IEEE Access
Multidisciplinary ⋮ Rapid Review ⋮ Open Access Journal

Received December 13, 2021, accepted January 3, 2022, date of publication January 7, 2022, date of current version January 18, 2022.

Digital Object Identifier 10.1109/ACCESS.2022.3141086

A Lightweight Optimal Scheduling Algorithm for Energy-Efficient and Real-Time Cloud Services

JOOHYUNG SUN[1,2] **AND HYEONJOONG CHO**[2]
[1]Electronics and Telecommunications Research Institute, Daejeon 34129, South Korea
[2]Department of Computer Convergence Software, Korea University, Sejong City 30019, South Korea

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

This work was supported by the National Research Foundation of Korea (NRF) Grant through the Korea Government under Grant NRF-2021R1F1A1049202.

ABSTRACT To support ever-chainging user needs such as large storage volumes, web search, and high-performance computing, numerous companies have expanded their systems to cloud computing servers. Cloud environment systems generally consume large amounts of electrical power, leading to tremendously high operational costs. In addition, they require computing infrastructures to run various real-time applications such as financial analysis, cloud gaming, and web-based real-time services. To represent performance guarantees, the negotiated agreements in real-time computing, expressed as deadline (or latency), can be specified by *service level agreements* of cloud services between users and cloud server providers. Thus, a number of research works have started focusing on reducing the energy consumption and simultaneously satisfying the temporal constraint in a cloud environment. Although we previously proposed an optimal real-time scheduling algorithm for multiprocessors, it is difficult to use it for cloud environments handling a large number of cloud services because of the high computational complexity of $\Omega(N^3 logN)$, where N is the number of tasks. Thus, we introduce a real-time task scheduling algorithm for cloud computing servers, which alleviates the computational complexity of $O(N^2)$ from the complexity of the previous algorithm using a novel flow network-based optimization method. To the best of our knowledge, our scheduling algorithm in a cloud environment, which ensures optimality for real-time tasks and achieves energy savings using dynamic power management simultaneously, is the first in the problem domain. We show that the proposed scheduling algorithm guarantees an optimal schedule for real-time tasks and achieves energy savings simultaneously. Our experimental results show that the proposed algorithm outperforms the latest existing algorithms in terms of both time complexity and energy efficiency.

INDEX TERMS Cloud computing, dynamic power management, energy-aware algorithm, flow network problem, optimal scheduling, real-time computing.

I. INTRODUCTION

Numerous companies (e.g., Amazon, Google, Facebook, and Yahoo) have expanded their systems and operations to cloud computing servers for satisfying ever-chainging user demands, such as large storage volumes, web search, and high-performance computing. Cloud computing offers a new computing model for sharing computing resources such as servers, networks, memory, and storage. It also delivers computing services to users as a utility in a pay-as-you-go manner [1]. According to [2] and [3], cloud computing will continue to grow at an unprecedented rate in the coming years. They also list the following detailed insights:

(i) Sixty percent of all information technology (IT) investment in 2018 would be on cloud-based solutions and this figure will be even higher in the future.

(ii) The public cloud market would grow at an annual rate of 22% and reach US$236 billion in 2020.

(iii) Enterprise cloud investment will grow at an annual rate of 16% until 2026.

Various cloud service providers have put in place a large-scale computing infrastructure in data centers to supply on-demand system resources to approved users over the Internet. Data centers are cost-effective infrastructure consisting of thousands of computing servers, routers, switches,

The associate editor coordinating the review of this manuscript and approving it for publication was Alon Kuperman.

and a higher memory bandwidth. They can be dynamically shared by a number of users through virtualization of physical resources.

According to [4] and [5], data centers consume large amounts of electrical power, leading to tremendously high operational costs. They also predicted that the cost of electrical power will in a few years' time exceed the cost of the infrastructure for the data center itself. In 2013, U.S. data centers consumed an estimated 91 billion kWh of electricity, equivalent to the annual output of 34 large (500-megawatt) coal-fired power plants. The annual electricity consumption of data centers was projected to increase to approximately 140 billion kWh by 2020.

Furthermore, many applications of pay-as-you-go-based cloud computing require *service-level agreements* (SLAs) between users and cloud service providers. In real-time computing, the negotiated agreements expressed as *deadline* (or *latency*) can be specified by the SLA of cloud services. According to [6], latency in cloud gaming applications determines not only how players experience online gameplay but also how to games should be designed so that they meet player expectations. The authors show that the effect of latency on actions can be categorized by the precision and deadline demands of the action, the game's interaction model, and the player's perspective. Several examples that demand real-time constraints are as follows:

(i) Cloud gaming [6]: 100 ms for first-person shooter and racing, 500 ms for sports and role-playing games, and 1000 ms for real-time strategy and simulation games.

(ii) Long-term evolution technology [7]: Mobile operators reported handset-to-base-station latencies of approximately 2 ms (round-trip time of 4 ms).

(iii) Amazon web services [7]: 500 ms is the maximum latency observed for Amazon Web Services according to CloudPing, available at http://www.cloudping.info.

(iv) Dynamic web pages [8]: Commonly, the latency of static or simple dynamic web pages can be assumed to range from 0.3 to 150 ms.

In this study, we propose an energy-efficient scheduling algorithm for cloud environments that guarantees an optimal schedule for real-time tasks and achieves energy savings simultaneously. Meeting deadlines is an important issue because they are directly related to the performance metrics of SLAs. Thus, ensuring the optimality of real-time scheduling, which can maximize the degree of SLAs, is a valuable contribution to cloud services. In the context of this paper, we define the optimality of the real-time schedule as the same as in [12] for ease of understanding.

Definition 1 (RT-optimality): An optimal real-time schedule meets all 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 addition, RT-optimality is well suited to *dynamic power management* (DPM). DPM can dynamically transit an idle state of processors into low-power states by disabling some of the system parts to reduce leakage of power consumption. It can also shut down idle processors for the entire system runtime. Thus, the scheduler reduces energy consumption by utilizing a minimum number of processors only to guarantee RT-optimality, and then it turns off all the other residual processors. Considering that cloud computing servers such as super computers and data centers have numerous processing resources, DPM offers a number of opportunities for reducing leakage power consumption. In addition, these servers consume considerable power for a cooling (or ventilation) system to prevent system failures because they are sensitive to high temperatures. Moreover, leakage power (i.e., static energy consumption) increases exponentially with temperature, unlike dynamic power [35]. Therefore, applying DPM to a cloud environment can result in high energy savings.

In summary, the proposed algorithm designed to guarantee RT-optimality and simultaneously achieve energy savings in a cloud environment is capable of

- Finding the minimum number of active processing elements to process users' request,
- Initiating and staying at low-power states of active processors as long as possible, and
- Shutting down inactive processors for whole system runtime.

However, there are some practical issues that need to be addressed as follows.

- Increasing resource utilization by the additional cloud service requests of users.
- Overheads of VM migration for global load balancing of processing elements.
- High temporal complexity of the scheduling algorithms.

To address the first issue of additional requests, a reformulation of the scheduling problem is required. The objective of the reformulated scheduling problem is to minimize the number of active PEs by deliberately migrating VMs that cause their resource utilization to exceed the resource capacity. To achieve this goal, the cloud provider should determine which PE will be activated and which VMs will be migrated.

However, migrating VMs between different PMs imposes time overheads for changing the state of the processor, the contents of memory, etc. Fortunately, the overhead can be reduced using the *live VM migration* scheme introduced in [33]. Live VM migration can move a running virtual machine or application between different PMs without disconnecting the client or application. Thus, in this study, we also assume that a VM required to be migrated is managed by live VM migration. More details about the procedures for live VM migration are explained later.

In this study, we significantly reduced the computational complexity of the scheduling algorithm compared with the latest works [12]. Because the complexity of the previous algorithm was $\Omega(N^3 logN)$, where N is the number of real-time tasks, it is difficult to dynamically schedule real-time tasks during system runtime. In particular, this overhead is a critical problem in cloud environments that have a large number of services to be performed. To solve this problem,

TABLE 1. Related works about energy issue and real-time theory.

Algorithm	Type of VM allocation	Energy-saving technique	RT-optimality (type of RT constraint)
Dong [22]	static	DPM and Shutdown	X
Cao [23]	static	DVFS	X (hard)
Hadji [25]	dynamic	-	X
Ding [26]	dynamic	DVFS	X (soft)
Guo [27]	dynamic	Shutdown	X
Kim [9]	dynamic	DVFS	X (soft)
Ours	dynamic	DPM and Shutdown	O (hard)

we first show where the most time-intensive part is and how to alleviate the temporal overhead. Then, we formulated the problem as a flow network and proposed a dedicated solver algorithm while reducing performance degradation. We also evaluated the proposed algorithm by measuring the elapsed time of the proposed solver to find a feasible solution and the energy efficiency of the scheduling algorithm in a cloud environment.

A. ORGANIZATION

The remainder of this paper is organized as follows. Section II introduces the related works. Section III describes the system models. Section IV describes the proposed algorithm and presents its computational complexity with theoretical proofs. Section V presents the experimental evaluations of the time complexity and energy efficiency. Section VI presents our in-depth discussions and future work, and Section VII concludes this paper.

II. RELATED WORKS

A. VIRTUALIZATION FRAMEWORKS FOR REAL-TIME COMPUTING

Two leading open-source hypervisors, *Xen* and *kernel-based virtual machine* (KVM), are generally used to manage scheduling tasks or VMs. Although KVM and Xen have comparable performance, Xen has the advantage of supporting hyper-threading [14]. In addition, according to [15], KVM shows degraded performance when the number of virtual machines equals or exceeds the number of physical cores available. Thus, our discussion focuses on the Xen hypervisor. More details about the comparison between these two hypervisors can be found in [14] and [15]. In particular, the Xen hypervisor was originally developed at the University of Cambridge and is now distributed by Citrix Systems, Inc. The first public release of Xen occurred in 2003 [16]. The Xen hypervisor [17] supports several different virtual CPU schedulers with different properties. The role of a hypervisor in resource management is to dynamically map the virtual CPUs (vCPUs) in VMs to the physical CPUs (pCPUs) in hosts according to their own scheduling policy. According to [17], the CPU schedulers in the Xen hypervisor are as follows:

(i) *Credit* scheduler is a general-purpose weighted fair share scheduler, and is the current default.

(ii) The *Credit2* scheduler is an extension of Credit for higher scalability with a latency-sensitive workload, while still being based on a general purpose and weighted fair share scheduling policy.

(iii) The *sEDF* (simple earliest deadline first) scheduler is a real-time scheduler that provides weighted CPU sharing in an intuitive manner. Unfortunately, sEDF was removed from Xen 4.6 since behavior and performance were unideal and unreliable [17].

(iv) *RTDS* (real-time deferrable server) scheduler is a real-time scheduler aiming at supporting real-time workloads in the cloud. The RTDS is designed for soft and firm real-time tasks.

(v) The *ARINC653* scheduler is an embedded (automotive and avionics) real-time scheduler. ARINC653 is designed for hard real-time tasks, avionics, drones, and medicine, but its multicore support has not yet been implemented.

