<table>
<thead>
<tr>
<th><strong>Title</strong></th>
<th>Flow network-based real-time scheduling for reducing static energy consumption on multiprocessors</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Author(s)</strong></td>
<td>Sun, Joohyung; Cho, Hyeonjoong; Easwaran, Arvind; Park, Ju-Derk; Choi, Byeong-Cheol</td>
</tr>
<tr>
<td><strong>Date</strong></td>
<td>2018</td>
</tr>
<tr>
<td><strong>URL</strong></td>
<td><a href="http://hdl.handle.net/10220/48703">http://hdl.handle.net/10220/48703</a></td>
</tr>
<tr>
<td><strong>Rights</strong></td>
<td>© 2018 IEEE. Translations and content mining are permitted for academic research only. Personal use is also permitted, but republication/redistribution requires IEEE permission. See <a href="http://www.ieee.org/publications_standards/publications/rights/index.html">http://www.ieee.org/publications_standards/publications/rights/index.html</a> for more information.</td>
</tr>
</tbody>
</table>
Flow Network-Based Real-Time Scheduling for Reducing Static Energy Consumption on Multiprocessors

JOOHYUNG SUN1, HYEONJOONG CHO1, ARVIND EASWARAN2, (Member, IEEE), JU-DERK PARK3, AND BYEONG-CHEOL CHOI3

1Department of Computer Convergence Software, Korea University, Sejong 30019, South Korea
2School of Computer Science and Engineering, Nanyang Technological University, Singapore 639798
3Hyper-Connected Communication Research Laboratory, ETRI, Daejeon 34129, South Korea

Corresponding author: Hyeonjoong Cho (raycho@korea.ac.kr)

This work was supported in part by the National Research Foundation of Korea (NRF) through the Basic Science Research programs under Grant NRF-2015R1D1A1A01057018 and Grant NRF-2018R1D1A1B07049078 (Ministry of Education) and in part by the Ministry of Education, Singapore, under Tier-2 Grant ARC9/14.

ABSTRACT

The energy management for embedded real-time systems is crucial due to their restricted power supplies. With the advancement of technologies, the static energy consumption of the embedded systems that is caused by their leakage power is growing. Thus, a number of research works have started focusing on reducing the static energy consumption by making the systems transit into low-power states, wherein some hardware components are temporarily shut down. Specifically, when a processor is idling, they attempt to set the processor into one of several low-power states. To make a processor remain in the low-power state as long as possible to minimize the energy consumption, the idle time should be maximally clustered. At the same time, in order to satisfy the real-time constraints of the tasks, the length of the clustered idle time should be estimated accurately. To achieve our objective, we propose energy-efficient real-time scheduling algorithms on symmetric homogeneous multiprocessors with a dynamic power management scheme for periodic real-time tasks. The proposed algorithms rely on a flow network model that effectively helps to cluster the idle time while respecting the real-time constraints. In our experimental evaluation, the proposed algorithms consume a comparable static energy to an existing off-line scheme that is the only suitable existing algorithm in the problem domain. Furthermore, we show that the proposed algorithms consume less static energy than the existing one in a case where the total workload of the given task set is low and the task completion is earlier than expected.

INDEX TERMS

Dynamic power management, energy-aware algorithm, flow network problem, multiprocessor unit, micro-controller unit, real-time scheduling.

I. INTRODUCTION

Embedded real-time systems are now prevalent due to the technology over the last several decades. Since their power supplies such as batteries are limited, it has become critical to efficiently manage their energy consumption. For the energy management, the two schemes, dynamic voltage/frequency scaling (DVFS) and dynamic power management (DPM), are commonly used. DVFS is a power management scheme that adjusts the supply voltage of processor units to reduce their dynamic energy consumption. The dynamic energy is consumed by running tasks on the active processors in which the capacitance of the complementary metal-oxide semiconductor (CMOS) transistors is charged or discharged. On the other hand, DPM is a power management scheme that sets the processors to low-power states. Since the CMOS transistors are continuously powered-on even on the inactive processors, some amount of leakage power always occurs. To reduce the leakage power consumption (or static power consumption), DPM transits processors into low-power states by disabling some of the processor parts. In the past, the dynamic energy consumption was dominant over the static power consumption. Since the dynamic energy $E$ is proportional to the square of the supply voltage $V_{dd}$ and clock frequency $f$, i.e., $E \propto V_{dd}^2 \times f$ in [1], DVFS-based
algorithms are focused on adjusting $V_{dd}$ or $f$ to save energy [2]–[4], [6], [7]. Nowadays, the static power consumption is no longer negligible compared with the dynamic power consumption. According to [14], the operational voltage ($V_{opp}$) has been scaled down at the historical rate of 30% per CMOS technology generation in order to keep power dissipation and power delivery costs under control. It implies that the advantage of DVFS has decreased since the lowered operational voltage has reduced the portion of the dynamic power consumptions. Additionally, the threshold voltage ($V_{th}$) that is the minimum voltage needed to create a conducting path between the source and drain terminals of a transistor has also been reduced at the same rate. Since the static power consumption is proportional to $1/V_{th}^2$ [14], the static power consumption has become increasing over the dynamic power consumption.

To reduce the static power consumption, several DPM-based algorithms have been introduced [5], [6], [15]–[18]. They switch a processor into one of low-power states when they are not in use. It is known that the deeper low-power state consumes less power than the shallower low-power state. Therefore, when the processor is idling, an improved energy saving can be achieved by putting the idle processors into a deeper low-power state as long as possible. This finding implies that the processor idle time should be collected as much as possible to ensure that the processors remain in one of the low-power states during the collected idle time interval. However, the transitions between different states are not free, since the waking up of the processors from the low-power states requires time and energy. As stated in [32], the challenge is to predict the length of idle time with high accuracy in the presence of multiple low-power states so that the most energy efficient low-power state can be selected for the idle processor. It also implies that the length of the collected idle time should be estimated accurately; for example, if the length is estimated shorter than the actual value, a less-efficient energy saving occurs. Alternatively, if a longer length is estimated, the task deadline can be missed.

The DPM-based algorithms, for example, can be used in determining the minimum number of active processors for scheduling a set of tasks and shutting down the unused processors. It is extremely energy-efficient because it disconnects the supply voltage from the unused processors, which results the dynamic and static energy consumption into zero. In order to use the processor shutdown technique, the scheduling algorithm should be able to compute the minimum number of required processors.

In this paper, we propose the scheduling algorithm mainly focusing on three issues regarding the saving of static energy using real-time scheduling with DPM scheme, as follows:

- clustering the idle times as long as possible,
- estimating the clustered idle time accurately, and
- meeting all of the implicit-deadlines of periodic tasks, i.e., achieving RT-optimality.

**Definition 1:** (RT-optimal) An optimal real-time schedule meets all of the task deadlines when the total utilization demand $U$ of a given task set does not exceed the total processing capacity $M$, which is the RT-optimal in this study.

**Contributions:** In this study, we focus on clustering the idle time distributed over all the processors in order to make a processor stay in a low-power state as long as possible without jeopardizing the real-time task scheduling. The contributions of this study are summarized as follows.

