Resource and Instance Hour Minimization for Deadline Constrained DAG Applications Using Computer Clouds

Hao Wu, *Student Member, IEEE*, Xiayu Hua, *Student Member, IEEE*, Zheng Li, *Student Member, IEEE* and Shangping Ren, *Senior Member, IEEE*,

Abstract—In this paper, we address the resource and virtual machine instance hour minimization problem for directed-acyclic-graph-based deadline constrained applications deployed on computer clouds. The allocated resources and instance hours on computer clouds must: (1) guarantee the satisfaction of a deadline constrained application's end-to-end deadline; (2) ensure that the number of virtual machine (VM) instances allocated to the application is minimized; (3) under the allocated number of VM instances, determine application execution schedule that minimizes the application's makespan; and (4) under the decided application execution schedule, determine a VM operation schedule, i.e., when a VM should be turned on or off, that minimizes total VM instance hours needed to execute the application. We first give lower and upper bounds for the number of VM instances needed to guarantee the satisfaction of a deadline constrained application's end-to-end deadline. Based on the bounds, we develop a heuristic algorithm called minimal slack time and minimal distance (MSMD) algorithm that finds the minimum number of VM instances needed to guarantee the application's deadline and schedules tasks on the allocated VM instances so that the application's makespan is minimized. Once the application execution schedule and the number of VM instances needed are determined, the proposed VM instance hour minimization (IHM) algorithm is applied to further reduce the instance hours needed by VMs to complete the application's execution. Our experimental results show that the MSMD algorithm can guarantee applications' end-to-end deadlines with less resources than the HEFT [32], MOHEFT [16], DBUS [9], QoS-base [40] and Auto-Scaling [25] heuristic scheduling algorithms in the literature. Furthermore, under allocated resources, the MSMD algorithm can, on average, reduce an application's makespan by 3.4% of its deadline. In addition, with the IHM algorithm we can effectively reduce the application's execution instance hours compared with when IHM is not applied.

Index Terms—Cloud, Scheduling, Cost minimization, Makespan minimization, Resource Minimization, Real-time,MSMD, Instance Hour Minimization

1 Introduction

The advancement of computer and network technology has brought the world into a new computer cloud era. The "pay-as-you-go" business model and the service oriented models allow users to have "unlimited" resources if needed and free from infrastructure maintenance and software upgrades. Cloud services are currently among the top-ranked high growth areas in computing and are seeing an acceleration in enterprise adoption with the worldwide market predicted to reach more than $270b in 2020 [3]. Many different types of applications are deployed on computer clouds. For instance, both Argonne National Laboratory and Fermi National Accelerator Laboratory provide their cloud platforms for scientific applications [22], [12]. Deadline constrained applications such as online media streaming applications [28], interactive deadline constrained e-learning [4], and online banking systems [31] are also seeking opportunities to utilize computer clouds.

The authors are with Illinois Institute of Technology, 10 W 31st street, 013, Chicago, IL, USA
{hwu28, xhua, zli80, ren}@iit.edu

The research is supported by in part by NSF under grant number CAREER 0746643, CNS 1018731 and CNS 1035894. We would like to thank Daniela Martinez for proof reading the paper and reviewers for their constructive and valuable comments that have improved the paper.

In our joint project with the Chicago Waste Water Treatment Plant on its air blower control system, we have designed a real-time control system that is used to dynamically control the air blower speed based on deadline constrained data and prediction models (including weather prediction, chemical process predictions, etc.) to reduce the electricity cost [1]. The application contains a set of dependent tasks, i.e., data sensing, data processing, data storage, monitoring, prediction, decision making, and actions to turn up or down the speed of the air blower. If the application miss its deadline, i.e., the blower's speed is not turned up by certain time, it can cause catastrophic consequences, such as causing not properly treated waste water disposed to the waterways.

With the support of cloud and cloud bursting technologies [36], companies do not need to provision their resources for the worst case scenarios anymore. With a private cloud, local resources need not be dedicated to the application at all time as the application's resource need can be low when the weather conditions are steady. When applications do need more computation resources, such as when weather changes more frequently and abruptly and more computation power is needed for modeling and prediction calculations, public cloud resources can be obtained. However, one of the major technical issues we are facing when deploying such deadline constrained applications on the cloud is how to minimize cost for deploying applications on cloud while guaranteeing the deadline requirement requested by the applications.

In the context of cloud computing, an application's execution cost refers to the monetary cost of renting resources on public cloud. In the literature, good amount of effort has been made in addressing the issue of how to minimize the cost for applications using cloud platforms. For instance, some researchers have focused on minimizing the cost by reducing the applications' makespan [32], [9], [27], some considered cost as one of the QoS requirements and thus have transformed the problem into multi-objective optimization problems that balance the trade offs between the cost and other QoS requirements [29], [20], [6], [15], [11], and others have studied cost minimization problem on hybrid clouds [35], [34], [33], [8], [10], [13].

The operational cost on private cloud is usually negligible compared to the cost on public cloud. Therefore, for hybrid cloud, minimizing an application's makespan does not necessarily minimize the cost. For instance, if we only utilize resources in a private cloud, an application's makespan is ten hours. If we use two additional VMs on a public cloud for the same application, we can reduce the application's makespan to five hours. Clearly, by utilizing public cloud, we have reduced application's makespan, but also increased the application's execution cost.

For a deadline constrained application, meeting the application's deadline requirement is critical, but there is no incentive to finish the application earlier. On the other hand, in private cloud, reducing applications' makespan can increase system's throughput. Hence, if we can guarantee an application's deadline requirement with the least number of resources and then further minimize application's makespan under the given least number of resources, both clients and service providers benefit the most. Furthermore, as in a cloud environment, virtual machine instances are charged only when they are running. Such feature enables users to further reduce cost by running virtual machines intelligently.

The work presented in this paper addresses virtual machine instances and VM instance hour minimization issue for deadline constrained applications deployed on computer clouds. We take two steps to target the problem. First, we reduce the number of VM instances, called horizontal reduction, and then we reduce the instance hours consumed by the virtual machines, called vertical reduction. In particular, for a given distributed deadline constrained application with an end-to-end deadline constraint and a computer cloud with an *unspecified* number of VM instances, we first decide the number of VM instances needed and then create a schedule on each VM instance to guarantee: 1) the application's end-to-end deadline is satisfied, 2) the number of VM instances needed for executing the application tasks is minimized, 3) under the minimized number of VM instances, the application's makespan is minimized; and (4) decide a running strategy for each VM instance allocated for the application such that the total cost (i.e., the total charge for the VM instance hours) is minimized.

The rest of the paper is organized as follows: Section 2 discusses related work. In Section 3, we first introduce the models and terms, then formally define the cost minimization problem the paper is to address. Section 4 presents an analysis to quickly calculate the resource bounds needed to guarantee a DAG-based deadline constrained application's end-to-end deadline. Section 5 gives a heuristic algorithm to decide the number of VM instances needed for a given deadline constrained application and the task schedule on each VM instance. Section 6 gives the algorithm to minimize the instance hours. Experimental evaluations are presented in Section 7. We conclude and point out future work in Section 8.

2 Related Work

The essence behind resource minimization and application makespan minimization problems can be drilled down to a task scheduling problem which is proven to be NP-complete when there are more than two computers [17]. Thus, many heuristic approaches have been proposed. List scheduling is one of the basic approaches used for makespan minimization and it has a $(2 - \frac{1}{m})$ approximation to the optimal makespan [18], where m is the number of processors. The idea of list scheduling is to list tasks in an order and then schedule tasks based the ordered list. Hence, ordering the task list becomes critical when designing list based algorithms.

Researchers have made significant efforts on ordering tasks and have developed many list scheduling based heuristic algorithms to solve application makespan minimization problems [14], [32], [9]. A well-known list scheduling based algorithm is the Coffman-Graham (CG) algorithm [14]. The CG algorithm takes a set of partially ordered tasks and assigns task priorities based on their order. The CG schedules the task with the highest priority in the list to the computer that has the earliest available time at the time of scheduling. When there are only two homogeneous computers, the CG scheduling algorithm is proven to be the optimal [14]. In order to schedule independent tasks on heterogeneous computers, min-min algorithm was proposed [24].

However, neither the CG algorithm nor min-min algorithm can be directly applied to DAG-based applications unless the dependencies among tasks in a DAG-based application are resolved. One commonly used approach to decoupling task dependencies is to list the DAG-based application in a topological order. Another commonly used approach is to assign and list tasks by their priorities. The prioritization scheme is based on when each task finishes, i.e. counting from the bottom of the DAG-based application task graph (b_{level}), or when each task starts, i.e. counting from the top (t_{level}). Many existing makespan minimization algorithms adapt this approach as their prioritization basis. The heterogeneous-earliest-finish-time (HEFT) algorithm [32] is one of them and it uses the summation of a task's b_{level} and t_{level} values as its priority and hence provides an $O(|V|^2m)$ list-based heuristic algorithm for minimizing the makespan.

A duplication based bottom up scheduling (DBUS) algorithm [9] is another heuristic algorithm that takes the b_{level} and t_{level} approach as the basis of its prioritization method. Unlike the HEFT algorithm, the DBUS approach takes the t_{level} and an additional static top level st_{level} as the tasks' priorities. The DBUS also duplicates tasks on each machine at the scheduling phase and thus is a $O(|V|^2m^2)$ heuristic.

Under cloud computing environment, in addition to minimize application's makespan, reducing application's execution cost on computer clouds becomes another important resource management objective. There are two major research efforts on application execution cost on computer clouds, one is how to schedule applications on a computer cloud to meet their deadlines under a given budget constraint [37], [26], [38]. The other is how to schedule applications on a computer cloud such that the applications meet their deadlines and the cost is minimized [39], [25], [23].

For applications deployed on public computer clouds, Durillo *et al.* extended the HEFT algorithm [32] and developed a parto-based list scheduling heuristic called MOHEFT. The objectives of the MOHEFT algorithm are to optimize application's makespan and its execution cost a public computer cloud [16]. Yu *et al.* proposed a QoS-based workflow scheduling algorithm to minimize the workflow's execution cost on a public cloud while meeting the workflow's deadline, and Khanli [23] proposed a Markov Decision Process based approach to minimizing the execution cost while meeting tasks' deadline.