With respect to real-time scheduling, sEDF uses the traditional EDF algorithm on every CPU using a local queue, but it lacks global load balancing on multiprocessors. As a result, sEDF cannot guarantee RT-optimality on each processor because migration is not allowed for VMs that have already been assigned to processors. RTDS follows a scheduling policy based on the *preemptive global earliest deadline first* (Global-EDF) using a global queue, but Global-EDF does not guarantee RT-optimality. From the beginning, RTDS was proposed to change the scheduling model from the quantum-driven (i.e., credit scheduler) to an event-driven model in order to incur less scheduling overhead. The scheduler also uses a deferrable server to improve the average response time of the aperiodic requests.

In addition, S. Xi *et al.* [18] developed the first real-time virtual machine manager that supports hierarchical real-time scheduling in Xen, which is called RT-Xen. This framework provides an open-source platform for researchers and integrators to develop and evaluate real-time scheduling techniques. A key technical contribution of RT-Xen is the instantiation and empirical study of a suite of fixed-priority servers (e.g., deferrable server, periodic server, polling server, and sporadic server) within a VMM. In addition, J. Lee *et al.* [19] extended RT-Xen to support a compositional real-time scheduling capacity that uses periodic resource models as component interfaces. All of these works bridge the gap between real-time scheduling and hypervisor, thus developing an attractive virtualization platform for real-time systems.

B. SCHEDULING VIRTUAL MACHINES ON CLOUD COMPUTING

Many studies focused on improving resource utilization and reducing energy consumption in cloud computing have been conducted [20]–[23], [25]–[27]. The allocation of VMs to PMs in data centers is a key optimization problem for cloud providers to improve resource utilization and reduce energy consumption. As resource utilization increases, energy efficiency decreases, and vice versa. Thus, there exists a trade-off between performance and energy savings in the allocation of VMs to PMs. It is well known that the VM allocation problem is a special case of a classic bin-packing problem, which is an *NP-hard* problem.

In cloud computing, there are two approaches to VM allocation: *static* and *dynamic*. The static approach considers allocating VMs onto available PMs to satisfy the required hardware resources based on SLAs. This is normally applied for initial VM allocation, but resource utilization can be degraded if there is a large fluctuation in workload. In contrast, the dynamic approach is used to deal with the largely fluctuating workloads by dynamically reallocating all or several VMs onto the available PMs. When the resource usage of the allocated VMs on a PM increases, this approach can find a VM that needs to be migrated to satisfy the SLAs. In addition, this approach suits the various types of SLAs determined by cloud providers and users. It can also cope well with newly added or modified user requests online without waste of resource usage. However, the cost of VM migration can increase if a large number of VMs migrate too frequently between PMs. Generally, the initial VM allocation is determined by the static approach, and then the VM reallocation is preceded by the dynamic approach for the VM allocation problem.

Dong *et al.* [22] proposed a novel greedy algorithm based on an abstracted combination of the bin-packing problem (Best-Fit) and the quadratic assignment problem. It was proposed to meet multiple resource constraints, such as the physical server size (CPU, memory, storage, bandwidth, etc.) and network link capacity to improve resource utilization. This work reduces both the number of active physical servers and network elements to reduce energy consumption.

Cao *et al.* [23] proposed an energy-efficient scientific workflow scheduling algorithm to minimize energy consumption and CO2 emissions while satisfying a certain quality of service (QoS). This algorithm was designed for resource provision and allocation problems by applying the DVFS scheme to reduce energy consumption. The optimal frequency for executing each task was determined under the deadline constraint and resource utilization simultaneously. However, according to [21] and [24], DVFS implementations suffer from cubic time complexities $O(N^3)$, where N is the number of tasks.

Hadji *et al.* [25] proposed a minimum cost maximum flow algorithm for VM placement in cloud computing to serve multiple users and tenants with time-varying demands and workloads. Their algorithm was compared to an exact method that generalizes the classic bin-packing formulation using a linear integer program. This algorithm can seek a near-optimal solution that can achieve performance close to bin-packing algorithms that are optimal for virtual machine placement in virtualization enabled physical resources when the demand is known in advance [25]. The complexity of this algorithm is $O(min\{N^2 * flow, N^3 * f_{cost}\})$, where *flow* is the obtained flow on the graph and f_{cost} is the corresponding minimum cost. They stated that this complexity can be considered as low and negligible compared to the modified bin-packing problem. However, it is still expensive to use for dynamic VM allocation.

Ding *et al.* [26] proposed an energy-efficient scheduling algorithm for VMs using DVFS in cloud computing with heterogeneous physical machines considering the deadline constraint. This study focuses on the dynamic scheduling of virtual machines to achieve energy efficiency by determining the optimal frequency for a PM. To achieve these goals, they first found the VMs needed to be scheduled and allocated them to the PMs, and then computed the optimal frequency of each active core based on the sum of the required resources of the VMs on it. When some VMs are finished using this optimal frequency, this algorithm reconfigures the consolidated physical resources onto PMs or reallocates VMs to further reduce the energy consumption. However, it fully utilizes the computing resources of each active PM, and thus, an excessive number of PMs should be activated. This is also caused by the usage of the DVFS scheme because the lowered frequency by using idle time increases the execution time.

Guo *et al.* [27] proposed a game-based consolidation algorithm for VMs in cloud data centers with energy and workload constraints. This study focused on improving resource utilization and reducing the number of VM migrations simultaneously. They forecasted the resource load using gray theory to reduce the delay of load throttling and then attempted to reduce the number of online PMs to save energy consolidation.

Kim and Zeghlache [9] deployed compositional real-time computing and real-time virtual machine techniques to achieve real-time service on virtualized cloud resources. The compositional and hierarchical real-time framework [10], [11] enables a group of real-time applications to be a single real-time resource requirement for the upper layer of real-time environments. For a given real-time applications, they anayzed the required CPU utilization on the base machine. Thus, the real-time service can be guaranteed when the allocated virtual machine keeps providing the required amount of processing capacity by the deadline. Therefore, they modeled a real-time service as a *real-time virtual machines* (RT-VM) request and then applied the *dynamic voltage/frequency scaling* (DVFS) scheme to each RT-VM. In this paper, we will also use this definition of RT-VM to represent the requirements of real-time services in a cloud environment.

TABLE 2. A set of real-time tasks for an example.

Tasks	WCET	Period(=Deadline)	Utilization
τ_1	4	5	0.8
τ_2	5	10	0.5
τ_3	15	15	1.0
τ_4	10	20	0.5
τ_5	25	25	1.0
τ_{Idle}	5	25	0.2 (U_{Idle})

III. SYSTEM MODEL

A. TASK MODEL

A set $\mathbf{\Gamma}$ that contains N periodic real-time tasks, denoted by $\mathbf{\Gamma} = \{\tau_1, \tau_2, \ldots, \tau_N\}$, where the tasks are mutually independent, was considered. The task τ_i has a period T_i and worst-case execution time (WCET) C_i. It was assumed that the relative deadline D_i of τ_i is equal to T_i, that is, each τ_i has the *implicit deadline*. The utilization u_i of τ_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 τ_i at time t is denoted by $\tau_i(t)$. Each active job 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 $c_i(t)$. $d^{max}(t)$ is defined as the largest $d_i(t)$ of all active jobs at time t.

A set $\mathbf{B}$ that contains the current time t, all of the release times, and the absolute deadlines of jobs within the time interval $[t, d^{max}(t)]$ is defined, that is, $\mathbf{B} = \{b_0, b_1, \ldots, b_K\}$. b_k is called the *temporal boundary*, and it is assumed that $\mathbf{B}$ 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. REAL-TIME VIRTUAL MACHINE MODEL

We assume that *RT-VM* (*Real-time virtual machine*) is required to provide a real-time service, as in the existing work by Kim *et al.* [9]. RT-VM V_i is parameterized with three elements u_i, m_i, and $d_i(t)$ at time t, where m_i is the *million instructions per second* (*MIPS*) rate of the virtual machine. Without the loss of generality, we assume that u_i and $d_i(t)$ are the same as in the above definition, and m_i is the fixed rate for the specification of the base machine. For example, V_i must provide $u_i \times m_i$ processing capacity (i.e., c_i) by the deadline d_i when the virtual machine is allocated for real-time services. Note that this terminology will also be used in the experimental simulation of *CloudSim* [32]. More details about modeling real-time virtual machines can be found in a previous study [9].

IV. ALGORITHM FOR SOLVING THE OPTIMIZATION PROBLEM

In this section, we introduce the scheduling methodology that guarantees *RT-optimality* and minimizes active processing elements. We propose a newly structured flow network compared with our previous study, which simplifies the process of finding augmenting paths and a solver algorithm. We then show the computational complexity of the proposed algorithm. Note that the detailed process of finding augmenting paths using the previous flow network is described in the appendix.

A. CONSTRUCTING THE FLOW NETWORK

We propose a methodology for constructing a flow network that is designed to find a feasible solution (i.e., maximizing the actual flows) by visiting all edges only once. To limit the finding of additional augmenting paths, we need to control the use of idle time. Thus, we first divide an existing flow network containing both the flow of real-time tasks and processor idle time into two separate flow networks, where one contains the flow of real-time tasks and the other contains the flow of idle time. Figure 1 (a) and (b) show the flow network containing the flow of real-time tasks and its active job area for the example in Table 2. To limit the waste of idle time, a capacity constraint at every edge between the task and window nodes is newly defined as $cap(e(\tau_i, W_k)) = l_k \times U_i$ instead of $cap(e(\tau_i, W_k)) = l_k$, where all tasks are allowed to execute their own utilization. For example, τ_2 can have a maximum flow of $5 * 0.5 = 2.5$ within W_1 and can have a maximum flow of $5 * 0.5 = 2.5$ within W_2.

Figure 1 (c) and (d) show the flow network for idle time and its active job area, respectively, where I_k is the time interval allowed for idle time within $[b_k, b_{k+1}]$. The amount of flow at edges between left-side and right-side nodes in this flow network represents the idle time allowed for the real-time tasks within each window. This flow of idle time can be used to execute the real-time tasks over their allowed time based on its utilization within the window. If the flow of real-time task at $e(\tau_2, W_1)$ is 2.5 in Figure 1 (a) and the flow of idle time at $e(\tau_2, I_1)$ is 1 in Figure 1 (c), then the reserved execution time of τ_2 within W_1 is 3.5.