We propose a flow network-based scheduling algorithm, flow network-based DPM (fnDPM), that is for executing periodic real-time tasks on homogeneous multiprocessors. fnDPM relies on DPM to save the static energy. Since there exists a trade-off between energy-efficiency and time-complexity, we propose two approaches, the fnDPM with fine-grained windows (fnDPM-fw) and the fnDPM with coarse-grained windows (fnDPM-cw). fnDPM-fw uses the fine-grained time windows to generate the longer idle time than that of the fnDPM-cw, which results in high computational complexity. Alternatively, the fnDPM-cw uses the coarse-grained time windows to focus more on the reduction of its time complexity than the fnDPM-fw.

Experimental evaluations of the proposed algorithms were conducted in comparison with the latest existing algorithm, LPDPM, in terms of both the static energy consumption and the state-transition overheads. The reason why LPDPM was selected as the counterpart is that it supports the RT-optimality with consideration of the multiple low-power states, as summarized in Table 1. The experiment results show that the proposed online scheduling algorithms, fnDPMs, save comparable static energy with the offline scheduling algorithm, LPDPM. Especially, in cases with the low processor-utilization demand of the given task set and the
earlier completion of the tasks than expected, both fnDPMs save more static energy than LPDPM.

Organization: The remainder of this paper is organized as follows. Section II introduces the related works. Section III defines our models. Section IV and V describe the flow network problem, the procedures of the fnDPM algorithms, and the complexity of the algorithms. Section VI presents the experimental evaluations with their results. Section VII and VIII describe future works, discussion, and conclusion.

II. RELATED WORKS
Langen and Juurlink [8] showed that when the static power dissipation becomes more significant, employing the maximum number of processors to maximize the amount of slack that can be used to lower the supply voltage is no longer beneficial from an energy perspective. Therefore, they proposed a leakage-aware multiprocessor scheduling algorithm (LAMPS) for determining minimum number of processors which consume minimum energy for a set of periodic tasks. Their algorithm sets upper and lower bounds on the number of processors and applies binary search approach to find the minimum number of processors. However, they did not apply the shutdown of processors or the procrastination of job execution for idle processors.

Chen et al. [9] proposed an energy-efficient scheduling algorithm for periodic real-time tasks on homogeneous multiprocessor environment in which the leakage current was non-negligible. They sorted all tasks in a non-increasing order of their workloads by the largest task first (LTF) strategy and then, the sorted tasks were assigned on the processors according to the first fit (FF) strategy to minimize the number of processors in use. In addition, they applied the online procrastination algorithm for turning the processor into the dormant mode (shutdown mode) by delaying the arrival time of the next job in order to reduce the static energy consumption. However, they did not consider the possible overhead by assuming that the processor in the dormant mode consumed zero power and waking up the processors from the dormant mode took no time.

Awan and Petters [16] proposed an approach, called enhanced race-to-halt (ERTH), to save energy for a single processor supporting multiple low-power states. Their technique uses an offline analysis to compute the break-even time for each mode, where the break-even time is the minimum time interval required for a processor to stay in the low-power state in order to save energy. In addition, they accumulated the additional slack time generated by early completion of high-priority tasks to save extra static energy by allowing the processor to stay in the low-power mode for a longer time.

Bhatti et al. [15] presented a DPM strategy for real-time multiprocessor systems called assertive dynamic power management (AsDPM). AsDPM determines the number of active processors to satisfy the execution requirement of released tasks at runtime. Then, with Global-EDF or Global-LLF scheduling policies, AsDPM extracts and clusters the idle times that are distributed across the processors. However, neither Global-EDF nor Global-LLF is not RT-optimal, which results AsDPM in being non-optimal.

Awan [21] proposed a leakage-aware energy management algorithm, which is called global scheduler and power management (GPM), for a system using the Global-EDF scheduler on a homogeneous multicore platform. In order to save the static energy consumption, they utilize two types of slack time that are defined as usable execution slack and usable idle slack, which are obtained from the early completion of jobs and the residual capacity from a non-fully loaded platform, respectively. The GPM exploits these usable slack types to either put a core into a sleep state, or to prolong the sleep interval of the cores. This algorithm was evaluated on a simulator modeled after a Freescale PowerQUICC III-based multicore platform with several low-power states.

Legout et al. [17] proposed an offline approach called linear programming dynamic power management (LPDPM). LPDPM generates a feasible schedule on multiprocessors using the linear programming problem formulated with several constraints and the objective function of the minimization of the static energy consumption. It encourages the neighboring idle times to be merged as much as possible while considering the characteristics of the low-power states. The formulation takes all of the possible scheduling events into account by considering the hyper-period of the given tasks, where the hyper-period is defined as the least common multiple of all of the periods of the given tasks. After this offline procedure, LPDPM runs real-time tasks in each time interval between two adjacent scheduling invocations using the fixed-priority until zero laxity (FPZL) [19]. FPZL helps collecting the idle times that are generated when a task is completed earlier than expected. LPDPM is RT-optimal since its schedule is computed using the linear programming problem formulated for the hyper-period. However, the offline procedure-based LPDPM is not applicable to dynamic scenarios where the given task set changes at runtime when a new task can be added dynamically. Furthermore, when the specification of the low-power states for a system changes, the offline LP formulation of LPDPM must be recomputed even if the given task set is the same.

Nair et al. [18] proposed an energy-efficient proportional fair scheduling algorithm to reduce the static energy consumption in multiprocessors called by early-release fair scheduler with suspension on multiprocessors (ESSM). ESSM is based on the early-release fair (ERfair) scheduling algorithm, i.e., a variant of the proportional fair (Pfair) scheduling approach [20], the first RT-optimal scheduling algorithm for multiprocessors. ESSM uses a procrastination scheme that postpones the execution of tasks for keeping the processor in a low-power state to maximize the duration of

---

1 According to [18], the RT-optimality of LPDPM combined with FPZL is arguable since FPZL is not RT-optimal for generic periodic tasks having arbitrary period lengths.
idle time. However, since ESSM obtains the slack time from early released tasks for the procrastination scheme within adjacent scheduling points called quantum, this algorithm maximizes the idle time only locally while keeping fairness. In addition, ESSM assumes a single low-power state only and its extension for multiple low-power states has not been introduced yet.

III. SYSTEM MODEL
   A. TASK MODEL
A set $\Gamma$ that contains $N$ periodic real-time tasks, denoted by $\Gamma = \{\tau_1, \tau_2, ..., \tau_N\}$ where the tasks are mutually independent, was considered. The task $\tau_i$ has its period $T_i$ and worst-case execution time (WCET) $C_i$. It was assumed that the relative deadline $D_i$ of $\tau_i$ is equal to $T_i$, i.e., each $\tau_i$ has the implicit deadline. The utilization $u_i$ of $\tau_i$ is defined as $C_i/T_i$ and the total utilization $U$ is the sum of $u_i$. The active job (i.e., instance) of $\tau_i$ at time $t$, denoted by $\tau_i(t)$, has its arrival time $a_i(t)$ subject to $t \in [a_i(t), a_i(t) + T_i)$, the absolute deadline $d_i(t)$, and the remaining execution time $c_i(t)$. $d_i^{\text{max}}(t)$ is defined as the largest $d_i(t)$ of all of the active jobs at the time $t$.