There are also significant amount of work done in the area of minimizing application's execution cost deployed a hybrid computer cloud [35], [34], [33], [8], [10], [13]. For instance, Ruben *et al.* proposed an online cost-efficient scheduling algorithm to schedule deadline constrained applications with parallel tasks on hybrid cloud so that the application's deadline is met and financial cost on public cloud is minimized [35]. Luiz *et al.* proposed a hybrid cloud optimized cost scheduling algorithm to minimize the cost of DAG-based application on hybrid cloud [8].

However, many existing cost minimization approaches do not consider that cloud service charges are based on instance hours or minutes. As pointed out in [25], the integral instance hour increases the difficulty for solving the cost minimization problem. The auto scaling scheduling algorithm [25] is one of the algorithms that aims to minimize the cost by considering integral instance hours. In their algorithm, they assign tasks' local deadlines using the same technique as developed in [40]. After assigning local deadlines, they decide the number and the types of virtual machines needed to execute the application. Finally, they schedule the tasks using the global EDF algorithm.

The research briefly summarized above has been mainly focused on how to schedule tasks *under fixed amount of resources* to maximize the system's throughput and minimize an application's makespan, rather than to minimize the number of computers needed to guarantee a deadline constrained application's deadline. In fact, it is possible that an application can be scheduled on n computers with the same makespan as on m $(m > n)$ computers by different heuristic algorithms.

Research on resource bound problem for DAG-based deadline constrained applications can be traced in the early 1960's [19] and good amount of research results are obtained since then [30], [5]. However, neither T.C. Hu's original lower bound [19] nor Ramamoorthy's improvement [30] can be directly applied to the DAG-based application with different task execution times if tasks are not allowed to migrate among computers. Al-Mouhamed further extend and improve the resource lower bound with the consideration of heterogeneous task execution time and communication cost [5]. However, Al-Mouhamed's method to calculate the lower bound is too expensive to be applicable in practice for large scale applications and for on-line cloud applications.

In this paper, we use similar steps as the auto-scaling algorithm in minimizing the cost of instance hour: we first determine the minimum number of virtual machines needed for a deadline constrained DAG-based application, then reduce the instance hours on each virtual machine instance by deciding when to turn on and off VM instances. We will use the HEFT [32], MOHEFT [16], DBUS [9], QoS [40] and auto-scaling [25] algorithms from the literature as base lines to evaluate our proposed approach.

3 Problem Formalization

In this section, we first introduce the models and terms our work is based upon and then formulate the cost minimization problem the paper is to address.

3.1 Application Model

A deadline constrained application A is modeled as a weighted directed-acyclic-graph (DAG) $G(V, E)$, where each task $\tau_i \in A$ is represented by a node $v_i \in V$, the weight $w(\tau_i)$ on the node v_i represents task τ_i's worst case execution time (WCET) on a unit speed VM instance, and an edge $(v_i, v_j) \in E$ represents dependency between τ_i to τ_j. A task can only start after all its predecessors complete.

It is worth highlighting that the edge does represent data transfers between tasks. However, in our system, the network speed is much faster than disk IO speed. Hence, the communication time between adjacent nodes is assumed to be negligible. Each application is given a release time T_R and a relative end-to-end deadline T_D. The relative end-to-end deadline is the time interval from when an application is released to when the application must be finished. Hence, the application's absolute end-to-end deadline is $T_R + T_D$. When a deadline constrained application arrives, application's precise information are known to the system, i.e., the application's task graph, each task's WCET and application's end-to-end deadline are all given.

Tasks without any predecessors or any successors are defined as *entry* tasks and *exit* tasks, respectively. Without loss of generality, we assume each application has one entry task denoted as τ_{entry} and one exit task denoted as τ_{exit} [1]. Fig. 1 gives an example of a DAG-based application task graph, where $\tau_{entry} = \tau_0$ and $\tau_{exit} = \tau_{10}$.

3.2 System Model

A computer cloud in this paper is modeled as a set of virtual machine instances, i.e. $C = \{c_1, \ldots, c_M\}$. We assume that there is only one virtual machine type, i.e. all virtual machine instances are homogeneous with unit speed. In addition, as many cloud infrastructures are built on a shared file system such as SAN-based storage systems [36] and

1. If an application has multiple entry tasks or multiple exit tasks, we add a *virtual entry* task and a *virtual exit* task with zero execution time and connect from the virtual entry task to all actual entry tasks and from all actual exit tasks to the virtual exit task, respectively.

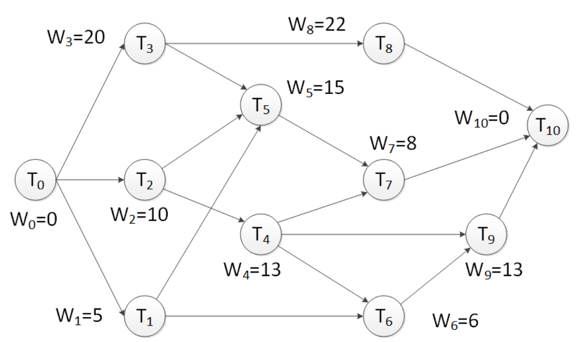


Fig. 1: An Example of DAG-Based Application Task Graph with End-to-End Deadline $T_D = 60$

VMs for the same application can use the same shared data files, hence data communication and movement cost become negligible. Therefore, under this computer cloud model, task execution time does not change when it is deployed to different virtual machine instances in the cloud.

Task executions are non-preemptive and each virtual machine instance can only execute one task at any given time. Unprocessed tasks are buffered in a task queue by the virtual machine instance the tasks are assigned to. Virtual machine instances are charged by unit time such as by hours. If an VM instance is running less than a unit time, such as less than an hour, it is charged as one hour. We assume that there is no overhead for powering on or off a virtual machine.

Let $\mathcal{P}_i = \{(on_i^1, off_i^1), \ldots, (on_i^n, off_i^n)\}$ be the set of time intervals of a virtual machine instance c_i in running state, where on_i^j and off_i^j represent the j^{th} power on and off time points, respectively, and $\forall j, k$ if $j < k$, then $on_i^j < off_i^j < on_i^k < off_i^k$. We call $\mathcal{P}_i$ the virtual machine instance's operation pattern.

Given a virtual machine instance's operation pattern $\mathcal{P}_i$, the instance hours needed is

$$H(\mathcal{P}_i) = \sum_{j=1}^{|\mathcal{P}_i|} \left\lceil \frac{off_i^j - on_i^j}{U} \right\rceil \quad (1)$$

where U is a constant representing cloud service pricing time unit. For instance, if VM instance is charged by hour, while $off_i^j - on_i^j$ is in minutes, then either price unit is converted to minutes, or the VM running time is converted to hours. As we only consider one type of virtual machine instance, all virtual machine instances have the same hourly price. Hence, if we can minimize the instance hours for each instance, the total cost is minimized.

For a given application $A = \{\tau_1, \ldots, \tau_n\}$ and a set of VMs where application tasks are deployed on, a boolean function $S(\tau_i, c_j)$ defines whether a task $\tau_i \in A$ is deployed on VM instance c_j. In other words, if a VM instance c_j is used by application A, then $\exists \tau_i \in A$ such that $S(\tau_i, c_j) = 1$. We use $R(A)$ to denote the resource set utilized by application A:

$$R(A) = \{c_j | \exists \tau_i \in A, c_j \in C \wedge S(\tau_i, c_j) = 1\} \quad (2)$$

3.3 Definitions

Given a DAG-based application $A = \{\tau_1, \ldots, \tau_n\}$ with release time T_R and relative deadline T_D, and its corresponding task graph $G(V, E)$, we define the following terms.

Predecessors ($pred(\tau_i)$) and Successors ($succ(\tau_i)$)

For task $\tau_i \in V$, its predecessor and successor task sets are defined below:

$$pred(\tau_i) = \{\tau_j | \tau_j \in A \wedge (\tau_j, \tau_i) \in E\} \quad (3)$$
$$succ(\tau_i) = \{\tau_j | \tau_j \in A \wedge (\tau_i, \tau_j) \in E\} \quad (4)$$

Application Sequential Execution Time (T_{seq})

The sequential execution time of application A is defined as the summation of all composing tasks' execution time.

Task Earliest Start Time ($EST(\tau_i)$) and Latest Finish Time ($LFT(\tau_i)$)

For a given task $\tau_i \in V$, its earliest start time and latest finish time are recursively defined as follows:

$$EST(\tau_i) = \begin{cases} T_R & \textit{if } \tau_i = \tau_{entry} \\ \max\limits_{\tau_k \in pred(\tau_i)} \{EST(\tau_k) + w_k\} & \textit{otherwise} \end{cases} \quad (5)$$

$$LFT(\tau_i) = \begin{cases} T_R + T_D & \textit{if } \tau_i = \tau_{exit} \\ \min\limits_{\tau_k \in succ(\tau_i)} \{LFT(\tau_k) - w_k\} & \textit{otherwise} \end{cases} \quad (6)$$

A task's earliest start time and latest finish time are determined solely based on the application's task graph, its release time and relative deadline. They are independent of how tasks are assigned to VM instances.

Task Scheduled Start Time ($SST(\tau_i)$) and Schedule Finish Time ($SFT(\tau_i)$)

A task's scheduled start time ($SST(\tau_i)$) and scheduled finish time ($SFT(\tau_i)$) are defined as when the task is scheduled for execution and completed its execution on a VM instance, respectively. They can be different from its earliest start time and latest finish time since the slack time between tasks may vary under different schedules. A task is not necessarily started at its earliest start time, neither is finished at its latest finish time. In fact, we have $SST(\tau_i) \geq EST(\tau_i)$ and $SFT(\tau_i) = SST(\tau_i) + w_i$.

Task Ready Time ($ready(\tau_i)$)

A task's ready time $ready(\tau_i)$ is the latest scheduled finish time of all its predecessors, i.e.

$$ready(\tau_i) = \max_{\tau_k \in pred(\tau_i)} \{SFT(\tau_k)\} \quad (7)$$

Task Maximal Slack Time ($mslack(\tau_i)$)

For a given task $\tau_i \in V$, its maximal slack time is defined as

$$mslack(\tau_i) = LFT(\tau_i) - (EST(\tau_i) + w(\tau_i)) \quad (8)$$

Intuitively, the maximal slack time indicates how long a task can afford to wait before causing a deadline violation. For task τ_i, $mslack(\tau_i) = 0$ means τ_i must start at its earliest start time or it will cause the application to miss its end-to-end deadline.

