Deadline-constrained workflow scheduling in IaaS clouds with multi-resource packing

Zhu, Zhaomeng; Tang, Xueyan

2019

Zhu, Z. & Tang, X. (2019). Deadline-constrained workflow scheduling in IaaS clouds with multi-resource packing. *Future Generation Computer Systems*, 101, 880-893.
<https://dx.doi.org/doi.org/10.1016/j.future.2019.07.043>

<https://hdl.handle.net/10356/150516>

<https://doi.org/doi.org/10.1016/j.future.2019.07.043>

© 2019 Elsevier B.V. All rights reserved. This paper was published in *Future Generation Computer Systems* and is made available with permission of Elsevier B.V.

Downloaded on 26 Apr 2023 23:26:11 SGT

Deadline-constrained Workflow Scheduling in IaaS Clouds with Multi-resource Packing

Zhaomeng Zhu, Xueyan Tang

School of Computer Science and Engineering, Nanyang Technological University, Singapore

Abstract

Workflow is a common model to represent large computations composed of dependent tasks. Most existing workflow scheduling algorithms use computing resources in a non-multiprogrammed way, by which only one task can run on a service (machine) at a time. In this paper, we study a new workflow scheduling model on heterogeneous Infrastructure-as-a-Service (IaaS) platforms, which allows multiple tasks to run concurrently on a virtual machine (VM) according to their *multi-resource demands*. First, we propose a list-scheduling framework for the new multiprogrammed cloud resource model. In the order of a priority list, this framework gradually appoints tasks the best placements found on both existing and new VMs on the platform. Different task prioritization and placement comparison methods can be employed for different scheduling objectives. To fully exploit the heterogeneity of IaaS platforms, the VMs can be scaled up during the scheduling process. Then, we propose a deadline-constrained workflow scheduling algorithm (called DyDL) based on this framework to optimize the cost of workflow execution. This algorithm prioritizes tasks by their *latest start times* and appoints tasks the placements which can meet their latest start times and incur the minimal *cost increases*. Experimental results show that DyDL can achieve significantly better schedules in most test cases compared to several existing deadline-constrained workflow scheduling algorithms.

Keywords: Infrastructure as a Service, Directed Acyclic Graph (DAG), workflow scheduling, multi-resource packing.

1. Introduction

In scientific computing, workflow has frequently been used to model large applications composed of dependent tasks. The problem of scheduling workflows in traditional environments like grids and clusters has been extensively studied over many years. Recently, with the widespread adoption of cloud computing, the research in workflow scheduling has been shifting to executing workflows on Infrastructure as a Service (IaaS) platforms, which can provide on-demand Virtual Machines (VMs) on a pay-as-you-go basis.

The goal of workflow scheduling is to find proper mappings from the tasks to a set of parallel services (machines) which provide computing resources. Basically, there are two ways of using the services [1]. In the *non-multiprogrammed* use, a scheduler can assign only a single task to run on a service at a time, while in the *multiprogrammed* use, multiple tasks can be placed on the same service to run simultaneously.

So far, most existing workflow scheduling algorithms assume the non-multiprogrammed use of resources [1, 2, 3, 4]. Nowadays, multi-core processors are common, making the computing resources in a service abundant. To fully use such services, recent studies usually assume the tasks are moldable [5, 6, 7, 8]. That is, a task can be perfectly divided and executed using arbitrary number of cores, and its performance is directly related to the total processing capacity of the service (e.g., number of CPU cores or total GFLOPS). However, the tasks in

workflows are not always moldable. For example, some characterizations of typical workflow applications [9, 10, 11] have shown that most tasks in these workflows can only make use of one or two CPU cores. If a task cannot fully use the computing resources provided by a service, having it exclusively occupying the entire service will lead to resource *over-provisioning* [12]. Therefore, the multiprogrammed use of services is becoming increasingly important.