To find a feasible solution for real-time tasks and idle time simultaneously in a graph, two flow networks are combined, as shown in Figure 1 (e). In the combined flow network, the flow of real-time tasks and that of idle time can be managed together. At every boundary, the scheduling algorithm is invoked to reserve the execution time for all active jobs. Here, we formulate an optimization problem with a combined flow network, as in [12]. For convenience of description, we define four sets at the current time interval as follows:

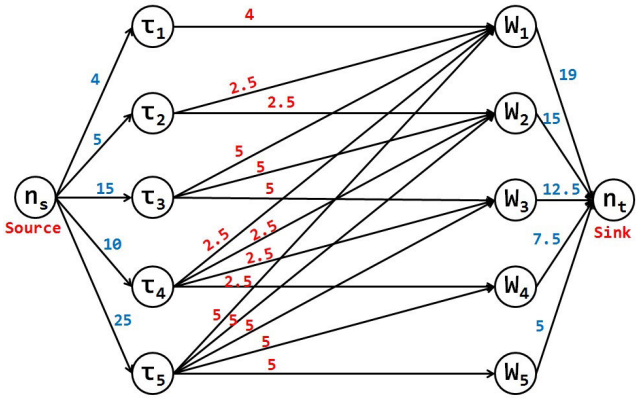
$$\boldsymbol{K_{RT}}(t_s, t_e) = \{k | W_k \subset [t_s, t_e]\}, \tag{1}$$

$$\boldsymbol{J_{RT}}(k, t) = \{i | W_k \subset [a_i(t), d_i(t)]\}, \tag{2}$$

$$\boldsymbol{K_I}(t_s, t_e) = \{k | I_k \subset [t_s, t_e]\}, \tag{3}$$

$$\boldsymbol{J_I}(k, t) = \{i | I_k \subset [a_i(t), d_i(t)]\}, \tag{4}$$

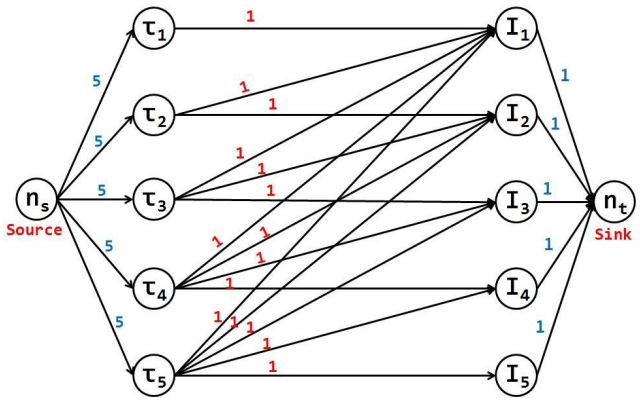
$\boldsymbol{K_{RT}}(t_s, t_e)$ and $\boldsymbol{K_I}(t_s, t_e)$ contain all of the indices k of the windows W_k and I_k, respectively, which are placed in the time interval $[t_s, t_e]$. $\boldsymbol{J_{RT}}(k, t_s)$ and $\boldsymbol{J_I}(k, t_s)$ contain all of the indices i of the active jobs at time t_s that are still active in W_k and I_k, respectively. Using these four sets, a flow network problem for scheduling real-time tasks was formulated as



(a) The flow network expressed real-time tasks

	W_1	W_2	W_3	W_4	W_5	Residual C_i
τ_1	$0 \le f \le 4$					4
τ_2	$0 \le f \le 2.5$	$0 \le f \le 2.5$				5
τ_3	$0 \le f \le 5$	$0 \le f \le 5$	$0 \le f \le 5$			15
τ_4	$0 \le f \le 2.5$	$0 \le f \le 2.5$	$0 \le f \le 2.5$	$0 \le f \le 2.5$		10
τ_5	$0 \le f \le 5$	$0 \le f \le 5$	$0 \le f \le 5$	$0 \le f \le 5$	$0 \le f \le 5$	25
Residual Cap(W_k)	19	15	12.5	7.5	5	Max. flow = 0

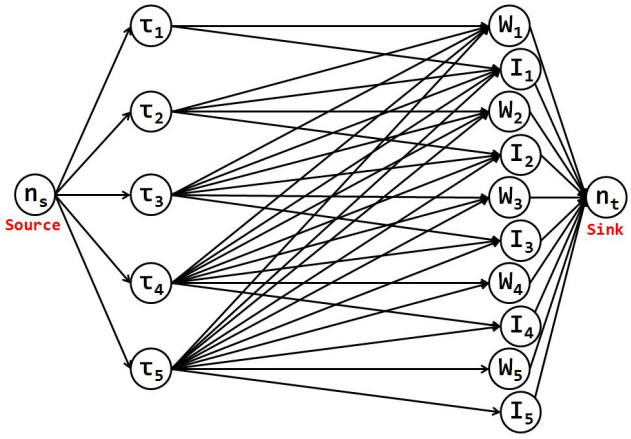
(b) The active job area expressed real-time tasks



(c) The flow network expressed idle time

	I_1	I_2	I_3	I_4	I_5	Residual Time
τ_1	$0 \le f \le 1$					5
τ_2	$0 \le f \le 1$	$0 \le f \le 1$				5
τ_3	$0 \le f \le 1$	$0 \le f \le 1$	$0 \le f \le 1$			5
τ_4	$0 \le f \le 1$	$0 \le f \le 1$	$0 \le f \le 1$	$0 \le f \le 1$		5
τ_5	$0 \le f \le 1$	$0 \le f \le 1$	$0 \le f \le 1$	$0 \le f \le 1$	$0 \le f \le 1$	5
Residual Cap(I_k)	1	1	1	1	1	Max. flow = 0

(d) The active job area expressed idle time



(e) The combined flow network expressed real-time tasks and idle time

FIGURE 1. The examples for combining real-time tasks and idle time in a flow network.

follows:

$$Maximize \sum_{\forall i} \sum_{\forall k} X_{i,k} \tag{5}$$

$$s.t. \sum_{\forall k \in \boldsymbol{K_{RT}}(t,d_i(t))} X_{i,k} \le c_i(t), 1 \le \forall i \le N \tag{6}$$

$$\sum_{\forall k \in \boldsymbol{K_I}(t,d_i(t))} X_{i,k} \le c_{Idle}(t), 1 \le \forall i \le N \tag{7}$$

$$\sum_{\forall i \in \boldsymbol{J_{RT}}(k)} X_{i,k} \le \left[\sum_{\forall i \in \boldsymbol{J_{RT}}(k,t)} C_i/T_i \right] \times l_k, 1 \le \forall k \le K \tag{8}$$

$$\sum_{\forall i \in \boldsymbol{J_I}(k)} X_{i,k} \le l_k \times U_{Idle}, 1 \le \forall k \le K \tag{9}$$

Algorithm 1 The Scheduling Algorithm for Dedicated Search

```
// Initially, G ← construct the combined graph including idle time
procedure Solver(G, t, curMode)
    if curMode is CF then
        τ_next ← A task had an earliest deadline
    else
        τ_next ← A task had an latest deadline
    end if
    while maxFlow is not Σ_i c_i(t) do
        if curMode is CF then
            Find an augmenting path using BFS from τ_next connected with windows of the front order
        else
            Find an augmenting path using BFS from τ_next connected with windows of the back order
        end if
        for Each edge e(u, v) in the augmenting path do
            Decrease the capacity of e(u, v) by bottleneck
            Increase the capacity of e(v, u) by bottleneck
        end for
        Increase maxFlow by bottleneck
        τ_next ← A task of the next order
    end while
    return {X_{i,1}|∀_i}, {X_{Idle,k}|∀_k}
end procedure
```

$$X_{i,k} \leq l_k \times U_i,\ 1 \leq \forall i \leq N \text{ and } 1 \leq \forall k \leq K \tag{10}$$

$$X_{i,k} \leq min(l_k \times U_{Idle}, l_k - l_k \times U_i),\ 1 \leq \forall i \leq N \text{ and } 1 \leq \forall k \leq K. \tag{11}$$

In $AJA(t)$, $X_{i,k}$ is the reserved execution time for τ_i within W_k and I_k. At a boundary, the corresponding AJA is established, and three types of constraints are defined as equations (5)-(11). Equation (5) is the objective function of the scheduling problem. Equation (6) indicates that each active job must complete its execution times within the allowed time interval $[a_i(t), d_i(t))$, which is called the *job completion constraint* (JCC). Equation (8) indicates 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). Equation (10) indicates that each active job within a given window does not simultaneously occupy more than one processor, which is called *no intra-task parallelism* (NIP). Equations (7), (9), and (11) indicate that each active job has a limitation for the use of idle time. In particular, the flow of idle time within each window is set to a value smaller than $l_k \times U_{Idle}$ and $l_k - l_k \times U_i$ in equation (11). This constraint indicates that the flow of idle time is upper-bounded by the length of its window. After a feasible solution is obtained from this scheduling problem, the amount of flow within W_1 and I_1 is used to allocate the computational resources to each active job for their execution. Therefore, the formulation (5)(11) corresponds to the combined flow network, as shown in Figure 1 (e). The amount of maximum flow from n_s to n_t is assumed to be $\sum_i c_i(t)$. If the maximum flow $\sum_i c_i(t)$ is found in the flow network, it is interpreted as a feasible schedule for $AJA(t)$. Then, the method of allocating $X_{i,1}$ to the processors within W_1 can easily be determined, for example, using McNaughton wrap around the algorithm in [36].

B. A SOLVER ALGORITHM THAT REGULATES THE FLOW

We propose a solver algorithm to find a feasible solution by visiting each edge of the given flow network only once, which is expressed in Algorithm 1. This algorithm is designed for the flow network constructed in the previous section and is named *the scheduling algorithm using dedicated search* (*DSEA*). DSEA is also capable of clustering the idle time to initiate a deeper low-power state to save static energy. For this, we use two modes, `ClusterBackward` (`CB`) and `ClusterForward` (`CF`), either to save idle times for later use or to cluster idle times for staying in a deeper low-power state for a long time. Specifically, the solver algorithm clusters the idle time close to the end of either the last boundary (b_k) or the current time (b_0) when the current mode is `CB` or `CF`, respectively. On the other hand, the previous algorithm in [12] assigned a parameter *cost* based on the current mode to each edge to prioritize a certain flow over the flow network, and its feasible solution was found by using the *min-cost-max-flow* solver algorithm. However, because of the high time complexity of the *min-cost-max-flow* solver, it is difficult to dynamically schedule real-time tasks during system runtime.

Thus, we apply a method that does not use the *cost* to the proposed solver algorithm to find a feasible solution close

	W_1	I_1	W_2	I_2	W_3	I_3	W_4	I_4	W_5	I_5	Residual C_i
τ_1	4	0									0
τ_2	2.5	1	1.5	0							0
τ_3	5	0	5	0	5	0					0
τ_4	2.5	0	2.5	1	2.5	1	0.5	0			0
τ_5	5	0	5	0	5	0	5	0	5	0	0
Residual Cap(W_k)	0	0	1	0	0	0	2	1	0	1	Max. flow = 56

(a) The computed flow based on CF

	W_1	I_1	W_2	I_2	W_3	I_3	W_4	I_4	W_5	I_5	Residual C_i
τ_1	4	0									0
τ_2	2	0	2.5	.5							0
τ_3	5	0	5	0	5	0					0
τ_4	0	0	2.5	.5	2.5	1	2.5	1			0
τ_5	5	0	5	0	5	0	5	0	5	1	0
Residual Cap(W_k)	3	1	0	0	0	0	0	0	0	0	Max. flow = 56

(b) The computed flow based on CB

FIGURE 2. The actual example of the amount of flow in the active job area.

to the solution from the *min-cost-max-flow* problem by the previous algorithm in [12]. In the combined flow network with idle time shown in the previous subsection, a feasible solution can be found in any order for all edges, so there is no need to control the flow using *cost*. To cluster the flow of several tasks at the beginning of *AJA*, the solver first searches the edges connected to that task and sends the flow as much as possible. Algorithm 1 is based on the *Edmonds-Karp* algorithm using breadth first search (BFS), except for flags CF and CB for clustering idle time. In line 3 of Algorithm 1, the BFS selects the starting node based on *curMode* to find the augmenting path. If *curMode* is CF, then the task is selected based on the earliest deadline first and vice versa. In lines 10 and 12, this algorithm finds the augmenting path using BFS from τ_{next} connected with windows of the front or back order. In line 14-17, it searches edges in the augmenting path and modifies the capacity of edges by *bottleneck* capacity, where *bottleneck* is the minimum capacity of any edge on the path. Finally, *maxFlow* is added by this amount of *bottleneck* in line 18. The procedure for finding the augmenting path will continue until *maxFlow* is equal to the summation of WCET for real-time tasks in line 8.