A set $\mathcal{B}$ that contains the current time $t$, all of the release times, and the absolute deadlines of jobs within the time interval $[t, d_i^{\text{max}}(t)]$ is defined, i.e., $\mathcal{B} = \{b_0, b_1, ..., b_K\}$. $b_j$ is called temporal boundary and it is assumed that $\mathcal{B}$ is sorted in an increasing order. The time interval $[b_k, b_{k+1}]$ is called the time window $W_k$. The total number of windows is $K$. The length of the window $W_k$ is denoted by $l_k = b_{k+1} - b_k$.

B. PROCESSOR MODEL
A system with symmetric and homogeneous multiple processors was assumed for this study. Accordingly, a set of $M$ processors is denoted by $\Pi = \{\pi_1, \pi_2, ..., \pi_M\}$. It was assumed that $M - 1 < U$ and $U < M$. If $U < M - 1$, then $M - \lfloor U \rfloor$ processors can be easily turned off to save energy. Each processor $\pi_j$ has a same set of low-power states and they can be transitioned from the active state to the $p$-th low-power state by disabling some parts of the chip. The low-power state is characterized using the following parameters:

- The power consumption in the $p$-th low-power state is denoted by $PC_p$.
- The time needed to wake up from the $p$-th low-power state is denoted by $WT_p$.
- The power overhead for the wake-up from the $p$-th low-power state is denoted by $PP_p$.
- Break-even time of the $p$-th low-power state is denoted by $BET_p$ according to [16].

It is known that the deeper low-power states consume less energy but causes the higher state-transition overheads. It was assumed that the deepest low-power state that was indexed with $L$ consumes the lowest power and its wake-up time is the longest one, i.e., $PC_1 > PC_2 > ... > PC_L$ and $0 < WT_1 < WT_2 < ... < WT_L$. Generally, the wake-up time from a low-power state is much longer than the time required to enter the low-power state [4]. Therefore, it was assumed that $BET_p$ is equal to $WT_p$. A more detailed explanation about the break-even time can be also found in [4] and [21].

IV. ENERGY-AWARE REAL-TIME SCHEDULER
   A. PRELIMINARY
To achieve the RT-optimality, several classes of real-time task scheduling algorithms on multiprocessors have been developed for periodic implicit-deadline tasks. One of the well-known classes is fluid schedule-based algorithms in [20] and [22], where each task execution attempts to track the fluid schedule that is known to be RT-optimal. The fluid schedule-based algorithms attempt to guarantee RT-optimal by referring to the ideal fluid schedule and allocating the fractional processing capacities to the tasks at each boundary. Several examples of the fluid schedule-based algorithm such as PD [20] and BF [23] have been introduced and their scheduling strategies are providing each task $\tau_i$ with the proper amount of computational capacity that is proportional to its utilization $u_i$ at every boundary. In this sense, the fluid-schedule-based algorithms are said to support fairness. To minimize the static energy consumption, however, the execution of some tasks should be advanced or delayed against the constraint of the fairness to generate a long idle time. Notably, several studies have focused on the unfairness without losing the RT-optimality, as is the case in [24].

To achieve our scheduling objective, we attempt to transform the real-time scheduling problem into a network-flow problem. The formulation of a network flow problem or a linear programming problem for real-time scheduling is not new [17], [30]. However, since the previous works formulated the problem in consideration of a very long time interval from 0 to hyper-period of the given task set, they have been used as the offline scheduling techniques. Instead, for the formulation of the problem in this study, only the active jobs at the current boundary were considered, which enables us to design online scheduling algorithms. In addition, the algorithms are no longer restricted by the fluid schedule notion, which means that they become unfair-but-optimal scheduling algorithms with DPM. In next subsection, our problem formulation is explained and the proof of RT-optimality is described in [34] in detail.

B. PROBLEM FORMULATION
At every boundary, the scheduling algorithm invokes to reserve the execution time for all active jobs. We here formulate it as an optimization problem. For the convenience of description, two sets are defined for the current time interval as follows:

$$K(t_s, t_e) = \{k|W_k \subset [t_s, t_e]\},$$

$$J(k) = \{i|W_k \subset [a_i(t), d_i(t)]\}.$$  

$K(t_s, t_e)$ contains all of the indices $k$ of the window $W_k$ that are placed in the time interval $[t_s, t_e]$. $J(k, t_s)$ contains all of the indices $i$ of the active jobs at time $t_s$ which are still active in $W_k$. 
Using these two sets, a flow network problem for scheduling real-time tasks is formulated as follows.

\[
\text{Maximize} \sum_{i} \sum_{k} X_{i,k} \quad (3)
\]

\[
\text{s.t.} \sum_{k \in D(i)} X_{i,k} \leq c_i(t), \quad 1 \leq i \leq N \quad (4)
\]

\[
\sum_{i \in D(k)} X_{i,k} \leq \text{Cap}(W_k), \quad 1 \leq k \leq K \quad (5)
\]

\[
X_{i,k} \leq l_k, \quad 1 \leq i \leq N \text{ and } 1 \leq k \leq K. \quad (6)
\]

For equation 5, \(\text{Cap}(W_k)\) is \(W_k\)'s processing capacity that is required for executing the active jobs and it is set as follows:

\[
\text{Cap}(W_k) = \left[ M - \sum_{i \in D(k)} C_i/T_i \right] \times l_k. \quad (7)
\]

Here, the **active job area** at time \(t\), denoted by \(AJA(t)\), is defined to be a collection of the maximum processing capacity per window that can be utilized to execute the active jobs at \(t\). In \(AJA(t)\), \(X_{i,k}\) is the reserved execution time for \(\tau_i\) within \(W_k\). At a boundary, the corresponding AJA is established and three types of constraints are defined as equations 4-6. The first constraint is to complete the execution of each active job within the permitted time interval, i.e., \([a_i(t), d_i(t)]\), which is called the **job completion constraint (JCC)**. The second constraint represents that the sum of the active job execution times within a given window does not exceed the permitted processing capacity of the window, which is called the **processing capacity constraint (PCC)**. The third constraint represents that each active job within a given window does not simultaneously occupy more than one processor, which is called **no intra-task parallelism (NIP)**. After a feasible solution is found for AJA with these constraints, the part of the feasible solution for \(W_1\) is used to allocate the computational resource to each task for their execution during \(W_1\). At the next boundary, the next AJA is established and a similar procedure follows.

The formulation based on equations 3-6 constructs a flow network that is represented by a directed and capacitated graph \(G = (V, E)\). \(G\) contains a set of nodes \(V\) and a set of edges \(E\), as follows.

\[
V = \{n_1, n_e\} \cup \{\tau_i|\forall i\} \cup \{W_k|\forall k\} \quad (8)
\]

\[
E = \{e(n_1, \tau_i)|\forall i\} \cup \{e(\tau_i, W_k)|\forall i, k\} \cup \{e(W_k, n_e)|\forall k\} \quad (9)
\]

The nodes are named after the tasks \(\tau_i\) and the windows \(W_k\). \(n_1\) and \(n_e\) denote the added source node and the sink node, respectively. \(e(n_1, \tau_i)\) denotes the edge from node \(n_1\) to node \(n_2\) and the actual flow \(f(n_1, n_2)\) is assumed to be sent along the edge. The amount of the maximum flow from \(n_1\) to \(n_e\) is supposed to be \(\sum_i c_i(t)\).