Task Topological Level **(**$Lev(\tau_i)$**)**

Given a DAG-based application A, its task τ_i's topological level $Lev(\tau_i)$ is defined as:

$$Lev(\tau_i) = \begin{cases} 0 & \text{if } \tau_i = \tau_{entry} \\ \max\limits_{\tau_k \in pred(\tau_i)} \{Lev(\tau_k)\} + 1 & \text{otherwise} \end{cases} \quad (9)$$

Critical Path P_c *and* ***Critical Path Execution Time*** **(**T_C**)**

For a given application task graph, a path execution time is defined as the summation of its composing tasks' execution times along the path. A critical path, denoted as P_c, is a path that starts at the entry task τ_{entry}, ends at the exit task τ_{exit}, and has the longest path execution time. There may exist more than one critical paths in an application's task graph. However, by definition, every critical path has the same path execution time. We denote the critical path execution time as T_C. Since different schedules can result different critical path execution times for the same task graph. Hence, in this paper, the T_C refers to an application's critical path execution time determined by its task graph.

Schedulable Application

For a given application A with relative deadline T_D, the application is *schedulable* if and only if its critical path execution time satisfies $T_C \leq T_D$.

VM Instance Earliest Available Time **(**$av(c)$**)**

For a given VM instance c, if its totally ordered task queue is $Q_c = \{\tau_1^c, \ldots, \tau_h^c\}$, then VM instance c's earliest available time for a new task τ_i ($\tau_i \notin Q_c$) is the time that the last task in the queue finishes its execution. The last task's finish time depends on when it starts its execution. Its start time is the latest time of all its predecessors' finish time or the finish time of its previous tasks in the queue, whichever one is the latest. Hence, the earliest available time for task not in the queue can be recursively calculated as follows:

$$av(c) = \begin{cases} 0 & h = 0 \\ \max\{SFT(\tau_{h-1}^c), ready(\tau_h^c)\} & \\ \qquad + w(\tau_h^c) & h > 0 \end{cases} \quad (10)$$

The $SFT\tau_h^c$ is the scheduled finish time of the last task in the queue.

Table 1 gives the EST, LFT, mslack, Lev, and whether a task is on a critical path for the example task graph given in Fig. 1. The concept of task priority will be discussed in Section 5.

TABLE 1: Example Application's Task Property

Tasks	EST	LFT	mslack	CP	Levels	Priority
τ_0	0	17	17	✓	0	1
τ_1	0	37	32		1	4
τ_2	0	28	18		1	3
τ_3	0	37	17	✓	1	2
τ_4	10	41	18		2	6
τ_5	20	52	17	✓	2	5
τ_8	20	60	18		2	7
τ_6	23	47	18		3	9
τ_7	35	60	17	✓	3	8
τ_9	29	60	18		4	10
τ_{10}	43	60	17	✓	5	11

3.4 Problem Formulation

Based on the models and definitions presented in the earlier subsections, we formally define the problem we are to address, i.e. minimize VMs and VM instance hours for deadline constrained applications deployed on computer clouds. To achieve the goal, we take three steps. The first step is to minimize the number of VM instances needed to guarantee the satisfaction of a DAG-based deadline constrained application's end-to-end deadline. Once the minimal number of VM instances needed is decided, the second step is to minimize the application's makespan under the minimized number of VM instances decided by the first step. The third step is to schedule the power on and off time for each VM to further reduce the total VM instance hours.

Objective 1: Minimize the Number of VM Instances Needed

Given an application $A = \{\tau_{entry}, \cdots, \tau_i, \cdots, \tau_{exit}\}$ with release time T_R and relative deadline T_D, its corresponding task graph $G(V, E)$, and sufficient set of VM instances, determine a subset of VM instances $\{c_1, c_2, \ldots, c_M\}$, such that

$$\begin{aligned} &\textbf{Objective 1:} && \min M \\ &\textbf{Subject to:} && SFT(\tau_{exit}) \leq T_R + T_D && (11) \\ &\textbf{and} && \forall \tau_i \in A, \sum_{j=1}^{M} S(\tau_i, c_j) = 1 && (12) \end{aligned}$$

where $S(\tau_i, c_j) = 1$ if and only if task τ_i is assigned to VM instance c_j. The first constraint given by (11) guarantees end-to-end deadline satisfaction and the second constraint given by (12) ensures that each task can only be deployed to one VM instance.

Objective 2: Minimize Makespan under Allocated VM Instances

Once the minimum number of VM instances (M) needed to guarantee the application's end-to-end deadline is determined, our next objective is to minimize the application's makespan on the M VM instances:

$$\begin{aligned} &\textbf{Objective 2:} && \min \quad SFT(\tau_{exit}) \\ &\textbf{Subject to:} && \forall \tau_i \in A, \sum_{j=1}^{M} S(\tau_i, c_j) = 1 && (13) \end{aligned}$$

Objective 3: Minimize Total VM Instance Hours

Once the minimum number of virtual machine instances (M) and the application execution schedule to minimize the application's makespan on M virtual machine instances are determined, our final task is to minimize the total virtual machine instance hours:

$$\textbf{Objective 3:} \quad \min \sum_{i=1}^{M} H_i$$

We take three steps to achieve goals. First, we target **objective 1** by theoretically prove the lower and upper bound of the number of VMs needed to guarantee a deadline constrained application's deadline. Then we develop a heuristic scheduling algorithm to minimize application's makespan under given number of VMs. Finally, once the schedule of a deadline constrained application is determined, we develop a heuristic algorithm to further reduce VM instance hours needed to execute the application. The following three sections give the details about each of the steps.

4 Resource Bounds

In this section, we study the attributes of DAG-based deadline constrained applications and determine the bounds for the minimum number of VM instances needed to guarantee the application's end-to-end deadlines.

Lemma 1. *Given a DAG-based deadline constrained application A, let the application's release time and relative end-to-end deadline be T_R and T_D, respectively, its sequential execution time be T_{seq}, critical path execution time be T_C, and the minimal number of VM instances needed to guarantee the application's end-to-end deadline be M. If the application is schedulable, i.e., $T_C \leq T_D$, then we have:*

$$M \geq \left\lceil \frac{T_{seq}}{T_D} \right\rceil \tag{14}$$

□

Proof. We prove Lemma 1 by contradiction. If $T_{seq} \leq T_D$, we have $\lceil \frac{T_{seq}}{T_D} \rceil = 1$. As the application's sequential execution time is less than its deadline, i.e. $T_{seq} \leq T_D$, trivially, with $M = 1$ VM instance, we can guarantee the application's deadline. If $T_{seq} > T_D$, assume the minimum number of VM instances needed to guarantee $SFT(\tau_{exit}) \leq T_R + T_D$ is M'. Let $M' < \lceil \frac{T_{seq}}{T_D} \rceil$. Given M' VM instances, the best scenario is that the work load is evenly distributed to the M' VM instances and all tasks are executed without waiting. Under such scenario, the application's makespan is $T_R + \frac{T_{seq}}{M'}$, which is the earliest possible time the application can complete. Hence, we have $SFT(\tau_{exit}) \geq T_R + \frac{T_{seq}}{M'}$.

Since M' is a positive integer, we have $M' < \frac{T_{seq}}{T_D}$, which implies $T_D < \frac{T_{seq}}{M'}$. From the conclusion $SFT(\tau_{exit}) \geq T_R + \frac{T_{seq}}{M'}$, we have $SFT(\tau_{exit}) > T_R + T_D$, contradicting the assumption that $SFT(\tau_{exit}) \leq T_R + T_D$. □

Lemma 2. *Given a DAG-based deadline constrained application A, let the application's release time and relative end-to-end deadline be T_R and T_D, respectively, its corresponding task graph $G(V, E)$, critical path execution time be T_C, the level of exit task be $Lev(\tau_{exit})$, and the number of VM instances needed to guarantee the application's end-to-end deadline be M. If the application is schedulable, i.e. $T_C \leq T_D$, then we have:*

$$M \leq |V| - Lev(\tau_{exit}) \tag{15}$$

where $|V|$ is the number of tasks in the application. □

Proof. It is obvious that if each task is scheduled to an idle VM instance and each VM instance only executes one task, application A can finish with a makespan of T_C. Since $T_C \leq T_D$, application A can finish before its end-to-end deadline under $|V|$ VM instances. Based on the definition of task's topological level given in Section 3, there must exist a path P_i that consists of at least $Lev(\tau_{exit}) + 1$ tasks and no two tasks are from the same level. Since path P_i must be sequentially executed, dispatching all tasks on P_i to the same VM instance will not affect the application's makespan. Hence, we can at least reduce $Lev(\tau_{exit})$ VM instances from total $|V|$ VM instances. As a result, $M = |V| - Lev(\tau_{exit})$ VM instances are sufficient to guarantee an application's end-to-end deadline. □

Combining Lemma 1 and 2, we have the following theorem that gives the bound on the minimum number of VMs a DAG-application needed to guarantee its deadline.

Theorem 1. *Given a DAG-based deadline constrained application A, let the application's release time and relative end-to-end deadline be T_R and T_D, respectively, its corresponding task graph $G(V, E)$, sequential execution time be T_{seq}, critical path execution time be T_C, level of exit task be $Lev(\tau_{exit})$, and the minimal number of VM instances needed to guarantee the application's end-to-end deadline be M. If the application is schedulable, i.e. $T_C \leq T_D$, we have:*

$$\left\lceil \frac{T_{seq}}{T_D} \right\rceil \leq M \leq |V| - Lev(\tau_{exit}) \tag{16}$$

□

In the next section, we present a heuristic scheduling algorithm based on the theorem.

5 Minimal Slack Time and Minimal Distance (MSMD) based Scheduling

In this section, we introduce a heuristic approach for the resource minimization problem formulated in Section 3.4. The basic idea of our heuristic approach is to search for a schedule that satisfies a given application's deadline using the minimal number of resources given by (14). Once a schedule is found, the number of VM instances used is the least. The search for a possible schedule has two phases: task prioritization phase which is based on application tasks' topological levels and slack time, and task scheduling phase which is based on the minimal distance between the resources' available time and the tasks' ready time.