1) (EXAMPLE)

Figure 2 (a) and (b) show the computed flow clustering the real-time tasks and idle time based on CF and CB using Algorithm 1. These feasible solutions were found by visiting each edge only once. In Figure 2 (a), the solver fills out the flow at edges connected with windows of the front order based on CF. A part of the search order is $\{e(\tau_1, W_1), e(\tau_1, I_1), e(\tau_2, W_1), \ldots, e(\tau_5, W_5), e(\tau_5, I_5)\}$. As a result, real-time tasks are clustered close to the beginning of AJA and the idle time is clustered close to the end of AJA. In Figure 2 (b), the solver fills out the flow at edges connected with windows of the back order based on CB. A part of the search order is $\{e(\tau_5, W_5), e(\tau_5, I_5), e(\tau_5, W_4), \ldots, e(\tau_1, W_1), e(\tau_1, I_1)\}$. As a result, real-time tasks are clustered close to the end of AJA and the idle time is clustered close to the beginning of AJA.

C. COMPLEXITY

The proposed solver does not require traditional solvers for the max-flow and min-cost-max-flow problems used in [12]. The existing solver in [12] incurs a high time complexity of $\Omega(N^3 logN)$. On the other hand, the proposed solver has a reduced complexity of $O(N^2)$. In the combined flow network, several window nodes for the idle time and edges connected to the nodes are added. As a result, $|E|$ is computed as the summation of $|\{e(n_s, \tau_i)|\forall_i\}| = N$, $|\{e(\tau_i, W_k)|\forall_{i,k}\}| = (N+1)(N+2)/2 = (N^2+3N+2)/2$, $|\{e(\tau_i, I_k)|\forall_{i,k}\}| = (N+1)(N+2)/2 = (N^2+3N+2)/2$, $|\{e(W_k, n_e)|\forall_k\}| = N+1$, and $|\{e(I_k, n_e)|\forall_k\}| = N+1$. In summary, $|E|$ is N^2+6N+4. Because our solver finds a feasible solution by visiting each edge only once, its complexity is $O(|E|) = O(N^2)$.

D. PROOF FOR LOWER COMPLEXITY OF THE PROPOSED ALGORITHM

The proposed algorithm generates an *unfair-but-optimal* schedule to satisfy the deadlines of real-time tasks and

simultaneously control the use of idle time. More specifically, Algorithm 1 can unfairly assign the reserved execution time based on the current mode CB or CF to active jobs at each boundary. Nonetheless, it ensures the optimality of real-time tasks within the range of AJA by finding a feasible solution from the flow network problem. In a previous study [12], we formulated a scheduling problem for real-time tasks with the permitted resource capacity at each boundary to find a feasible solution from all possible solutions. Because of the wide scope of problems, there is a limitation of high time complexity, which is one of the motivations for the current study. In the above sections, we introduce a methodology for alleviating computational complexity. Thus, we focus on the proof of finding an optimal solution with lower complexity than the previous study in this section. A proof of the optimality for real-time tasks using flow network models can be found in [12] and [13].

Lemma 1: Algorithm 1 finds a solution for maximizing the flow of real-time tasks over the flow network, as shown in Figure 1 (a), which results in a fairness schedule.

Proof: We will show the proposed algorithm finds a feasible solution within the specific scope of problems instead of all possible problems addressed in the previous study. First, Algorithm 1 ensures the *boundary fair* for real-time tasks by using the graph constructed with the minimum capacity of resources, as shown in Figure 1 (a). According to [40], a schedule is *boundary fair* if and only if the absolute value of the allocation error for any task τ_i at any boundary time b_k is less than one time unit. The allocation error of τ_i at boundary time b_k is defined as the difference between $b_k * u_i$ and the time unit allocation to τ_i before b_k in a schedule. In Figure 1 (a), the permitted processing capacity of the k-th window is limited to $b_k * u_i$ for each τ_i using Equation (8). Thus, the allocation error of τ_i is always less than one time unit at each boundary. Consequently, this ensures that *fairness* remains for real-time tasks. □

Lemma 2: In Figure 1 (e), the amount of flow associated with the vertices and edges, which is expressed as the use of idle time, does not affect the fairness of real-time tasks.

Proof: In the same way, the use of idle time within the range of AJA is also limited by Equations (9) and (11) instead of allowing all remaining capacity at each boundary to keep the fairness of real-time tasks. Then, the combination of the solution maximizes the flow of real-time tasks from Figure 1 (a), and the solution maximizes the flow of the use of idle time from Figure 1 (c) is used as the reserved execution time to generate the final schedule for tasks. Although the reserved execution time of each active job seems *unfair* because of the additional idle time, it does not affect *fairness* for real-time tasks because the idle time is also fairly used within AJA. This is expressed as the combined flow network for real-time tasks and idle time, as shown in Figure 1 (e). With that of the flow network, the proposed algorithm can find a feasible solution for real-time tasks and idle time simultaneously. □

Lemmas 1 and 2 show that the feasible solution still ensures *RT-optimality* even in the narrowed problem scope. Then, we will show that Algorithm 1 can find that of the solution with low complexity expressed as $O(N^2)$ in the previous section. In fact, the purpose of constructing the combined flow network, as shown in Figure 1 (e) is to prevent the procedure for finding additional augmenting paths. Therefore, Algorithm 1 can find a feasible solution by visiting edges in $\{e(\tau_i, W_k)|\forall_{i,k}\}$ only once.

For proof, let f_{CF} be the cumulative amount of *maxFlow* in Algorithm 1 and f_{MAX} be the theoretical maximum amount of flow, that is, $f_{MAX} = \sum_i c_i(t)$.

Lemma 3: After Algorithm 1 is finished searching $e(\tau_i, W_k)$ only once, where $\forall_{i,k}$, f_{CF} is the same as f_{MAX}.

Proof: For a proof by contradiction, the opposite of Lemma 3 can be expressed as $f_{CF} < f_{MAX}$ after the completion of Algorithm 1. This means that the proposed algorithm does not find the maximum flow over the given flow network by searching all edges in $\{e(\tau_i, W_k)|\forall_{i,k}\}$ only once.

First, we start with an explanation based on graph theory for the process of Algorithm 1 to find the maximum flow. Algorithm 1 finds an augmenting path in a residual graph using BFS in lines 9 and 11. The residual graph indicates the amount of flow allowed at each edge in the flow network. The amount of flow allowed in the residual graph is determined by the *bottleneck* in the augmenting path, and then it is added to *maxFlow*. If an edge in the original graph is saturated, that is, the capacity of the edge in the residual graph becomes zero, then the corresponding edge is omitted from the residual graph as it cannot admit any more flow. In other words, the flow becomes the maximum if there are no augmenting paths. In this process, the bottleneck is determined by the change in the allowed capacity at the edges, and various augmenting paths are found accordingly. In a previous study, the capacity of $e(\tau_i, W_k)$ was set as the length of each window to consider all possible feasible solutions. These are expressed as the following equations related to the capacities of $e(n_s, \tau_i)$ and $e(W_k, n_t)$. In the following equations, $Cap'(edge)$ is the capacity of *edge* and $e'(vertex, vertex)$ is the *edge* between two vertices, which were defined in a previous study.

$$Cap'(W_k) \leq \left[\left(\sum_{\forall i \in \boldsymbol{J_{RT}}(k,t)} C_i/T_i \right) + U_{Idle} \right] \times l_k \quad (12)$$

$$\sum_{\forall k \in \boldsymbol{J_{RT}}(k,t)} Cap'(e'(\tau_i, W_k) = \sum_{\forall k} l_k \quad (13)$$

On the other hand, the proposed algorithm finds a feasible solution from the narrowed problem scope because it considers the graph constructed with the minimum capacity that ensures fairness for real-time tasks. The corresponding capacity is expressed as follows:

$$Cap(W_k) \leq \left[\sum_{\forall i \in \boldsymbol{J_{RT}}(k,t)} C_i/T_i \right] \times l_k \quad (14)$$

$$\sum_{\forall k \in \boldsymbol{J_{RT}}(k,t)} Cap(e(\tau_i, W_k) = \sum_{\forall k} C_i \tag{15}$$

Using these equations, the proposed algorithm assigns the flow as much as possible within the capacity of the corresponding edges to maximize the flow at each iteration. Then, because the capacity of $e(\tau_i, W_k)$ becomes the bottleneck in most cases, the residual capacity of the edges is depleted, and the corresponding edges in the residual graph are omitted as a result. On the one hand, there is a case where the residual capacity of $e(\tau_i, W_k)$ is not depleted. This occurs when the remaining execution time of the active job is less than the worst-case execution time. Then, the bottleneck becomes $e(n_s, \tau_i)$ in this case, and the corresponding residual capacity is depleted before visiting the edges in $\{e(\tau_i, W_k)|\forall_k\}$. Because $e(n_s, \tau_i)$ is omitted, the corresponding edges in $\{e(\tau_i, W_k)|\forall_k\}$ are also omitted in the residual graph.

However, a time instance in which all tasks release jobs simultaneously is the worst case, as all edges in $\{e(\tau_i, W_k)|\forall_{i,k}\}$ must be visited once. In this case, the residual capacity is depleted when each edge is visited once, and the corresponding edges are omitted from the residual graph. This means that the augmenting path no longer exists in the residual graph after visiting all $e(\tau_i, W_k)$ once, and then f_{CF} becomes f_{MAX}. Because there is a counter example to the opposite of Lemma 3 is true by a proof by contradiction. □

From Lemma 1 to 3, we can derive the following theorem for Algorithm 1:

Theorem 1: Algorithm 1 generates an unfair-but-optimal schedule for real-time tasks in $O(N^2)$ from the combined flow network, as shown in Figure 1 (e).

Proof: From Lemmas 1 and 2, we showed the proposed algorithm ensures the optimality of real-time tasks even if it generates an unfair schedule for real-time tasks and idle time because of the use of idle time. In other words, Algorithm 1 finds the feasible solution in the given flow network shown in Figure 1 (e), and then generates an unfair schedule for tasks. In line 7 of Algorithm 1, the proposed algorithm iterates the procedure for finding the feasible solution until *maxFlow* (f_{CF}) becomes $\sum_i c_i(t)$ (f_{MAX}), which maximizes the amount of flow. From Lemma 3, we show that f_{CF} becomes f_{MAX} after the algorithm visits all edges only once in the flow network. Because the number of edges $|E|$ is N^2+6N+4, as computed in sub-section *D. Complexity*, its complexity is $O(N^2)$. Thus, Algorithm 1 generates an unfair schedule for real-time tasks in $O(N^2)$. □

V. EXPERIMENT

We proposed a real-time task scheduling algorithm for multiprocessors, which was designed to reduce static energy consumption with reduced computational complexity. To demonstrate its experimental performance, we used the following two metrics:

- Time complexity
  To evaluate the proposed algorithm in the first metric, we measured the elapsed time required to find a feasible solution in comparison with a traditional solver algorithm. The time taken by constructing a graph and sorting tasks by a deadline is excluded from the measurement of time because they are the same in both algorithms. Because the complexities of both algorithms for finding a feasible solution have an exponential relationship with N, we measured the elapsed time while varying N. Through this experiment, we show that the proposed solver can find a feasible solution in less time than the traditional solver.
- Static energy efficiency
  The static energy consumption can be reduced by using a decreasing number of processing elements (PEs) that must be in an active mode (i.e., powered-on). Thus, we compared the number of active PEs required for the given set of real-time tasks in this experiment. To clearly see the difference, we used a set of real-time tasks with heavy or lightweight utilization. Through this experiment, we show that the scheduling algorithm with the proposed solver is energy-efficient in a cloud environment.