At each boundary \(t\), AJA(t) and its corresponding flow network are repeatedly constructed. If the maximum flow \(\sum_i c_i(t)\) is found, it is interpreted as a feasible schedule that has been found for AJA(t). In addition, the flow on each edge is interpreted as the reserved execution time for the feasible schedule. Specifically, the flow from node \(\tau_i\) to node \(W_1\) is interpreted as \(X_{i,1}\) that is the reserved execution time of \(\tau_i\) within \(W_1\). Then, they way of allocating \(X_{i,1}\) to the processors within \(W_1\) can easily be determined, e.g., using McNaughtons wrap around algorithm in [25].

**V. FLOW NETWORK-BASED DYNAMIC POWER MANAGEMENT ALGORITHM**

To save more static energy using DPM, the processor needs to stay in the deeper low-power state as long time as possible. At the same time, estimation of the length of the clustered idle time in consideration of the overhead should be precise to avoid missing the deadlines of the real-time tasks. Additionally, the proposed algorithm should be computationally tractable to be used online. To achieve these goals, the formulation described in Section IV-B is considered.

To represent the idle time in the flow network, a virtual task called an **idle task** is added. Since the total capacity of the idle time is \((M - U) \times d_{max}^t\) in \([t, d_{max}^t]\), it is assumed that the idle task has its execution time \((M - U) \times d_{max}^t\) and its period \(d_{max}^t\). The additional idle task does not jeopardize the schedulability of the given task set because the total utilization \(U\) of the combination of the given tasks and the idle task always becomes equal to \(M\). Figure 1 shows the AJA(0) of the task set in Table 2.

![FIGURE 1. Active job area for the tasks in Table 2 at the time 0.](image)

**TABLE 2.** An example of the real-time task set including a virtual idle task.

<table>
<thead>
<tr>
<th>Tasks</th>
<th>WCET</th>
<th>Period</th>
<th>Utilization</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\tau_1)</td>
<td>3</td>
<td>4</td>
<td>0.75</td>
</tr>
<tr>
<td>(\tau_2)</td>
<td>3</td>
<td>4</td>
<td>0.75</td>
</tr>
<tr>
<td>(\tau_3)</td>
<td>4</td>
<td>6</td>
<td>0.666667</td>
</tr>
<tr>
<td>(\tau_4)</td>
<td>5</td>
<td>8</td>
<td>0.625</td>
</tr>
<tr>
<td>(\tau_{idle})</td>
<td>15</td>
<td>24</td>
<td>0.583333</td>
</tr>
</tbody>
</table>

**A. FLOW CONTROL OVER THE FLOW NETWORK**

The objective of a maximum flow algorithm is to send the maximal flow from the source to the sink. In general, multiple maximum flows that achieve the goal could exist, thereby implying that the formulated problem could have multiple solutions. The maximum flow algorithms arbitrarily find one of the feasible solutions. However, to generate the long idle time, it is necessary to collect the flow of idle task across the boundaries while maintaining the feasibility. This implies
that we should be able to prioritize certain flow over the flow network, which is not possible with the use of the simple maximum flow algorithms. Therefore, to control the flow, a parameter \( w(n_1, n_2) \) was assigned to the edge \( e(n_1, n_2) \) and the minimum cost flow algorithms [26], [27] were used. The minimum cost flow algorithms search for the flow that minimizes the total cost of flows that are calculated by \( \sum_{e(n_1, n_2)} w(n_1, n_2) f(n_1, n_2) \), when the amount of the maximum flow is given. Since the amount of the maximum flow is known as \( \sum_{e} c(f) \) in the present problem, the minimum cost flow algorithms are easily applicable. To change the flow over the flow network, a different set of the costs can be assigned. An example of the cost assignment is shown in Figure 2.

As shown in Figure 2, the flow for the idle task is controlled using costs \( w(\tau_{\text{idle}}, W_k) \). When the idle time needs to be clustered at the end of AJA’s time interval, the costs are assigned as follows:

\[
w(\tau_{\text{idle}}, W_k) = K - k + 1, \quad \forall k.
\]  
\[\text{(10)}\]

The costs for all of the other edges are set to 1. Then, the flow for the idle task avoids being sent through the higher-cost edges as long as it does not reduce the predefined amount of the maximum flow. It implies that the idle time is clustered at the end of the current AJA’s time interval as much as possible without the missing of any deadlines. This helps in the later accumulation of the idle time so that the idle time at the next boundary becomes longer.

Alternatively, when the idle time needs to be clustered at the current time, i.e., the start of AJA’s time interval, the costs are assigned as follows:

\[
w(\tau_{\text{idle}}, W_k) = k, \quad \forall k.
\]  
\[\text{(11)}\]

The costs for all of the other edges are also set to 1.

B. CLUSTERING OF IDLE TIMES

Using Equations 10 or 11, the idle time can be either clustered close to the end of the current AJA or clustered around the current time. Switching between the two modes, ClusterBackward (CB) and ClusterForward (CF), the setting of a current mode \( \text{curMode} \) is determined by the following rules:

1) Start in \( \text{ClusterBackward} \) mode.

2) In \( \text{ClusterBackward} \) mode, when the first window \( W_1 \) contains the idle time, switch to \( \text{ClusterForward} \) mode at the current AJA.

\[
\text{curMode} \leftarrow \text{CF}, \quad \text{if } X_{\text{idle}} > l_1.
\]  
\[\text{(12)}\]

It is implied that even if the algorithm attempts to cluster the idle time at the end of AJA’s time interval, the first window \( W_1 \) happens to contain the idle time. Thus, rather than wasting the capacity of the first window to run a piece of the idle task, it is preferable to start clustering the idle time.

3) In \( \text{ClusterForward} \) mode, when the first window \( W_1 \) contains the idle time whose length is less than the length of the \( W_1 \), switch to \( \text{ClusterBackward} \) mode at the next AJA.

\[
\text{curMode} \leftarrow \text{CB}, \quad \text{if } X_{\text{idle}} < l_1.
\]  
\[\text{(13)}\]

Since the clustered idle time ends in the middle of the first window, the idle time at the next window is going to be disconnected. Therefore, clustering the idle time close to the end of the time interval will be started at the next AJA.

C. ESTIMATING THE LENGTH OF THE CLUSTERED IDLE TIME

When the idle time is clustered, its length should be accurately estimated for the system to transit into the appropriate low-power state during the well-estimated time interval. However, a direct use of the flow-network formulation with the virtual idle task may cause wrong the estimation. For example, assume that \( W_1 = [0, 2], W_2 = [2, 4], W_3 = [4, 8] \) in AJA(0), as shown in Figure 3 (a). In ClusterForward mode, the flow network model generates the clustered idle time with \( \{X_{\text{idle},1}, X_{\text{idle},2}, X_{\text{idle},3}\} = [2, 2, 2] \), and the length of the clustered idle time is estimated as 6. At next boundary 2 (= \( d_1 \)), the AJA(2) is then constructed with \( \{W'_{1} = [2, 4], W'_{2} = [4, 6], W'_{3} = [6, 8]\} \). In AJA(2), the previous \( W_3 \) is divided into two new windows as \( W'_{2} = [4, 6] \) and \( W'_{3} = [6, 8] \). As shown in Figure 3 (b), both \( X_{\text{idle},2} \) and \( X_{\text{idle},3} \) can be computed as 1 and 1, respectively. It means that \( X_{\text{idle},3} \) was inaccurately estimated as 2 at the time 0, which later becomes separated into \( X_{\text{idle},2} \) and \( X_{\text{idle},3} \).