Given an application task graph G, Algorithm 1 outlines our heuristic approach, where Line 1 prioritizes tasks in the given application; Line 3 to Line 8 search for the minimal resources needed to satisfy the deadline using the *minimal slack time and minimal distance* (MSMD) heuristic scheduling algorithm. We discuss task prioritization and the MSMD scheduling algorithm in the next two subsections.

Algorithm 1: Schedule Searching

```
Input  : Application: $G(V, E), T_R, T_D$
Output: A schedule satisfies $SFT(\tau_{exit}) \leq T_R + T_D$
1 $L \leftarrow prioritize(G)$// Ordered list;
2 $m \leftarrow \lceil \frac{T_{seq}}{T_D} \rceil$;
3 do
4     $S[m] \leftarrow \{\emptyset\}$ // VM job queue;
5     $SFT(\tau_{exit}) \leftarrow MSMD(G, m, L, S)$;
6     $m \leftarrow m + 1$;
7 while $SFT(\tau_{exit}) \leq T_R + T_D$;
8 return $S[m]$
```

5.1 Minimal Slack Time based Prioritization

The goal of task prioritization is to assign a priority to every task in the application. To ensure task dependency relations are not violated, we assign task priorities based on task topological levels and their minimal slack time. In particular, tasks at a lower topological level have higher priorities than tasks at a higher level; for tasks at the same topological level, tasks with less slack time are given higher priorities. If two tasks at the same level have the same slack time, we arbitrarily assign one a higher priority. In general, a task with lower priority value means it has higher priority. The task topological levels and their priorities given in Fig. 1 are shown in Table 1. Tasks are then sorted by their priorities in decreasing order and stored in an ordered list L.

Intuitively, a leveled graph ensures that all predecessor tasks of a task τ_i are scheduled before τ_i. The minimal slack time based priority assignment ensures that the tasks that are more urgent are executed earlier. Once tasks are sorted, starting from minimal number of resources given by (14), we iteratively increase the number of VM instances until a schedule that meets the application's end-to-end deadline is found.

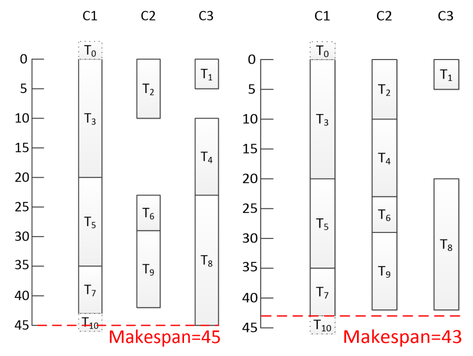
5.2 Minimal Slack time and Minimal Distance Scheduling Algorithm

For a given number of VM instances, the goal of the minimal slack time and minimal distance (MSMD) scheduling algorithm is to schedule a given application to a set of allocated VM instances so that the application's makespan is minimized.

As tasks on a critical path cannot be executed concurrently, hence assigning all critical tasks to the same VM instance does not increase the application's makespan. The question is how to assign tasks that are not on the critical path. One simple approach is to schedule these tasks to the VM instance that has the earliest available time at the time of scheduling. Again, take the application task graph given in Fig. 1 as an example, if we have three VM instances, based on the priority given in Table 1, at time 0, as $av(c_1) = av(c_2) = av(c_3) = 0$, τ_3 is scheduled to c_1, τ_2 to c_2, and τ_1 to c_3, respectively. At time 10, $av(c_3)$ is the least and hence τ_4 is scheduled on c_3. Fig. 2(a) shows the schedule produced by such an approach. We denote this approach as minimal slack and minimal available time based (MSMA) scheduling algorithm.

However, a task τ_i's start time on a VM instance c_j not only depends on the value of $av(c_j)$, but it also depends on its own ready time. Hence, if both VM instances c_j and c_k satisfy $av(c_x) \leq ready(\tau_i)$, task τ_i can be assigned to either one of them. For instance, in our application graph given in Fig. 1, τ_4 can be deployed either on c_2 or c_3. However, the decision may affect the following tasks, such as τ_8 in Fig. 2(a). If τ_4 is scheduled on c_2 rather than being scheduled on the VM instance with the earliest available time (c_3), τ_8 can start at time 20 which can reduce the application's makespan by 3 time units. The two different schedules are depicted in Fig. 2.

We introduce the concept of *distance* into our scheduling algorithm. The distance is used to indicate how close a task's *ready* time is to the VM instance's earliest available time. We



(a) MSMA Approach (b) MSMD Approach

Fig. 2: A Schedule for Task Graph (Fig.1)

formally define the distance ($Dis(\tau_i, c_j)$) between a task τ_i's ready time ($ready(\tau_i)$) and a VM instance c_j's available time ($av(c_j)$) as follows:

$$Dis(\tau_i, c_j) = \begin{cases} av(c_j) & \textit{if } ready(\tau_i) < av(c_j) \\ ready(\tau_i) - av(c_j) & \textit{otherwise} \end{cases} \tag{17}$$

Rather than scheduling tasks to the VM instance with the earliest available time, we assign tasks to the VM instance with minimal distance from its ready time. Doing so will provide more chances for tasks with lower priorities to start at their earliest start time. Furthermore, it allows some non-critical tasks share the same VM instance with critical tasks and hence further reduces application's makespan when other VM instances' earliest available time become larger than the critical task VM instance at the time of scheduling. As shown in Fig. 2(b), with the minimal slack time and minimal distance (MSMD) approach, τ_4 is scheduled on c_2. Hence τ_8 can start at its earliest start time of 20, which in turn reduces the application's makespan to 43 compared to 45 produced by the MSMA approach.

For a given application with task graph $G(V, E)$, m number of VM instances,ordered list L, and empty schedule S the MSMD scheduling algorithm is given in Algorithm 2. In the algorithm, Line 3 to Line 4 assign tasks on the critical path to the same VM instance, i.e. c_0, Line 7 to Line 20 find the VM instance that has the minimal distance from its available time to the current task's ready time. Line 7 to Line 9 and Line 17 to Line 20 enable non-critical tasks to be scheduled on critical task VM instance without interfering critical tasks. The complexity of the MSMD algorithm is $O(|V|^2 m)$.

6 Minimizing Instance Hours

Once the minimum number of virtual machine instances and schedule on each virtual machine instance are determined, our final goal is to find the operation pattern for each VM instance such that the total instance hours is minimized. However, finding such a set is not trivial. Fig. 3 illustrates an example of instance hours needed by different strategies when deciding the operation pattern for a virtual machine instance.

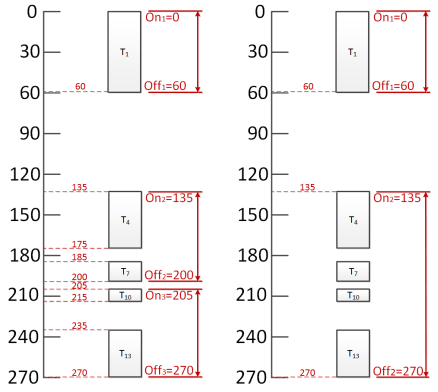
For instance, assume task $\tau_1, \tau_4, \tau_7, \tau_{10}$ and τ_{13} are scheduled on the same virtual machine instance. Task τ_1 is executed from time 0 to 60, τ_4 from 135 to 175, τ_7 from 185 to

Algorithm 2: MSMD(G, m, L, S)

```
1  $T[m] \leftarrow \{0\}$ // $av(c)$ ;
2  $P_c \leftarrow G$'s critical path;
3  for $i \leftarrow 0$ to $|L| - 1$ do
4      if $L[i] \in P_c$ or $m = 1$ then
5          Assign $L[i]$ to $S[0]$
6      end
7      else
8          $T[0] \leftarrow \max\{av(0), T[0]\}$;
9          $minDisComp \leftarrow 0$;
10         $distance \leftarrow Dis(L[i], 0)$;
11         for $j \leftarrow 1$ to $m - 1$ do
12             $T[j] = av(S[j])$;
13             if $Dis(L[i], j) < distance$ then
14                 $distance \leftarrow Dis(L[i], j)$;
15                 $minDisComp \leftarrow j$ ;
16             end
17         end
18         Assign $L[i]$ to $minDisComp$;
19         if $minDisComp = 0$ then
20             $T[0] \leftarrow T[0] + w(L[i])$
21         end
22     end
23 end
24 return $AFT(\tau_{exit})$
```

200, τ_{10} from 205 to 215, and τ_{13} from 235 to 270 as shown in Fig. 3. If the virtual machine instance is powered on at time 0 when the first task τ_1 starts and off at time 270 when the last task τ_{13} finishes, i.e. $\mathcal{P} = \{(0, 270)\}$, then $\lceil \frac{270}{60} \rceil = 5$ instance hours are needed.

As there is a large idle time (60 minutes) between task τ_1 and τ_4, we can save one instance hour if we shut down the virtual machine instance after τ_1 finishes and start the instance again when τ_4 starts to execute.



(a) 5 Instance Hours (b) 4 Instance Hours

Fig. 3: Instance Hours Comparison

However, shutting down the virtual machine during the idle time may not necessarily reduce the instance hours. For another VM operation pattern $\mathcal{P} = \{(0, 60), (135, 200), (205, 270)\}$ shown in Fig. 3(a), although we reduce one instance hour between the execution of task τ_1 and τ_4, the total instance hour is still five. However, the operation pattern $\mathcal{P} = \{(0, 60), (135, 270)\}$ as shown in Fig. 3(b) only requires 4 instance hours. In the remaining of the section, we first study the impact of different virtual machine instance running strategies on the number of instance hours, then we propose an algorithm on minimizing the number of instance hours for a given application.

Lemma 3. *For two consecutive tasks τ_1 and τ_2, assume their execution times on a given virtual instance are e_1 and e_2, respectively, the idle time between τ_1 and τ_2 is b, where $0 \le b < U$ and U is the pricing time unit. If any of the following three conditions holds:*

1) $e_1 \bmod U + e_2 \bmod U + b > 2U$
2) $(e_1 \bmod U = 0 \vee e_2 \bmod U = 0)$ $\wedge(e_1 \bmod U + e_2 \bmod U + b > U)$
3) $e_1 \bmod U = 0 \wedge e_2 \bmod U = 0$

then shutting down the virtual machine after the completion of τ_1 reduces the instance hours.