A. EXPERIMENTAL ENVIRONMENT

1) (TIME COMPLEXITY)

For the first metric in the above section, we compare the proposed solver, including the construction of a graph, to a traditional solver used in the previous algorithm. We referred to the source code written in the Python code in [31] to simulate the traditional solver. On the other hand, even if the procedures for constructing a graph from both algorithms are different, they are excluded from the measurement of time because their complexities are the same as $O(N^2)$. Thus, only the time taken to find a feasible solution after constructing the graphs was measured. More specifically, the measuring time starts immediately after the construction of a graph and then ends when the maximum flow is found.

We generated 1,000 sets of real-time tasks for each utilization value ranging from {4, 8, 16, 24, 32, 40, 48, 96}. The period of each task was randomly chosen using the uniform distribution in the interval between 1 ms and 100 ms. Then, the WCET was randomly chosen as a value from 0.1 to 1.0 times its period. The experiment was conducted in a system with an Intel(R) Core (TM) 2 Duo CPU E8400 @ 3.00 GHz, 4GB RAM, and an Intel SSD SC2CT240A3 ATA Device. Finally, through this simulation, we show that the proposed solver can find a feasible solution based on both `CF` and `CB`.

2) (STATIC ENERGY EFFICIENCY)

For the second metric in the previous section, we compare the scheduling algorithm using the proposed solver with *sEDF* in Xen. sEDF calculates the number of active PEs for a given set of real-time tasks based on task allocation in accordance with the utilization of each task. Note that the *RTDS* scheduler can also be comparable, but the task allocation

in RTDS works the same as sEDF at the critical instant, that is, the time at which all tasks are released at the same time.

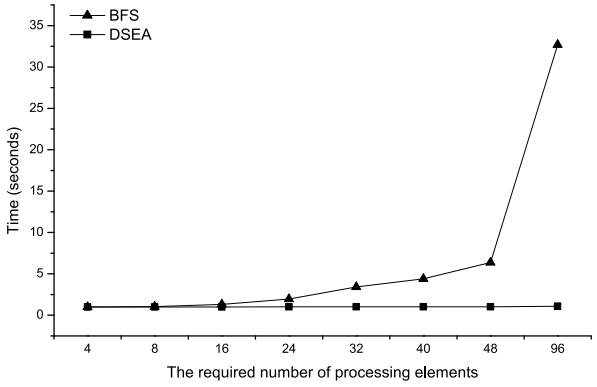
The experiment uses a simulation tool called *CloudSim* [32], which provides a simulated environment in which users can manage services and model cloud infrastructure. This is a commonly used simulation tool because it is difficult to conduct large-scale systems such as data centers or super-computing platforms. The task generation methodology is the same as that used for evaluating the first metric, time complexity. In addition, we generated 20,000 sets of real-time tasks classified into 10,000 sets of tasks consisting of light utilization of a task ($U \leq 0.5$) and 10,000 sets of tasks consisting of heavy utilization of a task ($U > 0.4$). The given task set is assigned to PEs as RT-VMs (similar to Kim *et al.* [9]). To accurately measure the required number of PEs according to the given task set, we assumed that a cloud server has sufficient computing resources. In other words, it is assumed that there are enough available PEs to accommodate all the given task sets. Finally, we show that the scheduling algorithm using the proposed solver uses fewer PEs and consumes less static energy than the sEDF.

B. EXPERIMENTAL RESULT

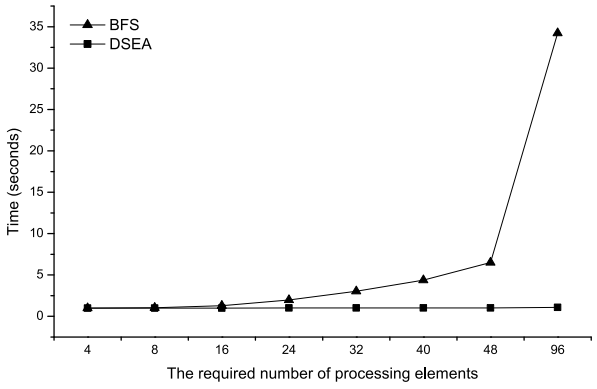
In all figures for this section, the x-axis is set as the required number of processing elements (or the total utilization to be the reference point). This was used as the reference point in the generation of the task sets. For example, if it is set to 16, then the task generator creates a task until the total utilization of tasks becomes [15, 16). Thus, when it is large, N becomes exponentially large.

1) (TIME COMPLEXITY)

We assume *BFS* (*breadth-first search*) as a counterpart because it is used in the latest algorithm [12]. Figure 3 (a) and (b) show the elapsed times when `ClusterForward` and `ClusterBackward` are applied to the proposed algorithm, respectively. The elapsed time is the average time required to find a feasible solution for each set of tasks. When the required number of processing elements is low, the elapsed times are almost the same because N is very small. However, as the required number of processing elements increases, the difference between both algorithms increases in the elapsed time. In particular, when the required number of PEs is 96, *BFS* is 30 times longer than *DSEA*. This gap is caused by the difference between their complexities as $O(N^3)$ for *BFS* and $O(N^2)$ for *DSEA*. The minimum and maximum number of tasks (i.e., N) from the generated set of tasks based on 96 were 154 and 190, respectively. We did not experiment when the required number of PEs was more than 96 because it took almost 10 h to find a feasible solution for 1, 000 task sets even if the number of PEs was 96. Through this experiment, we showed that *DSEA* can find a feasible solution based on both `CF` and `CB` in less time complexity than *BFS*.



(a) The elapsed time when `ClusterForward` is applied

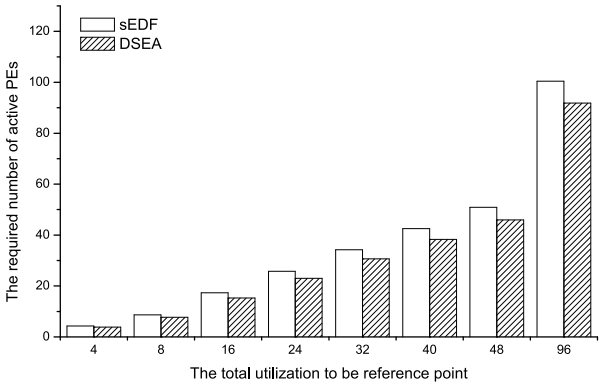


(b) The elapsed time when `ClusterBackward` is applied

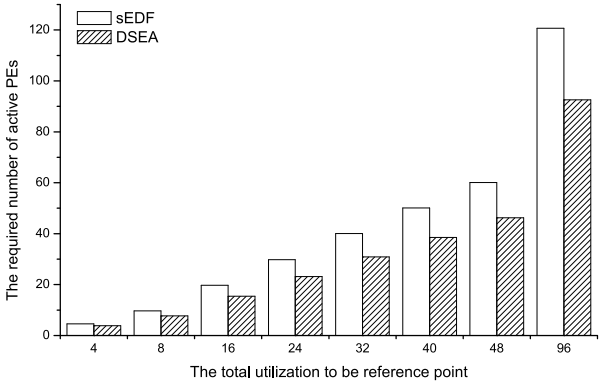
FIGURE 3. The elapsed time for finding a feasible solution.

2) (STATIC ENERGY EFFICIENCY)

Figure 4 (a) and (b) show the required number of active PEs to accommodate the given light- and heavy-weight task sets, respectively. In Figure 4 (a), *sEDF* continuously allocates RT-VM unless the total utilization of a PM exceeds 1.0 according to the utilization of the task. Thus, as *sEDF* is more likely to allocate two or more RT-VMs to a PM for the light-weight task set, the required number of active PEs can be low. To quantitatively see the evaluating result, Figure 5 shows the required number of active PEs normalized to the total utilization. In this figure, the dotted line represents a continuous value as the average number of required active PEs for all task sets. The solid line represents a discrete value as the actual number of required active PEs as the number of PEs must be an integer. In Figure 5 (a), this means that *sEDF* requires about 10% more PEs to be active mode than *DSEA* for all total utilization. In contrast, when the heavy-weight task set is used, as shown in Figure 4 (b), *sEDF* begins to rapidly increase the required number of active PEs. This is because the utilization of a heavy-weight task is set as greater than 0.4 and then *sEDF* can allocate at most two RT-VMs in a PM. Thus, in Figure 5 (b), *sEDF* required about 20% more

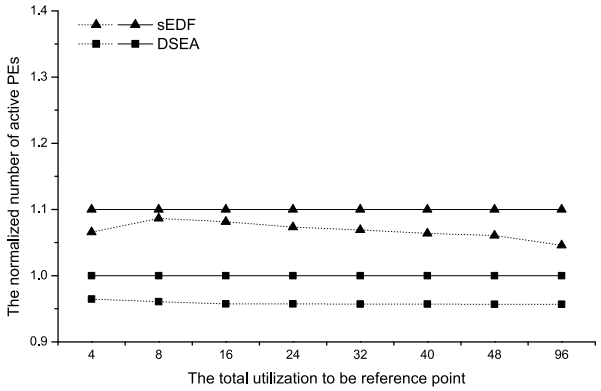


(a) The required number of active PEs for light-weight utilization

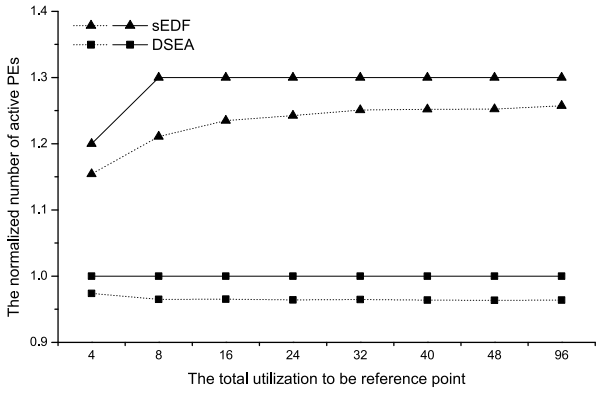


(b) The required number of active PEs for heavy-weight utilization

FIGURE 4. The required number of active processing elements.



(a) The normalized number of active PEs for light-weight utilization



(b) The normalized number of active PEs for light-weight utilization

FIGURE 5. The normalized number of required active processing elements.

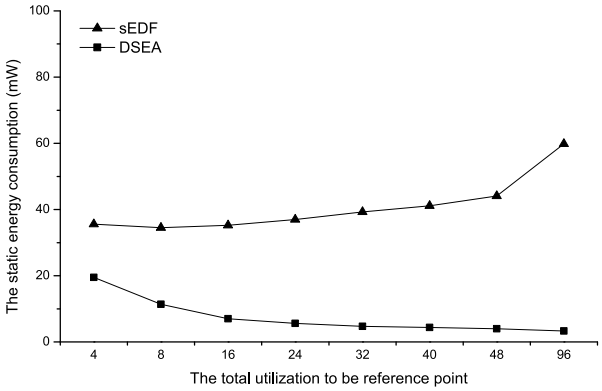
active PEs when the total utilization is 4 and about 30% more active PEs in the rest.