There are few studies allowing multiple tasks to share a service in the multiprogrammed manner [13, 14]. These studies consider a single type of resources only—tasks are assumed to occupy some CPU cores of a service during their execution. In real-world workflows, the tasks may have demands for not only the processors but also many other types of resources, such as memory and I/O bandwidth. Not considering all relevant resource demands may result in resource *under-provisioning*, in which the available resources of some types are less than needed [15]. Moreover, these studies usually assume the services are homogeneous, and all tasks have the same demands of one CPU core. However, today's IaaS platforms can be heterogeneous since they usually provide a large set of VM types for customers to choose, and different tasks may have widely different resource demands.

In this paper, we present a new workflow scheduling model on heterogeneous IaaS platforms. In our model, the execution of a task uses certain quantities of multiple types of computing resources, which are considered as its *multi-resource demands*. Different tasks may have different multi-resource demands, and multiple tasks can be packed onto a VM in the mul-

Email addresses: zhaomeng.zhu@gmail.com (Zhaomeng Zhu), ASXYTang@ntu.edu.sg (Xueyan Tang)

tiprogrammed manner as long as the resources in the VM are sufficient. The VMs are dynamically acquired and released, and they can be of various types comprising different capacities of multiple resource types; however, there might be limits on the quantities of concurrently used VMs. Workflow-oriented optimization criteria, including the makespan of workflow execution and the total monetary cost of renting VMs, can be considered. We first develop a list scheduling framework for the new model and then propose a deadline-constrained workflow scheduling algorithm which aims to minimize the workflow execution cost.

Our work is partially inspired by recent research on multi-resource packing of tasks in distributed systems. This line of research has shown that resource under/over-provisioning can be substantially reduced by properly packing multiple tasks onto servers according to their multi-resource demands. However, research along this direction usually pays more attention to the fairness among different users and the overall resource utilization of systems [16, 17, 18, 19], which is different from the typical concerns of workflow scheduling (makespan and execution cost). Meanwhile, although there has been much work on multi-resource packing for independent tasks [20, 21, 22], little research in this field considers the dependencies among tasks. Moreover, most existing multi-resource schedulers are designed for systems composed of fixed number and types of servers [15, 23, 24]. In contrast, we schedule workflows on the elastic IaaS platforms, where the tasks are hosted by the VMs dynamically acquired and released. To the best of our knowledge, our work is the first to introduce multi-resource packing to the workflow scheduling in clouds.

The main contributions of this paper include:

1. We present a new multi-resource workflow scheduling model on heterogeneous IaaS platforms.
2. We develop a list-scheduling framework for scheduling workflows with tasks having varying multi-resource demands on multiprogrammed cloud resources.
3. We propose a deadline-constrained workflow scheduling algorithm based on the above framework to optimize the cost of workflow execution. Extensive simulation experiments demonstrate the effectiveness of our algorithm.

The rest of this paper is organized as follows. Section 2 reviews the related work. Section 3 presents the new scheduling model. Section 4 presents a list-scheduling framework for the new problem. Section 5 presents a deadline-constrained workflow scheduling algorithm based on the framework. Section 6 presents and discusses the experimental results. Section 7 concludes the paper.

2. Related Work

2.1. Workflow Scheduling

Different workflow scheduling algorithms have been proposed for different goals and restrictions. Heterogeneous Earliest Finish Time (HEFT) [3] might be the most famous list

heuristic which aims to minimize the makespan of workflow execution in heterogeneous environments. It first topologically sorts the tasks in the workflow into a scheduling list and then assigns each task to the service which can finish it earliest in the order of the list. Recently, more scheduling criteria such as execution cost are considered by researchers [25, 26, 6, 27].