□

Proof. Let $e_1 = k_1 U + \delta_1$ and $e_2 = k_2 U + \delta_2$, where k_1, k_2 are non-negative integers and $0 \le \delta_1, \delta_2 < U$. If the virtual machine keeps running during the idle time, the total instance hours for executing two tasks will be:

$$\begin{aligned} H_{run} &= \lceil \tfrac{k_1 U + \delta_1 + k_2 U + \delta_2 + b}{U} \rceil \\ &= k_1 + k_2 + \lceil \tfrac{\delta_1 + \delta_2 + b}{U} \rceil \end{aligned} \tag{18}$$

If the virtual machine stops running during the idle time, the total instance hours for executing two tasks will be:

$$\begin{aligned} H_{stop} &= \lceil \tfrac{k_1 U + \delta_1}{U} \rceil + \lceil \tfrac{k_2 U + \delta_2}{U} \rceil \\ &= k_1 + k_2 + \lceil \tfrac{\delta_1}{U} \rceil + \lceil \tfrac{\delta_2}{U} \rceil \end{aligned} \tag{19}$$

If $H_{stop} < H_{run}$, then virtual machine shutting down reduces the instance hours. There are four cases need to be considered.

Case 1: $\delta_1 > 0, \delta_2 > 0$.

$$\begin{aligned} & H_{stop} < H_{run} \\ \Rightarrow \quad & 2 < \lceil \tfrac{\delta_1 + \delta_2 + b}{U} \rceil \Rightarrow \quad 2U < \delta_1 + \delta_2 + b \end{aligned}$$

Case 2: $\delta_1 = 0, \delta_2 > 0$.

$$\begin{aligned} & H_{stop} < H_{run} \\ \Rightarrow \quad & 1 < \lceil \tfrac{\delta_2 + b}{U} \rceil \Rightarrow \quad U < \delta_2 + b \end{aligned}$$

Case 3: $\delta_1 > 0, \delta_2 = 0$.

$$\begin{aligned} & H_{stop} < H_{run} \\ \Rightarrow \quad & 1 < \lceil \tfrac{\delta_1 + b}{U} \rceil \Rightarrow \quad U < \delta_1 + b \end{aligned}$$

Case 4: $\delta_1 = 0, \delta_2 = 0$. H_{stop} always smaller than H_{run}
Combine four cases, we get the lemma 3. □

Lemma 3 indicates whether turning off a VM reduces instance hours when idle time is less than the pricing time unit.

Lemma 4. *For two consecutive tasks τ_1 and τ_2, assume their execution times on a given virtual instance are e_1 and e_2, respectively, the idle time between τ_1 and τ_2 is b, where $U \le b < 2U$ and U is the pricing time unit. If any of the following two conditions holds:*

1) $(e_1 \bmod U + e_2 \bmod U + b \bmod U > U)$
2) $(e_1 \equiv 0 \bmod U \vee e_2 \equiv 0 \bmod U)$

then shutting down the virtual machine after the completion of τ_1 reduces VM instance hours.

□

Proof. Let $e_1 = k_1U + \delta_1$, $e_2 = k_2U + \delta_2$ and $b = U + \delta_3$, where k_1, k_2 are non-negative integers and $0 \le \delta_1 < U, 0 \le \delta_2 < U, 0 \le \delta_3 \le U$. If the virtual machine keeps running during the idle time, the total instance hours for executing two tasks will be:

$$\begin{aligned} H_{run} &= \lceil \tfrac{k_1U+\delta_1+k_2U+\delta_2+U+\delta_3}{U} \rceil \\ &= k_1 + k_2 + 1 + \lceil \tfrac{\delta_1+\delta_2+\delta_3}{U} \rceil \end{aligned}$$

If the virtual machine stops running during the idle time, the total instance hours for executing two tasks is given in equation (19). If the number of instance hours is reduced, then $H_{stop} < H_{run}$ still must hold. We consider four cases.
Case 1: $\delta_1 > 0, \delta_2 > 0$

$$\begin{aligned} &H_{stop} < H_{run} \\ \Rightarrow \quad &2 < 1 + \lceil \tfrac{\delta_1+\delta_2+\delta_3}{U} \rceil \Rightarrow \quad U < \delta_1 + \delta_2 + \delta_3 \end{aligned}$$

Case 2: $\delta_1 = 0, \delta_2 > 0$

$$H_{stop} < H_{run} \Rightarrow \quad 1 < 1 + \lceil \frac{\delta_2 + \delta_3}{U} \rceil$$

Since $\delta_2 > 0$, it is always true that $H_{stop} < H_{run}$.
Case 3: $\delta_1 > 0, \delta_2 = 0$

$$H_{stop} < H_{run} \Rightarrow \quad 1 < 1 + \lceil \frac{\delta_1 + \delta_3}{U} \rceil$$

Since $\delta_1 > 0$, it is always true that $H_{stop} < H_{run}$.
Case 4: $\delta_1 = 0, \delta_2 = 0$.
Always true that $H_{stop} < H_{run}$.

Combine four cases, we obtain lemma 4. □

Lemma 4 indicates whether turning off the VM reduces VM instance hours when the idle time is larger than the pricing time unit but less than two times of the pricing time unit.

Lemma 5. *For two consecutive tasks τ_1 and τ_2, assume their execution times on a given virtual instance are e_1 and e_2, respectively, the idle time between τ_1 and τ_2 is b, where $b \ge 2U$ and U is the pricing time unit. Shutting down the virtual machine after the completion of τ_1 reduces the instance hours.*

□

Proof. Let $e_1 = k_1U + \delta_1$, $e_2 = k_2U + \delta_2$ and $b = k_3U + \delta_3$, where k_1, k_2, k_3 are non-negative integers and $0 \le \delta_1 < U, 0 \le \delta_2 < U, 0 \le \delta_3 < U, k_3 \ge 3$. If the virtual machine keeps running during the idle time, the total instance hours for executing two tasks will be:

$$\begin{aligned} H_{run} &= \lceil \tfrac{k_1U+\delta_1+k_2U+\delta_2+k_3U+\delta_3}{U} \rceil \\ &= k_1 + k_2 + k_3 + \lceil \tfrac{\delta_1+\delta_2+\delta_3}{U} \rceil \\ &\ge k_1 + k_2 + 3 + \lceil \tfrac{\delta_1+\delta_2+\delta_3}{U} \rceil \end{aligned}$$

If the virtual machine stops running during the idle time, the total instance hours for executing two tasks will be:

$$\begin{aligned} H_{stop} &= \lceil \tfrac{k_1U+\delta_1}{U} \rceil + \lceil \tfrac{k_2U+\delta_2}{U} \rceil \\ &= k_1 + k_2 + \lceil \tfrac{\delta_1}{U} \rceil + \lceil \tfrac{\delta_2}{U} \rceil \le k_1 + k_2 + 2 \end{aligned} \tag{20}$$

It is obvious that $H_{stop} < H_{run}$ always holds. □

Lemma 5 indicates turning off the VM reduces VM instance hour when the idle time is two times larger than the charging interval.

Based on Lemma 3, Lemma 4 and Lemma 5, we present an instance hour minimization algorithm, i.e. the IHM as shown in Algorithm 3. As the minimum theoretic instance hours that a virtual machine consumes is when all tasks are executing continues, i.e. there is no idle time between any two consecutive tasks. If a virtual machine instance's operation pattern is turn on at the time the first task starts and off at the time the last task finishes, and under such pattern, the VM instance hours equals to the minimum theoretic instance hours, then we cannot further reduce the instance hours (Line 2 to 5). Otherwise, Lemma 3, Lemma 4 and Lemma 5 are applied to reduce the VM instance hours (Line 8 to 19).

Note that the Lemmas only determine the optimal solution between two consecutive tasks. If the VM instance is not turned off between two consecutive tasks, then we combine the two consecutive tasks as one task (Line 15 to 18). Then apply the Lemmas to the new task and its consecutive task. The time complexity to reduce the instance hours for all the scheduled VM instances is $O(n)$, where n is the number of tasks deployed on the VM instance.

Algorithm 3: Instance Hour Minimization Algorithm

```
Input  : Set of schedule $S = \{S_1, \dots, S_n\}$
Output: $\mathcal{P}_i$ for each schedule $S_i$
1  for $i \leftarrow 1$ to $|S|$ do
2      if $\lceil \frac{\sum_{k=1}^{|S_i|} w(S_i[k])}{C} \rceil = \lceil \frac{SFT(S_i[|S_{c_i}|]) - AST(S_i[1])}{C} \rceil$ then
3          $on \leftarrow$ AST($S_i[1]$), $off \leftarrow$ SFT($S_i[|S_i|]$) ;
4          push $(on, off) \rightarrow \mathcal{P}_i$;
5          continue;
6      end
7      else
8          $\tau_1 \leftarrow S_i[1]$;
9          for $j \leftarrow 2$ to $|S_i|$ do
10             $\tau_2 \leftarrow S_i[j]$, $b =$ AST($\tau_1$) $-$ AFT($\tau_2$);
11             Decide if need to shut down the VM
               during the idle time based on
               Lemma 3,Lemma 4 and Lemma 5;
12             if need to shut down then
13                 push the operation pattern for
                   $\tau_1 \rightarrow \mathcal{P}_i$, $\tau_1 \leftarrow S_i[j]$, $j \leftarrow j + 1$;
14             end
15             else
16                 Combine $\tau_1$ and $\tau_2$ as a new $\tau_1$;
17                 $j \leftarrow j + 1$;
18             end
19         end
20     end
21 end
```

It is worth pointing out that the developed IHM algorithm can be applied to any given schedule, not necessarily the schedule produced by the MSMD scheduling algorithm.

7 Empirical Evaluations

The purpose of the experiments is to evaluate the performance of the developed MSMD algorithm and the IHM algorithm. We evaluate the resource minimization algorithm (MSMD) and the instance hour minimization algorithm (IHM) by comparing it with six other algorithms published in the literature. They are the heterogeneous-earliest-finish-time (HEFT) algorithm [32], the multi-objective-heterogeneous-earliest-finish-time (MO-HEFT) algorithm [16], the duplication-based bottom up scheduling (DBUS) algorithm [9], the QoS-based workflow scheduling (QoS) algorithm [40] and the auto-scaling (Auto) scheduling algorithm [25].