However, we have already seen in [12] that *DSEA* guarantees RT-optimality on multiprocessors. Therefore, *DSEA* needs as many active PEs as the ceiling of the total utilization, even if the total utilization increases. This feature can be seen more clearly in terms of the static energy efficiency. Figure 6 shows the energy model in which the ratio between the dynamic and static energy consumption is 75% : 25%. As the required number of active PEs is increased by *sEDF*, as shown in Figure 4 and 5, the static energy consumption of *sEDF* is also significantly increased, as shown in Figure 6. In contrast, *DSEA* consumes less static energy than *sEDF* because it only activates PEs as necessary. In addition, as the total utilization increases, the static energy consumption by *DSEA* becomes very low because active PEs are used at almost 100%. In other words, *DSEA* has made resource utilization to 100% with a small number of active PEs. This is also because the actual case execution time is assumed to be the same as the worst-case execution time. However, even if the actual case execution time is less than the worst-case execution time, *sEDF* does not reduce the static energy consumption because it does not consider dynamic power management. For a comparison with the initial number of active PEs in Figure 6, we excluded the additional static energy savings achieved from transitioning to a deeper low-power state by clustering idle time. Despite this, the scheduling algorithm with *DSEA* uses fewer PEs than *sEDF*; therefore, *DSEA* consumes less static energy than *sEDF*.

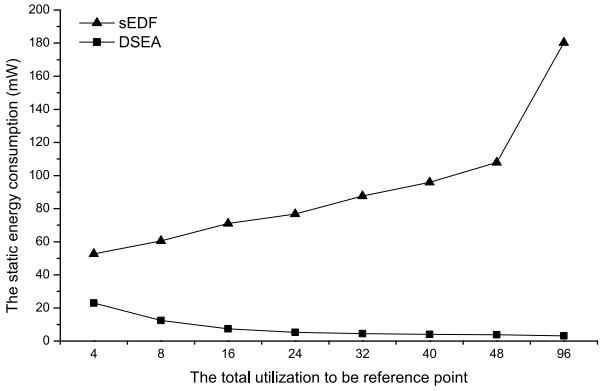
VI. DISCUSSION AND FUTURE WORKS

A. TRADE OFF BETWEEN COMPUTATIONAL COMPLEXITY AND ENERGY EFFICIENCY

There is a trade-off between the computational complexity and energy efficiency of the proposed algorithm. For example, the previous work in [12] clusters the idle time greedily to achieve high energy savings. To obtain an opportunity to cluster the number of idle times as much as possible, it has to continuously find the augmenting paths at every node over the given flow network. In other words, the previous algorithm attempts to find a feasible solution that includes the largest clustered idle times among many possible solutions using



(a) The static energy consumption for light-weight utilization



(b) The static energy consumption for heavy-weight utilization

FIGURE 6. **The static energy consumption within first boundary.**

a *min-cost-max-flow* problem. The procedures for achieving energy savings simultaneously cause high computational complexity. This may be acceptable in multiprocessor systems, but it makes it difficult to use the scheduling algorithm for cloud environments where the number of services is high.

On the other hand, the algorithm proposed in this paper finds a feasible solution among a few possible solutions that can be easily found by limiting the use of idle times. It proposed the use of a new flow network and a dedicated solver algorithm that can find a feasible solution by visiting edges only once. This algorithm has to search for a feasible solution among a few possible solutions, but it has only $O(N^2)$ computational complexity. In addition, this feasible solution also expects high energy savings because calculating the minimum number of active PEs and clustering idle times to initiate a deeper low-power state are still available. This has already been observed in the simulation results in the previous section.

B. LIVE VM MIGRATION

At the beginning of this study, we assumed that the temporal overhead caused by migrating VMs between different PMs is negligible by using *live VM migration*. Live VM migration is one of the major primitive schemes for managing virtualized platforms, such as data centers and cloud environments. It can synchronize the contents of memory, including the context of the CPU between the source and destination VMs. According to [33], live VM migration can maintain network connections and the application state during the process, thereby effectively providing seamless migration from a user's point of view. In the live VM migration scheme, two techniques are commonly used to synchronize the states of memory, which are called *pre-copy* and *post-copy*. These technologies can be classified according to the time required to copy memory. On the other hand, both technologies have a specific time interval to stop the VM and send the CPU state, which is called *downtime*. According to [34], at the minimum, this includes the transfer of the processor state. For pre-copy, this transfer also includes any remaining dirty pages. For post-copy, this includes other minimum execution states, if any, needed by the VM to start at the target. Even if one of the objectives in pre-copy and post-copy is to make a near-zero downtime, it can be critical in real-time computing. Thus, we believe that this downtime should be considered in the real-time scheduling problem for cloud environments and plan to proceed in future work.

VII. CONCLUSION

In this study, we proposed an energy-efficient scheduling algorithm for cloud environments that guarantees an optimal schedule for real-time tasks and achieves energy savings simultaneously. This algorithm was expanded from our previous work based on multiprocessors, which can reduce the total power consumed by unnecessarily activated processing elements and reduce the static energy consumption. To expand from multiprocessor systems to cloud environments, we analyzed several practical issues that need to be solved in our previous algorithm. Among these issues, the high time complexity of the previous algorithm, that is, $\Omega(N^3 logN)$, where N is the number of tasks, is the most critical. Because it makes it difficult to dynamically schedule tasks during system runtime, the time complexity must be alleviated. In addition, this can be a more critical problem in cloud environments that handle a large number of cloud services. To resolve this issue, we found that the most time-intensive part of the previous algorithm is the traditional solver that finds the solution of the given flow network. Thus, we propose a methodology for constructing a new structure of the flow network and designing a dedicated solver. With the newly constructed flow network, the proposed solver only takes time $O(N^2)$ to find a feasible solution. We showed the alleviated time complexity of the proposed solver by measuring the elapsed time to find a feasible solution in the experiment. As a result, we presented the scalability of the proposed algorithm for application on a cloud environmental platform in terms of time complexity. Finally, we present the experimental results for static energy efficiency through a simulated cloud environment.

APPENDIX A
LIMITATION OF THE PREVIOUS ALGORITHM

In this appendix, we show the procedure of the previous algorithm, that is, fnDPM-fw and fnDPM-cw in [12], to identify their limitations.

1) Formulate an optimization problem for scheduling real-time tasks.

$$Maximize \quad \sum_{\forall i}\sum_{\forall k} X_{i,k} \tag{16}$$

$$s.t. \quad \sum_{\forall k \in \boldsymbol{K}(t,d_i(t))} X_{i,k} \le c_i(t),\ 1 \le \forall i \le N \tag{17}$$

$$\sum_{\forall i \in \boldsymbol{J}(k)} X_{i,k} \le Cap(W_k),\ 1 \le \forall k \le K \tag{18}$$

$$X_{i,k} \le l_k,\ 1 \le \forall i \le N \text{ and } 1 \le \forall k \le K. \tag{19}$$

$$\boldsymbol{K}(t_s, t_e) = \{k | W_k \subset [t_s, t_e]\}, \tag{20}$$

$$\boldsymbol{J}(k) = \{i | W_k \subset [a_i(t), d_i(t)]\}, \tag{21}$$

$$Cap(W_k) = \left[M - \sum_{\forall i \notin \boldsymbol{J}(k,t)} C_i/T_i \right] \times l_k. \tag{22}$$

where $\boldsymbol{K}(t_s, t_e)$ contains all the indices k of the window W_k that are placed in the time interval $[t_s, t_e]$. $\boldsymbol{J}(k)$ contains all the indices i of the active jobs at time t_s, which are still active in W_k. $Cap(W_k)$ is the processing capacity required to execute active jobs within W_k.

2) Construct a flow network that is a directed and capacitated graph $\mathbf{G} = (\mathbf{V}, \mathbf{E})$ where $\mathbf{G}$ contains a set of nodes $\mathbf{V}$ and a set of edges $\mathbf{E}$ as follows.

$$\mathbf{V} = \{n_s, n_t\} \cup \{\tau_i | \forall_i\} \cup \{W_k | \forall_k\} \tag{23}$$

$$\mathbf{E} = \{e(n_s, \tau_i) | \forall_i\} \cup \{e(\tau_i, W_k) | \forall_{i,k}\} \cup \{e(W_k, n_t) | \forall_k\} \tag{24}$$

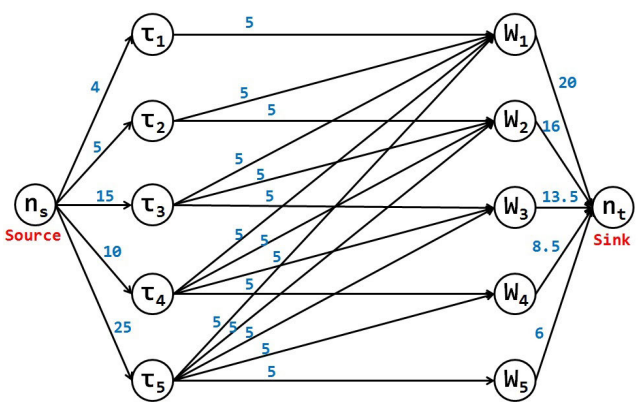
where the nodes are named after tasks τ_i and windows W_k. n_s and n_t denote the source and sink nodes, respectively. $e(n_1, n_2)$ denotes the edge from node n_1 to node n_2, and the actual flow $f(n_1, n_2)$ is assumed to be sent along the edge. The amount of maximum flow from n_s to n_t is assumed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

3) Assign a weight to each edge for clustering idle time (i.e., flow for idle time) according to user defined flag called `PushForward` or `PushBackward`.
4) Find a feasible solution which satisfies the above constraints and maximizes the summation of all flow from the flow network.
5) Repeat the procedure at every boundary.

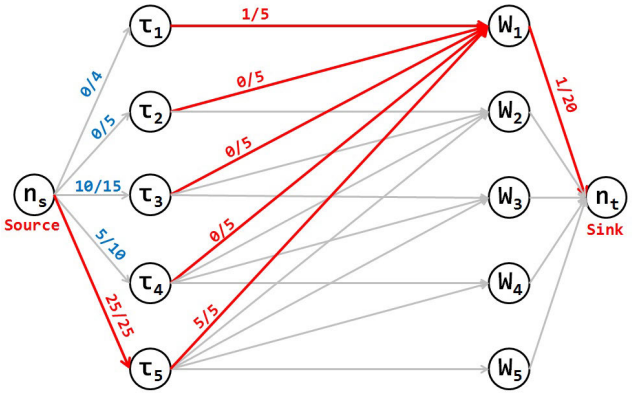
In step 4, the previous algorithms use existing solvers to find a feasible solution from the formulated *minimum cost and maximum flow* problem (*min-cost-max-flow*). The computational complexity of the solver was much higher than that of the other steps. Thus, the complexity of the solvers was dominant in both algorithms. For the *min-cost-max-flow* problem, the solver introduced by Orlin [37] is used. This solver is *an enhanced capacity-scaling algorithm*, and it comprises an $O(|V|log|E|SP_+(|V|, |E|))$ complexity, where $SP_+(|V|, |E|)$ denotes the time complexity of solving the single-source shortest path problem. *Dijkstra's algorithm with Fibonacci heaps* is known to provide an $O(|E| + |V|log|V|)$ bound for $SP_+(|V|, |E|)$ in [38]. For the *maximum flow* problem (*max-flow*), a strongly polynomial solver with $O(|V||E|)$ complexity [39] has recently been used. Normally, the previous algorithms consider a maximum of $N + 1$ windows in the time interval $[t, d^{max}(t)]$. Thus, its $|E|$ is the summation of $|\{e(n_s, \tau_i)|\forall_i\}| = N$, $|\{e(\tau_i, W_k)|\forall_{i,k}\}| = (N + 1)(N+2)/2 = (N^2+3N+2)/2$, and $|\{e(W_k, n_t)|\forall_k\}| = N+1$. In addition, the number of nodes $|V|$ was N. As a result, the computational complexity of the previous algorithms is $\Omega(N^3 logN)$. More details about the computational complexities are provided in [12].