The wrong estimation is caused by the fact that the simple flow network model does not consider the boundaries, which will be established by inactive jobs (or future jobs) within the
time interval \([0, d^{\text{max}}]\). Therefore, we propose an algorithm that constructs the fine-grained windows in consideration of all of the possible release times and deadlines of each task within the time interval of the current AJA, named \textit{flow network-based DPM with fine-grained windows (fnDPM-fw)}. Since no more windows will be generated within the time intervals of the current AJA, \textit{fnDPM-fw} accurately estimates the length of the clustered idle time. A drawback of \textit{fnDPM-fw} is that the number of windows increases, the time complexity of the flow network model proportionally increases.

Alternatively, we also propose \textit{flow network-based DPM with coarse-grained windows (fnDPM-cw)} to alleviate the time complexity. Instead of additionally considering all of the possible boundaries constructed by the inactive jobs, \textit{fnDPM-cw} considers the boundaries constructed by break-even times. More specifically, within the time interval of AJA(t), \textit{fnDPM-cw} considers boundaries that are established at the time \(t + \text{BET}_p\) by the break-even times of \(p\)-th low-power state. The flow network using the coarse-grained windows is then formulated for each low-power state to verify which low-power state is usable at the current time without jeopardizing the schedulability of the current jobs. A drawback of \textit{fnDPM-cw} is that the estimation of the longest length of the clustered idle time is restricted by the longest BET. However, it benefits from the alleviated time complexity compared with \textit{fnDPM-fw} because the total number of low-power states supported in a real system is usually limited.

### Algorithm 1 fnDPM-fw

```plaintext
// Initially, \(G \leftarrow \text{null}\) and prevMode \(\leftarrow \text{CB}
1: \text{procedure} \ Schedule(t, \text{prevMode})
2: \quad \text{curMode} \leftarrow \text{prevMode}
3: \quad \text{if} \ \text{curMode} \text{ is CB then}
4: \quad \quad \text{G} \leftarrow \text{constructG_IdleTask(curMode)}
5: \quad \quad \{X_{i,1}|V_i\}, \{X_{\text{Idle},k}|V_k\} \leftarrow \text{mincost}(G)
6: \quad \quad \text{if} \ X_{\text{Idle},1} \text{ is 0 then Go to line 13.}
7: \quad \quad \text{else curMode} \leftarrow \text{CF}
8: \quad \end{if}
9: \end{procedure}
```

### Algorithm 2 fnDPM-cw

```plaintext
// Initially, \(G \leftarrow \text{null}\) and prevMode \(\leftarrow \text{CB}
1: \text{procedure} \ Schedule(t, \text{prevMode})
2: \quad \text{curMode} \leftarrow \text{prevMode}
3: \quad \text{if} \ \text{curMode} \text{ is CB then}
4: \quad \quad \text{G} \leftarrow \text{constructG_IdleTask(curMode)}
5: \quad \quad \{X_{i,1}|V_i\}, \{X_{\text{Idle},k}|V_k\} \leftarrow \text{mincost}(G)
6: \quad \quad \text{if} \ X_{\text{Idle},1} \text{ is 0 then Go to line 18.}
7: \quad \text{end if}
8: \text{end if}
9: \text{for} \ p = L, L - 1, \ldots, 2, 1 \text{ do}
10: \quad \text{G} \leftarrow \text{constructG_DecCap(BET)_p}
11: \quad \{X_{i,1}|V_i\}, \text{sum_of_flow} \leftarrow \text{maxflow}(G)
12: \quad \text{if} \ \text{sum_of_flow} \text{ is equal to } \sum_{i} c(t) \text{ then}
13: \quad \quad \text{SelectPowerState(null, p)}
14: \quad \quad \text{Go to line 18.}
15: \quad \text{else \{X_{i,1}|V_i\} \leftarrow \text{null}
16: \quad \text{end if}
17: \end{for}
18: \text{end procedure}
```

At every boundary, both algorithms invoke the \textit{Schedule} procedure with arguments including the current time \(t\) and the previous mode \textit{prevMode}. Then, they construct a flow network depending on the value of the \textit{curMode}. At the end of this procedure, it returns the reserved execution time for all of the tasks and the \textit{curMode} to update the \textit{prevMode}. Further details about these procedures, from Algorithms 1 and 2, are described below.

- **Input**: The argument \textit{mode} is either \textit{ClusterForward} or \textit{ClusterBackward}. If the \textit{mode} is \textit{Cluster Backward}, then it assigns the costs to the edges of the idle task using Equation 10. Otherwise, it assigns the costs using Equation 11. In addition, each \(\textit{Cap}(W_k)\) is calculated using Equation (3) and then this value is used for the construction of \(G\).
- **Output**: The constructed graph including the virtual idle task is returned.

### D. SCHEDULING ALGORITHMS

The pseudo-codes of \textit{fnDPM-fw} and \textit{fnDPM-cw} are presented in Algorithms 1 and 2, respectively. Both algorithms use a variable \textit{curMode} for monitoring the current mode where \textit{curMode} is initially set to \textit{ClusterBackward}.

\begin{align}
\text{Cap}_\text{new}(W_k) &= \text{Cap}(W_k) - l_k, \\
&\text{where } \{k \mid b_k \leq t + time, \forall k\}
\end{align}
It is assumed that the subtracted capacity here will be used for the idle task. Therefore, the numbers of nodes and edges for the virtual idle task in the flow network are reduced, thereby alleviating the complexity of Algorithm 2.

- **Output**: The constructed graph without the virtual idle task is returned.

- \( \text{mincost}(G) \)
  - **Input**: \( G \) is the flow network.
  - **Output**: The set of the reserved execution time for all of the real-time tasks within the first window and the set of reserved execution time for the virtual idle task within all of the windows of the current AJA are returned by solving the problem of minimizing costs of flows.

- \( \text{maxflow}(G) \)
  - **Input**: \( G \) is the flow network.
  - **Output**: The set of reserved execution time for all of the real-time tasks within the first window and the total reserved execution time for all of the real-time tasks within all of the windows of the current AJA are returned by solving the problem of maximizing flows.

- SelectPowerState(flow, index)
  - **Input**: The flow is the actual flows of an idle task. The index can be either \( \text{null} \) or the index of the low-power state. If the index is \( \text{null} \), the procedure of SelectPowerState() finds the lowest possible power state where flow is less than its BET. If the index is \( \text{null} \), then it directly selects the index-th low-power state.

