, "On the complexity of fixed-priority scheduling of periodic, real-time tasks," *Performance Evaluation*, vol. 2, no. 4, pp. 237-250, 1982.
12. B. Andersson, S. Baruah, and J. Jonsson, "Static-priority scheduling on multiprocessors," in *Proceedings 22nd IEEE Real-Time Systems Symposium (RTSS 2001) (Cat. No. 01PR1420)*, London, UK, 2001, pp. 193-202.
13. S. K. Baruah and J. Goossens, "Rate-monotonic scheduling on uniform multiprocessors," *IEEE Transactions on Computers*, vol. 52, no. 7, pp. 966-970, 2003.
14. S. Baruah, S. Funk, and J. Goossens, "Robustness results concerning EDF scheduling upon uniform multiprocessors," *IEEE Transactions on Computers*, vol. 52, no. 9, pp. 1185-1195, 2003.
15. S. K. Baruah, N. K. Cohen, C. G. Plaxton, and D. A. Varvel, "Proportionate progress: a notion of fairness in resource allocation," *Algorithmica*, vol. 15, pp. 600-625, 1996.
16. T. Gonzalez and S. Sahni, "Preemptive scheduling of uniform processor systems," *Journal of the ACM*, vol. 25, no. 1, pp. 92-101, 1978.
17. J. M. Lopez, M. Garcia, J. L. Diaz, and D. F. Garcia, "Worst-case utilization bound for EDF scheduling on real-time multiprocessor systems," in *Proceedings 12th Euromicro Conference on Real-Time Systems (Euromicro RTS)*, Stockholm, Sweden, 2000, pp. 25-33.
18. J. Goossens, S. Funk, and S. Baruah, "Priority-driven scheduling of periodic task systems on multiprocessors," *Real-Time Systems*, vol. 25, no. 2-3, pp. 187-205, 2003.
19. S. K. Baruah, "Optimal utilization bounds for the fixed-priority scheduling of periodic task systems on identical multiprocessors," *IEEE Transactions on Computers*, vol. 53, no. 6, pp. 781-784, 2004.
20. D. S. Johnson, "Fast algorithms for bin packing," *Journal of Computer and System Sciences*, vol. 8, no. 3, pp. 272-314, 1974.

Sang-Gil Lee https://orcid.org/0000-0002-7642-4518

Sang-Gil Lee is Ph.D. candidate of Computer Science and Engineering at Chungnam National University, South Korea. He received the M.S. degree in Computer Science from Chungnam National University, Daejeon, South Korea in 2016. His research interests are in real-time performance for GPOS, system software, and embedded systems.

Cheol-Hoon Lee https://orcid.org/0000-0002-1926-0115

Cheol-Hoon Lee received M.S degree in Electrical and Electronic Engineering from the Korea Advanced Institute of Science and Technology (KAIST), Korea in 1988, and Ph.D. degree in Electrical and Electronic Engineering from KAIST, Korea in 1992. He is a professor in the Department of Computer Science and Engineering at Chungnam National University, Daejeon, South Korea. He joined the College of Engineering, Chungnam National University of Korea in March 1995. His research interests are in real-time systems, real-time performance for GPOS (Windows, Linux), system software, and secure OS for embedded computing environments.