The possibility of reducing the complexity lies in the fact that the specific part of the solver is repeatedly and redundantly executed. To identify the time-consuming part, we consider a graph $\mathbf{G} = (\mathbf{V}, \mathbf{E})$. $\mathbf{G}$ contains an edge $e = (u, v)$, where u and v are nodes. $cap(u, v)$ and $f(u, v)$ denote the capacity and flow of (u, v), respectively. In $\mathbf{G}$, we want to find the maximum flow from source node n_s to sink node n_t. For example, the *Ford-Fulkerson* algorithm can be used to solve the *max-flow* problem based on the idea of augmenting path [29], and it has two main steps. The first is a labeling process that searches for a flow augmenting path, that is, a path from n_s to n_t for which $f < cap$ along all forward edges and $f > 0$ along all backward edges. If this step finds a flow augmenting path, the second step changes the flow accordingly. *Max-flow* is found if and only if there is no augmenting path in the residual network. In other words, the solver must attempt to find an augmenting path to maximize the summation of flows.

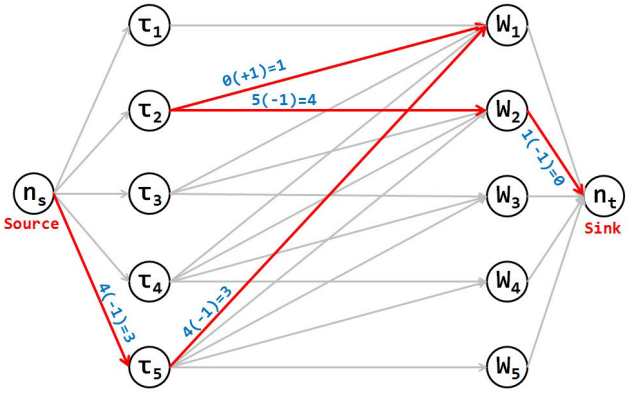
To find an augmenting path, *depth-first search* (DFS) and *breadth-first search* (BFS) are normally used in the *Ford-Fulkerson* [29] and *Edmonds-Karp* [30] algorithms, respectively. DFS is an edge-based technique that uses a stack data structure. This search algorithm first visits the source node and explores its branch node before backtracking. BFS is a vertex-based technique that uses a queue data structure. This search algorithm also visits the source node, but first explores its adjacent node before backtracking. Their time complexities are the same as $O(|V| + |E|)$, because every node and every edge will be explored in the worst case. Thus, the complexity of our algorithm for the *max-flow* problem is $N * O(|V| + |E|) = O(N^3)$, where $|V|$ is expressed as N and $|E|$ is expressed as N^2. As mentioned previously, the high complexity of scheduling algorithms is problematic in cloud environments that have a large number of services to be performed.



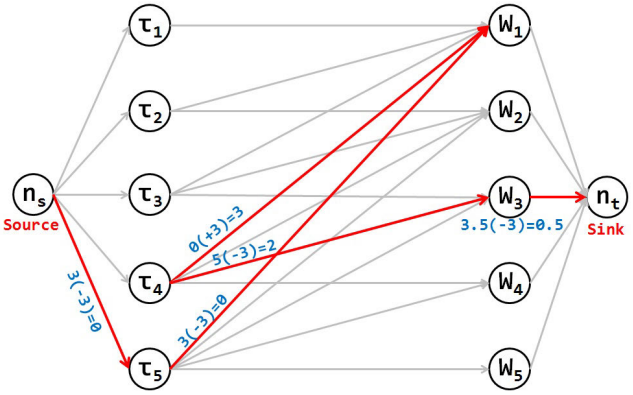
(a) An initial flow network



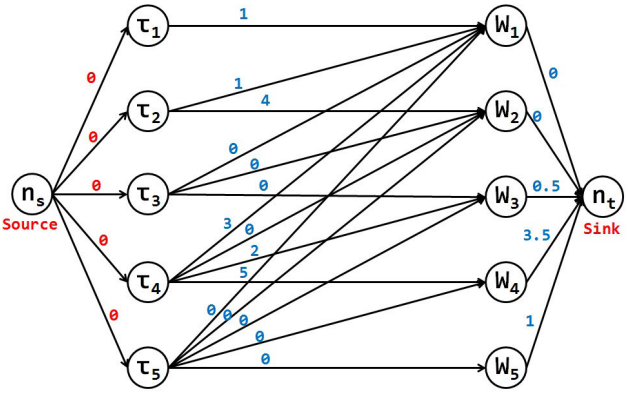
(b) A flow network after finding augmenting paths at the first window



(c) A first augmenting path



(d) A second augmenting path



(e) A final flow network found a feasible solution

FIGURE 7. The examples for showing the procedure to find an augmenting path in the flow network.

APPENDIX B
PROCESS FOR FINDING THE ADDITIONAL AUGMENTING PATHS

With the flow network constructed from the above scheduling problem, the procedures for finding the additional augmenting paths are described as follows. To demonstrate the procedures, we used the following example with its flow network.

(EXAMPLE)

Figure 7 shows the flow network constructed from a given set of task set in Table 2. Figure 7 (a) shows the initial flow network which is constructed based on the scheduling problem formulated by equations (16)-(22). In this figure, each number represents the residual capacity at the corresponding edges. Corresponding to Figure 7, Figure 8 shows the *active job area*, defined as a collection of the maximum processing

	W_1	W_2	W_3	W_4	W_5	Residual C_i
τ_1	$0 \le f \le 5$					**4**
τ_2	$0 \le f \le 5$	$0 \le f \le 5$				**5**
τ_3	$0 \le f \le 5$	$0 \le f \le 5$	$0 \le f \le 5$			**15**
τ_4	$0 \le f \le 5$	$0 \le f \le 5$	$0 \le f \le 5$	$0 \le f \le 5$		**10**
τ_5	$0 \le f \le 5$	$0 \le f \le 5$	$0 \le f \le 5$	$0 \le f \le 5$	$0 \le f \le 5$	**25**
Residual Cap(W_k)	**20**	**16**	**13.5**	**8.5**	**6**	Max. flow = 0

(a) An initial active job area

	W_1	W_2	W_3	W_4	W_5	Residual C_i
τ_1	**4**					**0**
τ_2	**5**	$0 \le f \le 5$				**0**
τ_3	**5**	$0 \le f \le 5$	$0 \le f \le 5$			**10**
τ_4	**5**	$0 \le f \le 5$	$0 \le f \le 5$	$0 \le f \le 5$		**5**
τ_5	$0 \le f \le 5$	$0 \le f \le 5$	$0 \le f \le 5$	$0 \le f \le 5$	$0 \le f \le 5$	**25**
Residual Cap(W_k)	**1**	**16**	**13.5**	**8.5**	**6**	Max. flow = 19

(b) An active job area after finding augmenting paths at the first window

	W_1	W_2	W_3	W_4	W_5	Residual C_i
τ_1	4					**0**
τ_2	**5(-1)=4**	**0(+1)=1**				**0**
τ_3	5	5	5			**0**
τ_4	5	5	0	0		**0**
τ_5	**1(+1)=2**	5	5	5	5	**4(-1)=3**
Residual Cap(W_k)	**0**	**1(-1)=0**	**3.5**	**3.5**	**1**	Max. flow = 56

(c) An active job area at the first augmenting path

	W_1	W_2	W_3	W_4	W_5	Residual C_i
τ_1	4					**0**
τ_2	4	1				**0**
τ_3	5	5	5			**0**
τ_4	**5(-3)=2**	5	**0(+3)=3**	0		**0**
τ_5	**2(+3)=5**	5	5	5	5	**3(-3)=0**
Residual Cap(W_k)	**0**	**0**	**3.5(-3)=0.5**	**3.5**	**1**	Max. flow = 59

(d) An active job area at the second augmenting path

	W_1	W_2	W_3	W_4	W_5	Residual C_i
τ_1	4					**0**
τ_2	4	1				**0**
τ_3	5	5	5			**0**
τ_4	2	5	3	0		**0**
τ_5	5	5	5	5	5	**0**
Residual Cap(W_k)	**0**	**0**	**0.5**	**3.5**	**1**	Max. flow = 59

(e) A final active job area found a feasible solution

FIGURE 8. **The examples for the active job area associated with Figure 7.**

capacity per window that can be utilized to execute the active jobs. In the flow network, the capacity at the edges between the task and window nodes is determined by the length of the window, which includes the amount of allowed flow

calculated using the real-time task and the idle time within the window. However, real-time tasks often have more flows than the flow allowed within the window because their upper limit is the length of the window by equation (19). Thus, τ_1, τ_2, τ_3, and τ_4 can be executed over the entire length of W_1, as shown in Figure 7 (b) with Figure 8 (b), that is, $f(e(\tau_1, W_1)) = 5, f(e(\tau_2, W_1)) = 5, f(e(\tau_3, W_1)) = 5$, and $f(e(\tau_4, W_1)) = 5$. For ease of understanding, the numbers 1 and 5 in Figure 7 (b), which is expressed as $1/5$, represents the residual capacity after finding augmenting paths in the first window and the initial residual capacity defined in Figure 7 (a), respectively. On the one hand, τ_5 has to send the flows of 25, that is, the required worst-case execution time, over the flow network to make the maximum flow. However, because the residual capacity at the edge $e(W_1, n_t)$ has only 1, τ_5 can have an amount of flow of at most 1. In addition, there is no additional residual capacity in the flow network to send the flows of 25 even if it spends all of the flow at the edges $\{e(\tau_5, W_2), e(\tau_5, W_3), e(\tau_5, W_4), e(\tau_5, W_5)\} = \{5, 5, 5, 5\}$. To send the residual amount flows of 4 for τ_5, the solver has to find the augmenting path again and re-determine the amount of flow at the other edges. This observation of finding an additional augmenting path can occur at other nodes to make the maximum flow.