Using a set of procedures above, fnDPM-fw and fnDPM-cw run differently depending on curMode, as follows:

- In ClusterBackward mode,
  - fnDPM-fw attempts to find a feasible solution with the clustered idle time that is close to the end of the AJA’s time interval in lines 3-9 of Algorithm 1. Using the constructG_IdleTask() in line 4, fnDPM-fw constructs the flow network \( G \) and assigns the costs to the edges between the idle task node and the window nodes using Equation 10. Then, it finds the solution using the mincost() in line 5. This solution is used for the scheduling if the first window \( W_l \) does not contain the idle time, as shown in line 6. This means that fnDPM-fw successfully clusters the idle time at the end of the AJA’s time interval; this forces the curMode to retain ClusterBackward in order to cluster the idle time at the end of the time interval again at the next boundary. This procedure is shown in lines 13-14. Otherwise, it prepares the clustering of idle time by setting the curMode to ClusterForward.

- fnDPM-cw performs a procedure that is the same as that of fnDPM-fw in lines 3-8 of Algorithm 2.

- In ClusterForward mode,
  - fnDPM-fw newly constructs \( G \) and assigns the costs to the edges of the idle task by Equation 11 in line 10 of Algorithm 1. Then, it finds a solution using mincost() in line 11. Especially in ClusterForward mode, this algorithm finds which low-power state is available using the length of the clustered idle time and it selects the available low-power state through the SelectPowerState() in line 12. Then, fnDPM-fw sets the curMode to ClusterBackward for the next boundary if the length of the idle time is less than \( l_1 \), thereby merging the current idle time with the previous idle time. If the length of the idle time is equal to \( l_1 \), fnDPM-fw sets the curMode to ClusterForward. As a result, the current idle time is merged with the previous idle time. These procedures are shown in lines 13-16.

- fnDPM-cw verifies which low-power state is possible to transit into in lines 9-17 of Algorithm 2. First, fnDPM-cw constructs \( G \) with the exclusion of the virtual idle task using the BET of low-power state by the constructG_DecCap() in line 10. Then, it finds the solution using the maxflow() in line 11. When the maximum flow occurs in the given problem, the left-hand sides of all of Equation 4 should be equal to the summation of \( c_i(t) \) values for all of the tasks. Thus, the feasibility of the solution can be verified by line 12. If this solution is feasible, then fnDPM-cw selects the low-power state corresponding to BET by calling the SelectPowerState() in line 13. Otherwise, it iterates the above procedures in lines 10-11 with the next shallower low-power state. The procedures for the updating of the curMode in lines 18-21 of Algorithm 2 are performed similarly to those of fnDPM-fw in ClusterForward mode.

To compare the proposed algorithms with LPDPM [17], the actual scheduling on the four processors is presented in Figure 4 using the task set in Table 2. Each number in these figures means the index of the real-time tasks and the character \( I \) means the idle task.

**Example 1**: Figures 4 (a) and 4 (b) show the actual scheduling by LPDPM and fnDPM when the actual execution time of each task is equal to its WCET, respectively. In Figure 4(a), LPDPM first clusters the idle time around the start time \( t = 0 \) as much as possible and merges the residual idle time around the time \( t = 24 \) upon the arrival of the next job. As a result, it generates two chunks of idle time that are used for the initiation of the low-power state. In Figure 4 (b), fnDPM sets the curMode to ClusterBackward at \( t = 0 \), but the idle time is contained in the first window. Therefore, fnDPM flips the curMode to ClusterForward, thereby generating a long length of idle time that is used for initiating the low-power state. However, fnDPM also clusters the idle time around \( t = 16 \) even though the length of the idle time is short. It is because at time 0, fnDPM does not consider the idle time after \( t = 24 \). Although fnDPM wastes 2 units of idle time at \( t = 16 \), it clusters the idle time similarly to that of the LPDPM in the most of the time.
Example 2: Figure 4 (c) and 4 (d) show the actual scheduling by LPDPM and fnDPM, respectively when the actual execution time of each task is less than its WCET. A set of the actual execution times is set to \{2,2,3,4,13\}, each of which is less than WCET in Table 2.

In Figure 4 (c), LPDPM first schedules the real-time tasks using the offline schedule. At the time 0, LPDPM first executes \(\tau_1, \tau_2, \tau_4\), and \(\tau_{Idle}\) on the four processors as determined by the offline schedule. When \(\tau_1\) and \(\tau_2\) early complete at \(t = 2\), the extra idle time is not added to the idle time at the last processor, and it is instead added to the first processor. This is because LPDPM can only make a slight change on the schedule generated by the offline procedure to maintain its feasibility.

Contrarily, as shown in Figure 4 (d), fnDPM continuously adds the extra idle time to the last processor. This is possible because fnDPM formulates and solves the scheduling problem in the consideration of the extra idle time when the early completion occurs. In this example, when \(\tau_1, \tau_2,\) and \(\tau_3\) early complete, fnDPM reconstructs the flow network including the extra idle time and it finds a solution that clusters the maximum idle time into the last processor. As a result, fnDPM retains the last processor in the low-power state for the long time interval.

E. COMPLEXITY OF THE PROPOSED ALGORITHM

The proposed algorithms are invoked at each boundary and if necessary, when the early completion occurs. Both algorithms use existing solvers for the maximum flow problem and the minimum cost problem. Since the computational complexities of the solvers are higher than those of the other procedures such as \(\text{constructG}\_\text{IdleTask}\) and \(\text{constructG}\_\text{DecCap}\), the complexity of the solvers is dominant in both of the algorithms. For the minimum cost flow problem in both algorithms, the solver that was introduced by Orlin [26] is used. This solver is called by an enhanced capacity-scaling algorithm and it comprises a \(O(|V| + |E|)\) bound for \(SP_+(|V|, |E|)\) in [27]. For the maximum flow problem in Algorithm 2, a strongly polynomial solver with \(O(|V||E|)\) complexity [28] is used, which was introduced recently.

In fnDPM-fw (Algorithm 1), the maximum number of boundaries of the task \(\tau_j\) in the time interval \([t, d_{\text{max}}(t)]\) is \(\lceil(d_{\text{max}}(t) - t)/T_j\rceil + 1\). The longest length of \([t, d_{\text{max}}(t)]\) is \(T_{\text{max}}\), where \(T_{\text{max}} = \max_{T_j}\{T_j\}\) and thus, \(K\) is proportional...
to \( \sum_{|V_i|}[T^{max}_i/T_i] \), where \( \sum_{|V_i|}[T^{max}_i/T_i] \) is denoted by \( N' \). Therefore, \(|V| \) is proportional to \( N' \) and \(|E| \) is proportional to \( N'N \). The computational complexity of Algorithm 1 is \( O(N^2N'\log N') \).

Alternatively, inDPM-cw (Algorithm 2) considers a maximum of \( N + 1 \) windows in the time interval \([t, d^{max}(t)]\). Thus, its \(|E| \) is the summation of \(|e(n_i, n_j)|v_j|\) = \( N \), \(|e(n_1, \ldots, N_k)|v_k|\) = \( (N+1)(N+2)/2 = (N^2+3N+2)/2 \), and \(|e(W^k, n_k)|v_k|\) = \( N + 1 \). Additionally, the number of nodes \(|V| \) is \( N \). As a result, the computational complexity in line 5 is \( O(N^3\log N) \). In addition, when constructG_DecCap() is run at line 10, the number of nodes \(|V| \) becomes \( 2N + 4 \) according to the consideration of a virtual idle task and an additional window. The number of edges \(|E| \) becomes \( (N^2 + 5N)/2 \), where both \(|e(n_i, n_j)|v_j|\) and \(|e(W^k, n_k)|v_k|\) are \( N \) and \(|e(n_1, \ldots, N_k)|v_k|\) is 1 + 2 + \ldots + \( N = N(N+1)/2 \) in the worst-case. Thus, the maximum flow network in line 11 comprises \( O(LN^3) \) and thereby the complexity of Algorithm 2 is \( O(LN^3\log N) \). Although \( L \) is multiplied to the complexity, \( L \) is usually much lower than \( N \) in modern hardware.