7.1 Experiment Settings

The DAG-based applications we used for the evaluation are obtained from the DAG-based applications benchmark provided by Pegasus WorkflowGenerator [2]. We use four sets of applications from the benchmark, i.e., CyberShake, Laser Interferometer Gravitational Wave Observatory (LIGO), Epigenomics (GENOME), and Montage. The CyberShake applications are highly paralleled applications. The LIGO applications are also highly paralleled, however, they have some critical nodes that have large number of child tasks and parent tasks. Both Epigenomics and Montage applications are combined with parallel execution tasks and sequential tasks. The detailed characteristics of the benchmark applications can be found in [7], [21]. Each set of applications we use for evaluation contains applications with number of tasks ranging ranging from 50 to 1000. Since the benchmark does not specify the deadlines for applications, we assign deadlines for benchmark applications. In order to ensure an application is schedulable, we first calculate its critical path execution time T_C and assign a deadline $T_D \geq T_C$. In order to observe the impact of deadline tightness on scheduling algorithms' performance, we assign five distinct deadlines for each application, i.e., $T_D = T_C$, $T_D = 1.5T_C$, $T_D = 2T_C$, $T_D = 2.5T_C$, and $T_D = 3T_C$.

7.2 Evaluation Criteria

One of the main objectives of the MSMD is to find the minimum number of VM instances needed to complete a given application's execution before its deadline. Hence, the first criterion to evaluate the performance of an algorithm is how many VM instances it uses to ensure an application finishes before its end-to-end deadline. We introduce the concept of *resource reduction rate* to indicate how many resources are reduced from the resource upper bound given by (15). It is defined as below:

$$\textit{Res. Red. Rate} = \frac{\text{Upper Bound - Actual Res. Used}}{\text{Upper Bound}} \tag{21}$$

The second goal of the MSMD is to minimize the makespan of an application under the given minimal resources. One way to evaluate the effectiveness of minimizing makespan is to see how much time is reduced from an application's deadline. We define *makespan reduction rate* for the evaluation. For a given application A with its release time T_R and relative deadline T_D, the makespan reduction rate is defined as:

$$\textit{MS Red. Rate} = \frac{T_R + T_D - SFT(\tau_{exit})}{T_D} \tag{22}$$

where $SFT(\tau_{exit})$ is the actual finish time of application A.

The ultimate goal of our work is to minimize the instance hours that an application needed to complete its execution. Since in homogeneous environment, when all the tasks in an application are sequentially executed, the instance hours it consumed is minimized. Hence, to evaluate the performance of instance hour minimization algorithms is to see how many instance hours increased from its minimum instance hours needed. We define *instance hour increase rate* for the evaluation. For a given application A, the *instance hour increase rate* is defined as:

$$\text{IH Inc. Rate} = \frac{\text{IH}(A) - \lceil T_{seq}/U \rceil}{\lceil T_{seq}/U \rceil} \tag{23}$$

where U is the pricing time unit defined in Section 3.2.

7.3 Comparison with the Optimal Solution

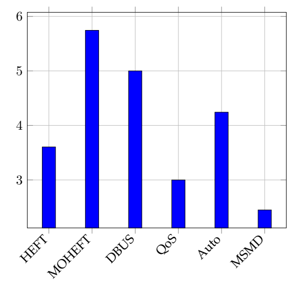
Since the proposed MSMD scheduling algorithm is a heuristic algorithm, the most straightforward way to evaluate its performance is to compare with the optimal solution when possible. We randomly generate 100 different applications and obtain the optimal solutions by exhaustively searching for all possible resource allocations that meet the application's deadline. Since the scheduling problem is NP-complete, to exhaustively search for the optimal solutions, we can only deal with random applications that are of small task size (8 tasks to 10 tasks). For each application, we first randomly generate tasks. Second, we randomly generate the number of levels an application has, ranging from 3 to its task size. We then randomly assign tasks to levels and connect them as a DAG. At last, we assign a random deadline from one time of its critical path length to three times of its critical path length.

We calculate the standard deviation of the number of VM instances needed by different algorithms from the optimal solutions found by exhaustive search. As shown in Fig. 4(a), the MSMD algorithm has the least standard deviation from the optimal solution. This implies that the MSMD algorithm can guarantee applications' end-to-end deadlines with the number of VM instances that is close to the optimal.

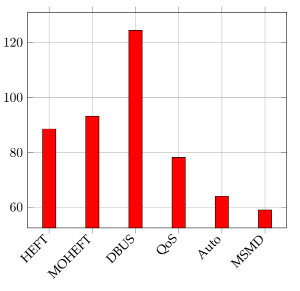
We also calculate the makespan standard deviation, since heuristic algorithms may need more VM instances to guarantee applications' end-to-end deadlines than the optimal solution. When extra VM instances are used for scheduling, it is possible that the applications' makespan are smaller than the makespan produced by the optimal solution. Hence, the calculation of makespan standard deviation only considers the cases when the heuristic algorithms use the same number of VM instances as the exhaustive search algorithm. As shown in Fig. 4(b), the MSMD algorithm has the smallest standard deviation from the optimal solution on both minimum number of VM instances needed and applications' makespan. It indicates that the performance of the MSMD algorithm is close to optimal.

7.4 Comparison among Different Heuristic Algorithms

In the previous subsection, as we need to find the optimal solutions through exhaustive search, both sample size and the application size are limited. In this section, we extend the test scale. In particular, we use four sets of benchmark DAG-based applications obtained from [2], and apply the



(a) S.D Comparison of Min Number of VM instances



(b) S.D Comparison of Makespans

Fig. 4: Standard Deviation Comparison

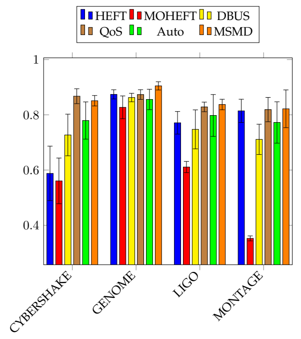


Fig. 7: Resource Red. Rate for Application with $T_D = 1.5T_C$

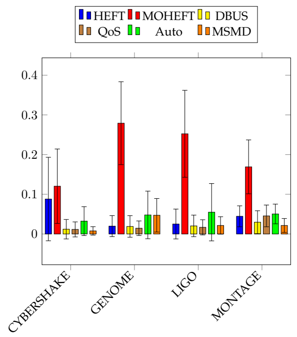


Fig. 8: Makespan Red. Rate for Application with $T_D = 1.5T_C$

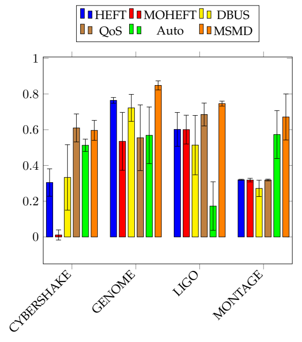


Fig. 5: Resource Red. Rate for Application with $T_D = T_C$

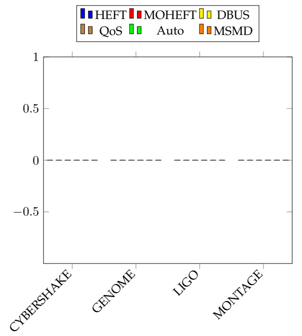


Fig. 6: Makespan Red. Rate for Application with $T_D = T_C$

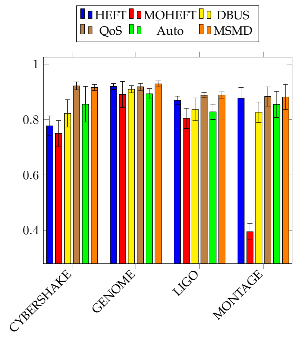


Fig. 9: Resource Red. Rate for Application with $T_D = 2T_C$

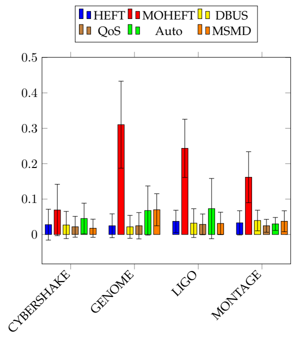


Fig. 10: Makespan Red. Rate for Application with $T_D = 2T_C$

six different algorithms, i.e. the HEFT, MOHEFT, DBUG, QoS, Auto and our MSMD algorithms, to these test cases.

Fig. 5 and Fig. 6 depict the average resource reduction rate and average makespan reduction rate of different application sets with deadline equals to application's critical path execution time under the six scheduling algorithms, respectively. The difference bar represents the standard deviation of the all applications' resource reduction rate. As shown in Fig. 6, there is no surprise that the makespan reduction rates for all six scheduling algorithms are zero. However, as Fig. 5 indicates, different algorithms use different number of VMs to achieve guarantee the applications' deadlines. It is not difficult to see from the figures that the MSMD uses the least number of VMs to guarantee the applications' deadlines in GENOME, LIGO and Montage applications. For the CyberShake applications, the QoS uses the least number of VMs to guarantee the applications' deadlines. The MSMD ranked second with only 1.4% difference with the QoS algorithm. If we take the error into consideration, the MSMD (0.0559) performs more steady than the QoS algorithm (0.0783).