In this example, Figure 7 (c) with Figure 8 (c) shows the augmenting path in the order of $\{n_s, \tau_5, W_1, \tau_2, W_2, n_t\}$. The flow of 1 from $e(\tau_2, W_1)$ can be sent to $e(\tau_2, W_2)$. As a result, this procedure increases the capacity at $e(W_1, n_t)$ to 1, and then it is used for τ_5 to send an additional flow of 1 through $e(\tau_5, W_1)$. Figure 7 (d) and Figure 8 (d) show the next augmenting path in the order of $\{n_s, \tau_5, W_1, \tau_4, W_3, n_t\}$. The flow of 3 from $e(\tau_4, W_1)$ can be sent to $e(\tau_4, W_3)$. As a result, this procedure also increases the capacity at $e(W_1, n_t)$ to 3, and then it is used for τ_5 again to send an additional flow of 3 through $e(\tau_5, W_1)$. Finally, the feasible solution for the *max-flow* problem in this example is shown in Figure 7 (e) and Figure 8 (e), which has a maximum flow of 59. On the one hand, this maximum flow is the same as the summation of the worst-case execution times for active tasks in Table 2. Although the maximum amount of flow is already known as 59, it is necessary to find additional augmenting paths again because of the idle time included in $sum_{\forall k}(W_k, n_t)$ as 64. In this example, the procedure for finding augmenting paths was performed twice. However, when the number of real-time tasks and windows is large, it may be necessary to find an increasing number of augmenting paths. Thus, the following sub-section introduces how to control the use of idle time included in the summation of windows' capacity to limit the procedure of finding additional augmenting paths.

REFERENCES

[1] A. Fox, R. Griffith, A. D. Joseph, R. H. Katz, A. Konwinski, G. Lee, D. A. Patterson, A. Rabkin, I. Stoica, and M. Zaharia, "Above the clouds: A Berkeley view of cloud computing," Dept. Elect. Eng. Comput. Sci., Univ. California, Berkeley, CA, USA, Tech. Rep. UCB/EECS 28.13, 2009.

[2] M. Abdallah, C. Griwodz, K. T. Chen, G. Simon, P. C. Wang, and C. H. Hsu, "Delay-sensitive video computing in the cloud: A survey," *ACM Trans. Multimedia Comput., Commun., Appl.* vol. 14, no. 3, pp. 1–29, 2018.

[3] L. Columbus. (2017). *Roundup of Cloud Computing Forecasts, 2017*. Forbes & Company Ltd. Accessed: Apr. 26, 2018. [Online]. Available: https://www.forbes.com/sites/louiscolumbus/2017/04/29/roundup-of-cloud-computing-forecasts-2017/

[4] P. Delforge, *America's Data Centers Consuming and Wasting Growing Amounts of Energy*. New York, NY, USA: Natural Resource Defence Councle, 2014.

[5] G. J. Koomey, "Estimating total power consumption by servers in the U.S. and the world," Lawrence Berkeley Nat. Lab., Berkeley, CA, USA, Final Rep., 2007.

[6] M. Claypool and K. Claypool, "Latency and player actions in online games," *Commun. ACM*, vol. 49, no. 11, pp. 40–45, 2006.

[7] A. Tasiopoulos, "On the deployment of low latency network applications over third-party in-network computing resources," Ph.D. dissertation, Dept. Electron. Elect. Eng., Univ. College London, London, U.K., 2018.

[8] D. P. Olshefski, J. Nieh, and D. Agrawal, "Inferring client response time at the web server," *ACM SIGMETRICS Perform. Eval. Rev.*, vol. 30, no. 1, pp. 160–171, Jun. 2002.

[9] K. H. Kim, A. Beloglazov, and R. Buyya, "Power-aware provisioning of cloud resources for real-time services," in *Proc. 7th Int. Workshop Middleware Grids, Clouds e-Sci. (MGC)*, 2009, pp. 29–34.

[10] X. Feng and A. K. Mok, "A model of hierarchical real-time virtual resources," in *Proc. 23rd IEEE Real-Time Syst. Symp. (RTSS)*, Dec. 2002, pp. 26–35.

[11] I. Shin and I. Lee, "Compositional real-time scheduling framework with periodic model," *ACM Trans. Embedded Comput. Syst.*, vol. 7, no. 3, pp. 1–39, Apr. 2008.

[12] J. Sun, H. Cho, A. Easwaran, J.-D. Park, and B.-C. Choi, "Flow network-based real-time scheduling for reducing static energy consumption on multiprocessors," *IEEE Access*, vol. 7, pp. 1330–1344, 2019.

[13] H. Cho and A. Easwaran, "Flow network models for online scheduling real-time tasks on multiprocessors," *IEEE Access*, vol. 8, pp. 172136–172151, 2020.

[14] S. G. Soriga and M. Barbulescu, "A comparison of the performance and scalability of xen and KVM hypervisors," in *Proc. RoEduNet Int. Conf. 12th Ed., Netw. Educ. Res.*, Sep. 2013, pp. 1–6.

[15] T. Deshane, Z. Shepherd, J. Matthews, M. Ben-Yehuda, A. Shah, and B. Rao, *Quantitative Comparison of Xen and KVM*. Boston, MA, USA: Xen Summit, 2008, pp. 1–2.

[16] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, "Xen and the art of virtualization," in *Proc. ACM Symp. Operating Syst. Princ.*, 2003, pp. 113–1118.

[17] Xen Project. *Currently Available Schedulers in Xen Project Scheduler*. Accessed: Apr. 2021. [Online]. Available: http://www.nxp.com/docs/en/data-sheet/LPC1850_30_20_10.pdf

[18] S. Xi, J. Wilson, C. Lu, and C. Gill, "RT-Xen: Towards real-time hypervisor scheduling in Xen," in *Proc. 9th ACM Int. Conf. Embedded Softw. (EMSOFT)*, Oct. 2011, pp. 39–48.

[19] J. Lee, S. Xi, S. Chen, L. T. X. Phan, C. Gill, I. Lee, C. Lu, and O. Sokolsky, "Realizing compositional scheduling through virtualization," in *Proc. IEEE 18th Real Time Embedded Technol. Appl. Symp.*, Apr. 2012, pp. 13–22.

[20] M. Ala'Anzy and M. Othman, "Load balancing and server consolidation in cloud computing environments: A meta-study," *IEEE Access*, vol. 7, pp. 141868–141887, 2019.

[21] H. A. Kurdi, S. M. Alismail, and M. M. Hassan, "LACE: A locust-inspired scheduling algorithm to reduce energy consumption in cloud datacenters," *IEEE Access*, vol. 6, pp. 35435–35448, 2018.

[22] J. Dong, X. Jin, H. Wang, Y. Li, P. Zhang, and S. Cheng, "Energy-saving virtual machine placement in cloud data centers," in *Proc. 13th IEEE/ACM Int. Symp. Cluster, Cloud, Grid Comput.*, May 2013, pp. 618–624.

[23] F. Cao and M. M. Zhu, "Energy efficient workflow job scheduling for green cloud," in *Proc. IEEE Int. Symp. Parallel Distrib. Process., Workshops Phd Forum*, May 2013, pp. 2218–2221.

[24] C. Da-Ren, C. Young-Long, and C. You-Shyang, "Time and energy efficient DVS scheduling for real-time pinwheel tasks," *J. Appl. Res. Technol.*, vol. 12, no. 6, pp. 1025–1039, 2014.

[25] M. Hadji and D. Zeghlache, "Minimum cost maximum flow algorithm for dynamic resource allocation in clouds," in *Proc. IEEE 5th Int. Conf. Cloud Comput.*, Jun. 2012, pp. 876–882.

[26] Y. Ding, X. Qin, L. Liu, and T. Wang, "Energy efficient scheduling of virtual machines in cloud with deadline constraint," *Future Gener. Comput. Syst.*, vol. 50, pp. 62–74, Sep. 2015.
[27] L. Guo, G. Hu, Y. Dong, Y. Luo, and Y. Zhu, "A game based consolidation method of virtual machines in cloud data centers with energy and load constraints," *IEEE Access*, vol. 6, pp. 4664–4676, 2017.
[28] P. G. Jeba Leelipushpam and J. Sharmila, "Live VM migration techniques in cloud environment—A survey," in *Proc. IEEE Conf. Inf. Commun. Technol.*, Apr. 2013, pp. 408–413.
[29] L. R. Ford and D. R. Fulkerson, "Maximal flow through a network," in *Classic Papers Combinatorics*. Boston, MA, USA: Birkhäuser, 2009, pp. 243–248.
[30] J. Edmonds and R. M. Karp, "Theoretical improvements in algorithmic efficiency for network flow problems," *J. ACM*, vol. 19, no. 2, pp. 248–264, 1972.
[31] N. Yadav. *Python Program for Implementation of Ford Fulkerson Algorithm*. Accessed: Jan. 2021. [Online]. Available: https://github.com/lrlucena/Algorithms-and-Data-Structures/blob/master/books/Algorithms%2BDataStructures/Algorithms/Graphs/ford1242fulkerson/python/max_flow.py
[32] R. Calheiros, R. Ranjan, A. Beloglazov, C. A. F. De Rose, and R. Buyya, "CloudSim: A toolkit for modeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms," *Softw., Pract. Exper.*, vol. 41, no. 1, pp. 23–50, 2011.
[33] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield, "Live migration of virtual machines," in *Proc. 2nd Conf. Symp. Netw. Syst. Design Implement.*, vol. 2, 2005, pp. 101–110.
[34] M. R. Hines and K. Gopalan, "Post-copy based live virtual machine migration using adaptive pre-paging and dynamic self-ballooning," in *Proc. ACM SIGPLAN/SIGOPS Int. Conf. Virtual Execution Environ. (VEE)*, 2009, pp. 51–60.
[35] P. R. Panda, A. Shrivastava, B. V. N. Silpa, and K. Gummidipudi, *Power-Efficient System Design*. Boston, MA, USA: Springer, 2010, ch. 2, pp. 11–39, doi: 10.1007/978-1-4419-6388-8.
[36] R. McNaughton, "Scheduling with deadlines and loss functions," *Manage. Sci.*, vol. 6, no. 1, pp. 1–12, 1959.
[37] J. B. Orlin, "A faster strongly polynomial minimum cost flow algorithm," *Oper. Res.*, vol. 41, no. 2, pp. 338–350, 1993.
[38] M. L. Fredman and R. E. Tarjan, "Fibonacci heaps and their uses in improved network optimization algorithms," *J. ACM (JACM)*, vol. 34, no. 3, pp. 596–615, 1987.
[39] J. B. Orlin, "Max flows in O(nm) time, or better," in *Proc. 45th Annu. ACM Symp. Symp. Theory Comput. (STOC)*, 2013, pp. 765–774.
[40] D. Zhu, X. Qi, D. Mossé, and R. Melhem, "An optimal boundary fair scheduling algorithm for multiprocessor real-time systems," *J. Parallel Distrib. Comput.*, vol. 71, no. 10, pp. 1411–1425, Oct. 2011.

JOOHYUNG SUN received the B.S., M.S., and Ph.D. degrees from the Department of Computer and Information Science, Korea University. He is currently a Postdoctoral Researcher at the Electronics and Telecommunications Research Institute, South Korea. His research interests include real-time computing and energy-aware scheduling on various types of processing elements (single/multiprocessors and cloud computing environments).

HYEONJOONG CHO received the B.S. degree in electronic engineering from Kyungpook National University, South Korea, in 1996, the M.S. degree in electronic and electrical engineering from the Pohang University of Science and Technology, in 1998, and the Ph.D. degree in computer engineering from the Virginia Polytechnic Institute and State University (Virginia Tech), in 2006. He was a Senior Software Engineer at Samsung Electronics, South Korea. Before he joined Korea University, in 2009, he was a Senior Researcher at the Electronics and Telecommunications Research Institute, South Korea. He is currently a Professor with the Department of Computer and Information Science, Korea University. His research focuses on machine learning, optimization techniques, and cyber-physical systems.