**VI. EXPERIMENT**

We conducted several experiments to compare the proposed algorithms with LPDPM using STORM, a simulator that was designed to evaluate the real-time schedulers on multiprocessors [29].

**TABLE 3. Specifications for the low-power states.**

<table>
<thead>
<tr>
<th>States</th>
<th>Current</th>
<th>Wake-up time</th>
<th>Penalty</th>
</tr>
</thead>
<tbody>
<tr>
<td>Active</td>
<td>74 mA</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Idle</td>
<td>20 mA</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Sleep</td>
<td>8.8 mA</td>
<td>5 ( \mu )s</td>
<td>1 mW</td>
</tr>
<tr>
<td>Deep sleep</td>
<td>0.145 mA</td>
<td>51 ( \mu )s</td>
<td>7 mW</td>
</tr>
<tr>
<td>Deep power-down</td>
<td>0.003 mA</td>
<td>200 ( \mu )s</td>
<td>28 mW</td>
</tr>
</tbody>
</table>

**A. SIMULATION ENVIRONMENT**

We assume that the tasks were running on four processors, where each processor can transit into one of three low-power states. Table 3 shows the parameters of each of the low-power states that were used in these experiments. They were determined by referring to NXP LPC1800-series micro-controllers (MCU) that was designed based on ARM Cortex-M3 processor. The MCU has been commonly used in several areas, e.g., motor-control, industrial-automation, and embedded-audio applications [31]. The parameters include the regulator supply current, the wake-up time, and the power penalty for each low-power state. The wake-up time and the power penalty are required and consumed to return to the active state, respectively. The break-even time was set to the same as the wake-up time, which is a reasonable assumption as described in [4].

In the active state, the cores are fully operational and can access the peripherals and memories that are configured by their running software. In the sleep state, the cores receive no clock pulse but peripherals and memories remain running. In both deep sleep and deep power-down states, all cores and peripherals except the peripherals in the always-on power domain are shut-down. Memories can remain powered for retaining memory contents as defined by the individual power-down state.

We generated 1,000 real-time task sets per each utilization \( U \) ranging from \([3.0, 4.0]\). During the generation of the task sets, every task set was abandoned if the time for the solving of its corresponding linear programming problem, as formulated by LPDPM, exceeds 10 min. It was simply to avoid a protracted experiment. The period of each task was randomly chosen using the uniform distribution in the interval between 0.01 ms and 10 ms. Then, the WCET was randomly chosen as a value from 0.1 to 1.0 times of its period.

For the evaluation of the energy-efficiency of the algorithms, the following three metrics were used:

- The static energy consumption.
- The total power overheads for the transition between the low-power states and the active state.
- The time interval during wherein the processors stayed in each low-power state.

Here, the static energy consumption depends on the other two metrics. For example, even if the total energy overhead is high, the static energy consumption can be reduced by staying in a low-power state for a long time. Another example is that staying in the shallow sleep state for a long time can be better than staying in the deep sleep state for a short time in minimizing the static energy consumption. These metrics will reveal the characteristic of each algorithm in energy savings.

**B. SIMULATION RESULTS**

The metrics for the comparison are specified as follows:

- The normalized static energy consumption is calculated as follows:
  \[
  \frac{\sum_{\{\text{low-power state } p\}} (PC_p \times t_p + PP_p)}{PC_{idle} \times t_{idle}}
  \]

  - \( t_p \) is the total time interval for which a DPM algorithm remains in the \( p \)-th low-power state.
  - \( t_{idle} \) is the total time that a non-DPM algorithm spends in the idle state.
  - \( PC_{idle} \) is the power that a non-DPM algorithm consumes in the idle state.

- The total power overheads is calculated as follows:
  \[
  \sum_{\forall p} (Tr_p \times PP_p)
  \]

  - \( Tr_p \) is the total number of transitions between the active state and the \( p \)-th low-power state.

- The normalized time spent in each low-power state is calculated as follows:
  \[
  \frac{t_p}{t_{idle}}, \quad \forall p
  \]
Figures 5 and 6 show the experiment results when the actual execution time of each task is equal to its WCET. Figure 5 (a) shows the static energy consumption of each algorithm that has been normalized by that of a non-DPM algorithm (B-Fair). This figure shows that compared with B-Fair, both LPDPM and the proposed algorithms significantly reduce the static energy consumption over the whole range of the utilization. The difference of the energy consumption between LPDPM and the proposed algorithms stays within 5%. Figure 5 (b) depicts the total energy overhead for each algorithm. It shows that fnDPM-cw incurs a higher total energy overhead than both LPDPM and fnDPM-fw, which implies that fnDPM-cw transits into the low-power states more frequently than the others.

Figures 6 (a), (b), and (c) show the total time spent in the low-power states, as normalized by the total idle time of the given tasks. Figure 6 (a) shows that LPDPM utilizes almost 100% of the idle time for staying in one of the low-power states. In addition, it stays in the deep-power down state for a long time. Alternatively, fnDPM-fw utilizes an idle time that is shorter than that of LPDPM. In addition, it also stays in the deep-power down state for a shorter time than that of LPDPM. A similar but attenuated trend is found in Figure 6(a) and (b) between LPDPM and fnDPM-cw. Note that as the utilization increases toward 4.0, the total idle time was decreased. Thus, as the utilization increased, the normalized total time spent in the low-power states affected the power consumption decreasingly.

On the other hand, Figures 7 and 8 show the results when the actual execution time of each task is allowed to be less than its WCET. Figure 7 (a) shows the static energy consumption of each algorithm, as normalized by that of B-Fair. The early completion of the tasks produces the extra idle time dynamically, thereby providing another chance for the reduction of the energy consumption. Especially when the utilization is low (3.0 < U < 3.5), both fnDPM algorithms consume less static energy than LPDPM. Although, fnDPM-cw incurs a higher total energy overhead than both LPDPM and fnDPM-fw in Figure 7 (b), as observed in the
FIGURE 7. The energy consumption and the power penalty of algorithms when the actual execution time is less than WCET. (a) Normalized static energy consumption. (b) Total power overheads for state-transitions. 

FIGURE 8. The time measurement of algorithms when the actual execution time is less than WCET. (a) Total time that LPDPM spent in each low-power state. (b) Total time that fnDPM-fw spent in each low-power state. (c) Total time that fnDPM-cw spent in each low-power state.

previous experiment. This finding implies that fnDPM-cw causes more frequent transitions into the low-power states for the maintenance of a continual presence in those states. Unlike fnDPM-cw, fnDPM-fw incurs less frequent transitions into the low-power states, since it accurately estimates the idle time longer than BET. When the utilization is high $(3.5 < U < 4.0)$, LPDPM consumes less static energy than the two fnDPM algorithms. The difference between the energy consumption of LPDPM and the proposed algorithms stays within 3%.