Fig. 7, Fig. 9, Fig. 11, and Fig. 13 illustrate the applications' resource reduction rate under different scheduling algorithms when the application's deadline equals to 1.5, 2, 2.5 and 3 times of its critical path execution time, respectively. Fig. 8, Fig. 10, Fig. 12, and Fig. 14 show the applications' makespan reduction rate under different scheduling algorithms when the application's deadline equals to 1.5, 2, 2.5, and 3 times of its critical path execution time, respectively. These figures indicate that the patterns of the resource reduction rate and makespan reduction rate for

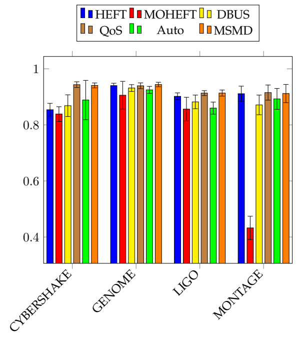


Fig. 11: Resource Red. Rate for Application with $T_D = 2.5T_C$

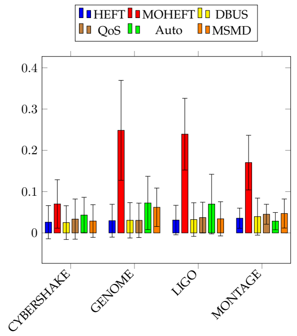


Fig. 12: Makespan Red. Rate for Application with $T_D = 2.5T_C$

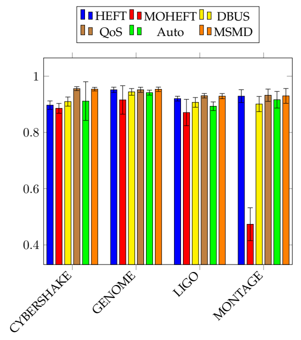


Fig. 13: Resource Red. Rate for Application with $T_D = 3T_C$

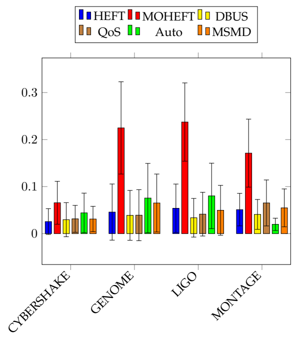


Fig. 14: Makespan Red. Rate for Application with $T_D = 3T_C$

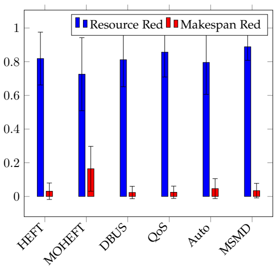


Fig. 15: Overall Performance

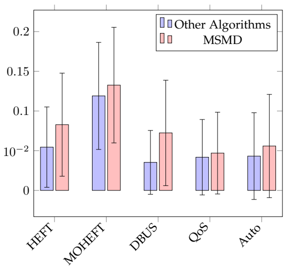


Fig. 16: Overall Makespan Red. Comparison

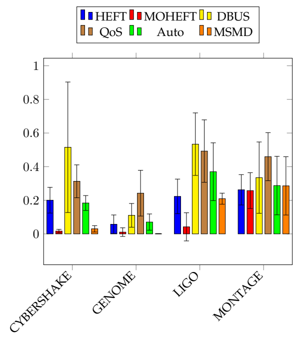


Fig. 17: Instance Hour Inc. Rate for Application with $T_D = T_C$

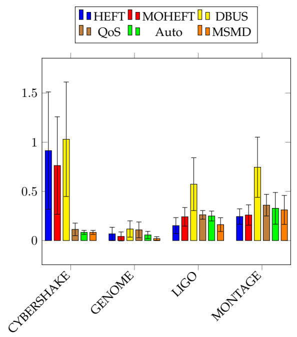


Fig. 18: Instance Hour nc. Rate for Application with $T_D = 1.5T_C$

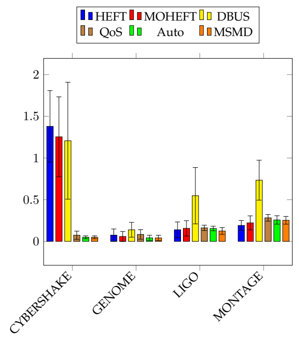


Fig. 19: Instance Hour Inc. Rate for Application with $T_D = 2T_C$

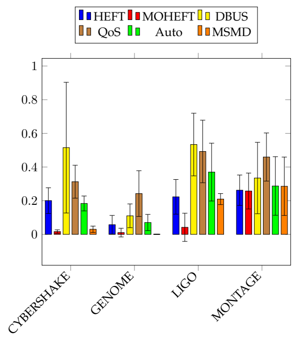


Fig. 20: Instance Hour Inc. Rate for Application with $T_D = 2.5T_C$

the six scheduling algorithms are the same when different deadlines are assigned to the same application. The resource reduction rate increases when the applications' deadlines are less tight.

From Fig. 8, Fig. 10, Fig. 12, and Fig. 14, we can also see that the MOHEFT can reduce much more makespan from applications' deadlines compared with all other scheduling algorithms. However, Fig. 7, Fig. 9, Fig. 11, and Fig. 13, also indicate that the MOHEFT uses much more VMs to schedule the applications. For large applications such as GENOME, LIGO and Montage, the upper bound for resources to guarantee an application's deadline is usually over 1000 VMs. One percent resource reduction rate difference means increase 10 more VMs. Hence, it is expected that when MOHEFT uses much more VMs to schedule applications, it can reduce the application's makespan. Since the primary goal of the paper is to minimize the number of resources that guarantee the applications' deadlines, hence the MOHEFT is not competitive comparing to other scheduling algorithms even though it has better performance on reducing the applications' makespan. For the other five algorithms, the MSMD has the largest resource reduction rate on most of the scenarios. The QoS algorithm has the similar performance on resource reduction rate as the MSMD.

Fig. 15 shows the average resource reduction rate and the average makespan reduction rate comparison on all tested DAG-based applications. It clearly indicates that the MSMD has the largest resource reduction rate. On average, it can further reduce 3.2% more resources than the QoS algorithm. The MOHEFT algorithm has the largest makespan reduction rate. However, it uses 7% more resources than the other algorithms. Overall, the MSMD has a 89% average resource reduction rate and a 3.4% average makespan reduction rate.

As discussed above, it is unfair to compare the algorithms' performance on reducing application's makespan if they are using different number of resources. In order to give an unbiased comparison of the six algorithms' performance on makespan reduction, we have compared each scheduling algorithms' makespan reduction rate to the MSMD algorithm under the constrain that they use the same number of VMs. The results are depicted in Fig. 16. It is clearly shown that when using the same number of VMs, on average, the MSMD can reduce more makespan compared with the other heuristic algorithms.

7.5 Instance Hour Minimization Evaluation

In order to fully evaluate the performance of the developed MSMD algorithm, we have to consider two aspects. The first is how well the VM instances minimization method reduces total VM instance hours. We evaluate this aspect by comparing instance hour reductions resulted by the MSMD with five other scheduling algorithms existed in the literature. The second aspect that needs to be evaluated is how many instance hours reduced by the IHM algorithm compared to the instance hours consumed by VMs when the VMs are not turned off until the last task is finished.

Fig. 17, Fig. 18, Fig. 19, Fig. 20, and Fig. 21 depict the average instance hour increase rate comparison when application's deadline equals to 1, 1.5, 2, 2.5, and 3 times of the application's critical path execution time, respectively. For GENOME applications, all the compared algorithms perform well in reducing VM instance hours. However, among all compared algorithms, the MSMD algorithm has

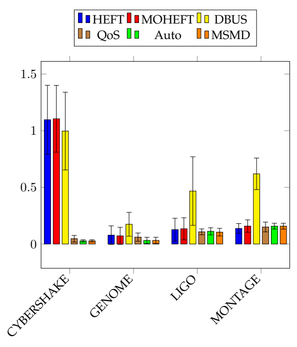


Fig. 21: Instance Hour Inc. Rate for Application with $T_D = 3T_C$

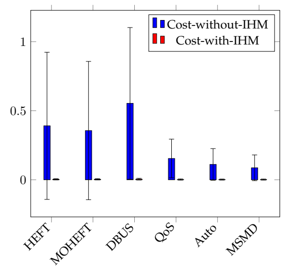


Fig. 22: Overall Instance Hour Inc. Comparison

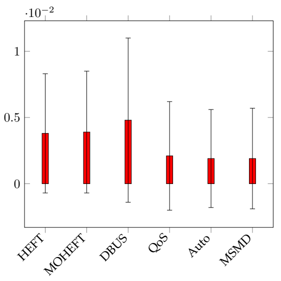


Fig. 23: Overall Instance Hour Inc. Comparison with Performing IHM

the least instance hour increase rate. The MSMD also has small instance hour increase rate on other sets of applications.

The experiment results also reveal that the MOHEFT algorithm has the least instance hour increase rate on LIGO applications when the applications' deadline equal to its critical path execution time and 2.5 times of its critical path execution time. However, it has very large instance hour increase rate on CyberShake applications. As a result, as shown n Fig. 22, it ranks only number four on instance hour minimization out of six algorithms.

Fig. 22 illustrate the average instance hour increase rate on all applications. The MSMD has the least instance hour increase rate (8%). The Auto-scaling algorithm ranks second with 11.1% instance hour increase rate. The QoS algorithm ranks the third with 15.43% instance hour increase rate.

The IHM developed in this paper aims to further reduce the instance hours by deciding the shutting down pattern during the VM idle time. It can be applied to any given scheduling algorithms. To evaluate whether the IHM can indeed help to reduce the instance hours, we apply the IHM to all six scheduling algorithms after they generated their schedule. As illustrated in Fig. 22 and Fig. 23, the IHM indeed further reduces the instance hours of a given schedule. This is because the tasks in the benchmark applications are all with large execution times. Hence the idle time between the two tasks also become very large, by shutting down the VMs during its idle time can sometimes significantly reduce the instance hours the application consumed on VMs. However, for applications with small execution time tasks, the effectiveness of the IHM algorithm in further reducing VM instance hours may be limited.

8 Conclusion

In this paper, we have addressed the issue of how to deploy deadline constrained applications to computer clouds so that (1) deadline constrained application's end-to-end deadline is guaranteed, (2) the number of VM instances allocated to the application is minimized, (3) under the allocated minimum number of VM instances, the application's makespan is minimized, and (4) under the given schedule, the total VM instance hour is minimized. We have shown that there is a lower and an upper bounds on the number of VM instances needed to guarantee a given real-application's deadline. Based on the bounds, we have developed a *minimal slack time and minimal distance* (MSMD) heuristic task deployment and scheduling algorithm that finds the minimum number of resources needed to guarantee an application's deadline and also minimizes the makespan of the application under the allocated VM instances. The time complexity for the MSMD is only $O(|V|^2 m)$. We further study VM instance hour minimization problem. Based on the theoretical analysis, we have developed the IHM heuristic cost minimization algorithm. The simulation results show that the heuristic MSMD algorithm can guarantee applications' end-to-end deadline with less resources compared with other heuristic scheduling algorithms and can on average reduce applications' makespans by 3.4% of their deadlines under the allocated resources. When the IHM algorithm is applied to a given application task execution schedul, the VM instance hours needed by the application can further be reduced.

However, in our current work, we have made three assumptions: (1) VM instances in the cloud are homogeneous, (2) there are no communications among application tasks, and (3) there is no overhead when power on and off virtual machines. Our immediate future work is to study deadline constrained application resource needs when these three assumptions are removed. In addition, the current task model is based on task's execution times (WCET) which is difficult to accurately predict in cloud envirnment. Another line of future work is to study the virtualization overhead in cloud environment and its impact on task's execution time so that the developed algorithm can be improved.

References

[1] Cps project on managing loosely coupled networked control systems with external disturbances: Wastewater processing. In *http://www.cs.iit.edu/ winet/CPS2011/index.html*.

[2] https://confluence.pegasus.isi.edu/display/pegasus/ workflow-generator.

[3] http://www.marketresearchmedia.com/?p=839.

[4] Interactive real-time elearning. www.irmosproject.eu.

[5] M. A. Al-Mouhamed. Lower bound on the number of processors and time for scheduling precedence graphs with communication costs. *Software Engineering, IEEE Transactions on*, 16(12):1390–1401, 1990.

[6] G. B. Berriman, G. Juve, E. Deelman, M. Regelson, and P. Plavchan. The application of cloud computing to astronomy: A study of cost and performance. In *e-Science Workshops, 2010 Sixth IEEE International Conference on*, pages 1–7. IEEE, 2010.

[7] S. Bharathi, A. Chervenak, E. Deelman, G. Mehta, M.-H. Su, and K. Vahi. Characterization of scientific workflows. In *Workflows in Support of Large-Scale Science, 2008. WORKS 2008. Third Workshop on*, pages 1–10. IEEE, 2008.

[8] L. F. Bittencourt and E. R. M. Madeira. Hcoc: a cost optimization algorithm for workflow scheduling in hybrid clouds. *Journal of Internet Services and Applications*, 2(3):207–227, 2011.

[9] D. Bozdag, U. Catalyurek, and F. Ozguner. A task duplication based bottom-up scheduling algorithm for heterogeneous environments. In *Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International*, pages 12–pp. IEEE, 2006.

[10] R. N. Calheiros and R. Buyya. Cost-effective provisioning and scheduling of deadline-constrained applications in hybrid clouds. In *Web Information Systems Engineering-WISE 2012*, pages 171–184. Springer, 2012.

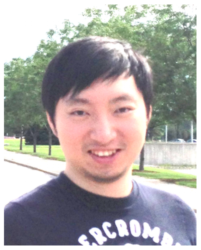
[11] H. Cao, H. Jin, X. Wu, and S. Wu. Serviceflow: Qos-based hybrid service-oriented grid workflow system. *The Journal of Supercomputing*, 53(3):371–393, 2010.

[12] K. Chadwick. Fermigrid and fermicloud update. *In 2012 International Symposium on Grids and Clouds*, 2012.

[13] N. Chopra and S. Singh. Survey on scheduling in hybrid clouds. In *Computing, Communication and Networking Technologies (ICCCNT), 2014 International Conference on*, pages 1–6. IEEE, 2014.

[14] A. P. E. Coffman Jr and R. L. Graham. Optimal scheduling for two-processor systems. *Acta Informatica*, 1(3):200–213, 1972.

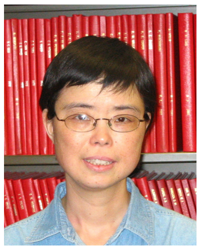
[15] F. Ding, R. Zhang, K. Ruan, J. Lin, and Z. Zhao. A qos-based scheduling approach for complex workflow applications. In *ChinaGrid Conference (ChinaGrid), 2010 Fifth Annual*, pages 67–73. IEEE, 2010.
[16] J. J. Durillo, R. Prodan, and H. M. Fard. Moheft: a multi-objective list-based method for workflow scheduling. In *Proceedings of the 2012 IEEE 4th International Conference on Cloud Computing Technology and Science (CloudCom)*, pages 185–192. IEEE Computer Society, 2012.
[17] M. R. Garey and D. S. Johnson. *Computers and intractability*, volume 174. Freeman San Francisco, CA, 1979.
[18] R. L. Graham. Bounds for certain multiprocessing anomalies. *Bell System Technical Journal*, 45(9):1563–1581, 1966.
[19] T. C. Hu. Parallel sequencing and assembly line problems. *Operations research*, 9(6):841–848, 1961.
[20] S. Jayadivya and S. M. S. Bhanu. Qos based scheduling of workflows in cloud computing. *International Journal of Computer Science and Electrical Engineering (IJCSEE) ISSN*, (2315-4209).
[21] G. Juve, A. Chervenak, E. Deelman, S. Bharathi, G. Mehta, and K. Vahi. Characterizing and profiling scientific workflows. *Future Generation Computer Systems*, 29(3):682–692, 2013.
[22] K. Keahey, R. Figueiredo, J. Fortes, T. Freeman, and M. Tsugawa. Science clouds: Early experiences in cloud computing for scientific applications. *Cloud computing and applications*, 2008, 2008.
[23] L. M. Khanli and M. Analoui. Qos-based scheduling of workflow applications on grids. In *Proceedings of the Third Conference on IASTED International Conference: Advances in Computer Science and Technology*, ACST'07, pages 276–282, Anaheim, CA, USA, 2007.
[24] M. Maheswaran, S. Ali, H. Siegal, D. Hensgen, and R. F. Freund. Dynamic matching and scheduling of a class of independent tasks onto heterogeneous computing systems. In *Heterogeneous Computing Workshop, 1999.(HCW'99) Proceedings. Eighth*, pages 30–44. IEEE, 1999.
[25] M. Mao and M. Humphrey. Auto-scaling to minimize cost and meet application deadlines in cloud workflows. In *High Performance Computing, Networking, Storage and Analysis (SC), 2011 International Conference for*, pages 1–12. IEEE, 2011.
[26] M. Mao, J. Li, and M. Humphrey. Cloud auto-scaling with deadline and budget constraints. In *Grid Computing (GRID), 2010 11th IEEE/ACM International Conference on*, pages 41–48. IEEE, 2010.
[27] M. Mezmaz, N. Melab, Y. Kessaci, Y. C. Lee, E.-G. Talbi, A. Y. Zomaya, and D. Tuyttens. A parallel bi-objective hybrid metaheuristic for energy-aware scheduling for cloud computing systems. *Journal of Parallel and Distributed Computing*, 71(11):1497–1508, 2011.
[28] Netflix. Netflix, 2013. http://www.netflix.com/.
[29] L. Ramakrishnan, J. S. Chase, D. Gannon, D. Nurmi, and R. Wolski. Deadline-sensitive workflow orchestration without explicit resource control. *Journal of Parallel and Distributed Computing*, 71(3):343–353, 2011.
[30] C. Ramamoorthy, K. Chandy, and M. J. Gonzalez. Optimal scheduling strategies in a multiprocessor system. *Computers, IEEE Transactions on*, 100(2):137–146, 1972.
[31] TEMENOS. Temenos, 2013. www.temenos.com.
[32] H. Topcuoglu, S. Hariri, and M.-y. Wu. Performance-effective and low-complexity task scheduling for heterogeneous computing. *Parallel and Distributed Systems, IEEE Transactions on*, 13(3):260–274, 2002.
[33] R. Van den Bossche, K. Vanmechelen, and J. Broeckhove. Cost-optimal scheduling in hybrid iaas clouds for deadline constrained workloads. In *Cloud Computing (CLOUD), 2010 IEEE 3rd International Conference on*, pages 228–235. IEEE, 2010.
[34] R. Van den Bossche, K. Vanmechelen, and J. Broeckhove. Cost-efficient scheduling heuristics for deadline constrained workloads on hybrid clouds. In *Cloud Computing Technology and Science (CloudCom), 2011 IEEE Third International Conference on*, pages 320–327. IEEE, 2011.
[35] R. Van den Bossche, K. Vanmechelen, and J. Broeckhove. Online cost-efficient scheduling of deadline-constrained workloads on hybrid clouds. *Future Generation Computer Systems*, 29(4):973–985, 2013.
[36] H. Wu, S. Ren, G. Garzoglio, S. Timm, G. Bernabeu, H. W. Kim, K. Chadwick, H. Jang, and S.-Y. Noh. Automatic cloud bursting under fermicloud. In *Parallel and Distributed Systems (ICPADS), 2013 International Conference on*, pages 681–686. IEEE, 2013.
[37] J. Yu and R. Buyya. A budget constrained scheduling of workflow applications on utility grids using genetic algorithms. In *Workflows in Support of Large-Scale Science, 2006. WORKS'06. Workshop on*, pages 1–10. IEEE, 2006.
[38] J. Yu and R. Buyya. Scheduling scientific workflow applications with deadline and budget constraints using genetic algorithms. *Scientific Programming*, 14(3):217–230, 2006.
[39] J. Yu, R. Buyya, and C. Tham. Cost-based scheduling of scientific workflow applications on utility grids. In *e-Science and Grid Computing, 2005. First International Conference on*, pages 8–pp. IEEE, 2005.
[40] J. Yu, R. Buyya, C. K. Tham, et al. Qos-based scheduling of workflow applications on service grids. In *Proc. of 1st IEEE International Conference on e-Science and Grid Computing*, 2005.



Hao Wu is now a Ph.D candidate in Computer Science Department at Illinois Institute of Technology. He received B.E in Information Security from Sichuan University, Chengdu, China, 2007. He received M.S. in Computer Science from University of Bridgeport, Bridgeport, CT, 2009. His current research interests mainly focus on cloud computing, real-time distributed open systems, Cyber-Physical System, parallel and distributed systems, and real-time applications.

Xiayu Hua is a Ph.D. student in the Computer Science Department at Illinois Institute of Technology. His research interest is in distributed file system, virtualization technology, real-time scheduling and cloud computing. He earned his B.S. degree from the Northwestern Polytechnic University, China, in 2008 and his M.S. degree from the East China Normal University, China, in 2012.

Zheng Li received the B.S. degree in Computer Science and M.S. degree in Communication and Information System from University of Electronic Science and Technology of China, in 2005 and 2008, respectively. He is currently a Ph.D. candidate in the Department of Computer Science at the Illinois Institute of Technology. His research interests include real-time embedded and distributed systems.



Dr. Shangping Ren is an associate professor in Computer Science Department at the Illinois Institute of Technology. She earned her Ph.D from UIUC in 1997. Before she joined IIT in 2003, she worked in software and telecommunication companies as software engineer and then lead software engineer. Her current research interests include coordination models for real-time distributed open systems, real-time, fault-tolerant and adaptive systems, Cyber-Physical System, parallel and distributed systems, cloud computing, and application-aware many-core virtualization for embedded and real-time applications.