Figures 8 (a), (b), and (c) show that, when the early completion occurs, both of the fnDPM algorithms utilize a longer idle time in the low-power states than LPDPM. It causes both fnDPM algorithms to consume less energy than LPDPM when the utilization is low $(3.0 < U < 3.5)$, even if the two fnDPM algorithms stay in the deep power-down state for a shorter time than LPDPM. This finding implies that the fnDPM strategy of the frequent transiting into the shallow sleep states, rather than an intermittent transiting into the deep sleep state, is advantageous, especially when the extra idle time is dynamically available and the task utilization demand is low. The result is consistent with the example that is illustrated in Figure 4.

VII. FUTURE WORKS AND DISCUSSION

A real-time scheduling algorithm based on both DVFS and DPM can reduce the dynamic and static energy consumption but it has a trade-off in handling the slack time. To reduce the dynamic energy consumption, DVFS-based scheduling algorithms scale down the operating frequency using the slack time and thus, it makes the execution time of tasks increase. The increased execution time however reduces the length of an idle time which DPM-based algorithms can utilize for entering the low-power state. This trade-off makes it difficult to find the energy optimal solution for reducing both dynamic and static energy consumption.

In order to resolve the above problem, many researchers studied interplay of DVFS and DPM [8], [10]–[13]. In [12], the scheduling algorithms supporting the interplay of DVFS and DPM generally have to determine: (a) the switching time instant for turning the processor from a low-power state to an
active state, (b) the target frequency and the time instant for scaling, and (c) the time instant for turning the processor back to a low-power state. As mentioned above, since it is difficult to consider these features at once, the algorithms based on interplay of DVFS and DPM use the slack time for applying DVFS first and subsequently use the residual slack time for applying DPM, or vice versa.

Langen and Juurlink [10] proposed several scheduling algorithms, extensions of their previous work in [8], by incorporating DVFS, shutdown, and procrastination. These algorithms find the optimal number of active processors like LAMPS [8] and also heuristically determine either the scaled frequency or the time point for the processor shutdown.

Lee [11] proposed a heuristic scheduling algorithm for light workloads on multi-core platform. This algorithm lowers the frequency of the overabundant cores to reduce the dynamic energy consumption and then it turns off rarely used cores after moving their tasks to other cores to reduce the static energy consumption.

Chen et al. [12] presented an energy efficient framework that considers the interplay of DVFS and DPM. In this framework, a power management strategy first determines the optimal frequency based on current workloads without jeopardizing the schedulability of real-time tasks and then uses the idle time interval to turn processors to sleep modes.

Moulik et al. [13] proposed an energy-aware real-time scheduling algorithm for hard real-time multicore systems using interplay of DVFS and DPM. This algorithm guarantees RT-optimality because it is based on RT-optimal DP-Fair scheduling algorithm (deadline partitioning-fair [33]). Their power management strategy first determines the minimum operating frequency and then compares it with critical frequency. As defined in [5], if the operating frequency is less than critical frequency, then the static energy consumption dominates dynamic energy consumption, which is known to result in increasing the total energy consumption. Therefore, the critical frequency can be the lower bound for achieving high energy savings. The algorithm proposed by Moulik et al. [13] assigns current tasks to each core with the minimum operating frequency to fully use the processor capacity, when the minimum operating frequency is higher than critical frequency. Otherwise, it scales the operating frequency up to the critical frequency and calculates the residual processor capacity for determining the time interval to enter the sleep state. Instead of minimizing the number of active processors, this algorithm chose to keep all processors powered on to scale the frequency at any time based on DVFS.

Our study is based on two premises for achieving energy savings. First, the strategy of shutting down as many processors as possible is more energy-efficient than keeping processors idle for applying DVFS later, as claimed in [8] and [9]. Second, the static power consumption has been growing over the dynamic power consumption [5], [14]. These premises have led us to designing DPM-based scheduling algorithms that generate the longer idle time for entering the deeper low-power state.

Our experimental results shows that the proposed algorithms reduce the static energy consumption similarly with the latest DPM-based offline algorithm [17]. In addition, our algorithm can be easily extended to reduce the dynamic energy consumption by adopting the interplay of DVFS and DPM. For example, when the break-even time of a processor is longer than the available idle time, our algorithm can be extended to use the idle time for scaling down the operating frequency of the processor. However, even if this simple heuristic enables the proposed algorithm to use both DVFS and DPM without wasting the idle time, it does not guarantee that it is the most energy-efficient approach. Instead, we believe that there should be a way to consider both DVFS and DPM in a single mathematical formulation, that eventually provides us with the optimal energy-efficiency. In this sense, we claim that our mathematical formulation in this paper is a good starting point toward obtaining the optimal energy-efficiency.

In order to find a solution, we used the solvers (such as [26]–[28]) for the general minimum cost flow problem. The complexity of the solvers is dominant in the proposed algorithms, since the solution for the general minimum cost flow problem has relatively high complexity. To reduce the complexity further, we need to use a different solver especially designed for our problem. For example, the scheduling problem in this paper always leads to a form of unbalanced bipartite graph where (1) the vertex set is partitioned into two subsets of tasks and windows and (2) the number of tasks and the number of windows are not always the same. It can give a chance for us to use solvers specialized in the unbalanced bipartite graph, which is computationally lighter than the solver for the general minimum cost flow problem [26]–[28].

VIII. CONCLUSIONS

An online real-time scheduling algorithm, fnDPM, is proposed in this work, using the DPM on a symmetric homogeneous multiprocessor to schedule periodic real-time tasks and to achieve static energy savings. The focus is the clustering of the distributed idle times to make a processor remains in an appropriate low-power state for a long time without jeopardizing the real-time constraints while using the minimum number of active processors. Since fnDPM is based on the flow network model, fnDPM efficiently generates long clustered idle times while satisfying the real-time constraints. An experimental evaluation of the proposed algorithms was conducted in comparison with an existing offline approach that is the only counterpart in the problem domain. Through the experiments, it is shown that the proposed algorithms achieved the static energy savings in manner that is similar to that of the existing algorithm. Especially, it is shown that, in the case of an early completion that occurs during a low utilization of real-time tasks, the proposed algorithms consume less static energy than the existing algorithm. For future works, an investigation on a procedure for the further reduction of the total energy consumption
through the integration of DVFS and DPM will be performed.

REFERENCES


JU-DERK PARK received the B.S., M.S., and Ph.D. degrees in information and communication engineering from Chungbuk National University, Cheongju, South Korea, in 1995, 1997, and 2011, respectively. He has been with the Electronics and Telecommunications Research Institute, Daejeon, South Korea, since 2000. His major research areas include the analysis of EM fields and communications for the Internet of Things.

BYEONG-CHEOL CHOI received the B.S. degree from Hanyang University, in 1987, and the M.S. degree from Hannam University, in 1997, both in electrical engineering, and the Ph.D. degree in computer science from Busan National University, South Korea, in 2004. He has been with the Electronics and Telecommunications Research Institute, Daejeon, South Korea, since 1993, leading many projects in the area of wireless sensor networks. Currently, his major research is in communications for the Internet of Things.