In [28], a Partial Critical Paths (PCP) heuristic is proposed to address the deadline-constrained workflow scheduling problem in grids. This algorithm recursively distributes the overall deadline across the critical and partial critical paths of the workflow, and then, the tasks are scheduled to the cheapest services that can finish them before their sub-deadlines. In [29], IaaS Cloud-Partial Critical Path (IC-PCP) further adapts PCP to IaaS platforms. Different from PCP, having determined a (partial) critical path, IC-PCP directly schedules the entire path to the cheapest applicable VM instead of assigning sub-deadlines to the tasks on the path. Also, several meta-heuristics have been developed to solve the deadline-constrained workflow scheduling problem, although they are usually more time-consuming than heuristics. For example, in [5], Rodriguez and Buyya use Particle Swarm Optimization (PSO) to address the workflow scheduling problem in clouds, and in [7], a Coevolutionary Genetic Algorithm with Adaptive penalty function (CGA²) is proposed. However, all the above work assumes the non-multiprogrammed use of services and does not consider the multi-resource demands of tasks. In this paper, we study workflow scheduling with multi-resource packing of tasks. We shall experimentally compare our proposed solution with the above work.

In [13], Lee and Zomaya propose algorithms which allow each task to occupy a CPU core instead of an entire service. With this finer-grained strategy, these algorithms can reduce resource usage without increasing the makespan of workflow execution. Using the same model, they further propose a Maximum Effective Reduction (MER) algorithm in [14], which trades the minimal makespan increase for the maximal resource usage reduction. However, as aforementioned, these studies assume that the services are homogeneous and all tasks have the same resource demands of only one CPU core.

2.2. Multi-resource Packing

Recently, many multi-resource schedulers have been proposed for packing tasks in distributed systems. Notably, Dominant Resource Fairness (DRF) [16] can achieve fairness and better system throughput by equalizing the dominant resource share of each user, which is the maximum share of any type of resources allocated to the user. Motivated by DRF, resource allocation algorithms are further proposed for various computing environments [18, 19, 30, 17]. Nevertheless, all these algorithms consider independent tasks and ignore the potential dependencies among tasks.

In [15], a new multi-resource scheduler Graphene is presented to schedule DAG-modeled jobs. This scheduler identifies the tasks that run for long time or are hard to pack and place them first; then, the remaining tasks are filled into the resultant holes in the resource-time space. However, Graphene is

designed for clusters composed of fixed servers and thus cannot be directly used to schedule tasks on the elastic IaaS platforms.

3. Problem and Preliminaries

3.1. Cloud Resource Model

The IaaS platforms deliver computing resources to customers via virtual machines (VMs). Usually, a broad range of VM types comprising different combinations of resources (e.g., CPU numbers and memory sizes) are provided. We define a set $P = \{p_1, p_2, \dots, p_m\}$ to denote all the available VM types and use a k -tuple $C_p = (r_p^1, r_p^2, \dots, r_p^k)$ to denote the capacities of various computing resources (e.g., CPU cores, memory, disk, network bandwidth) provided by a VM of type p , where k is the number of resource types considered.

Pricing Model: VMs are charged for their running times from acquisition to release, and the running time is usually rounded up to the next *charging unit* defined by the pricing scheme of the platform. For example, Amazon Elastic Compute Cloud (EC2) charges customers by *instance-hour*, where every partially used hour is charged as a full hour. We denote the charging unit by CU and the per-unit price of a p -type VM by $price_p$. Then, the cost of running a p -type VM for a period τ is

$$Cost(p, \tau) = price_p \times \left\lceil \frac{\tau}{CU} \right\rceil. \quad (1)$$

Resource Limits: The IaaS platforms may have default limits on the quantities of computing resources that a customer can use at the same time. These limits occur in different forms. Most platforms restrict the numbers of VMs that a user can use concurrently [31, 32, 33]; some of them have more specific quantity limits on particular types of VMs [31]; moreover, some platforms set default quotas for different kinds of computing resources [32, 34].

Much existing work ignores these limits. However, using resources exceeding the limits requires users to manually request limit increases from platform providers. The increases are not granted immediately and may take hours or days to become effective. This inconvenience would be especially significant when the computations are small and can finish in minutes or hours, in which cases the interactions with platform providers may take more time than the computations themselves.

Therefore, we consider these platform limits. We restrict the maximum number of VMs that a schedule can use concurrently to be L . For Amazon EC2, the default value of L is 20 [31]. We choose this simplest limit form to facilitate discussions; however, our framework is also applicable to other limit forms.

3.2. Workflow Application Model

A workflow is modeled by a Directed Acyclic Graph (DAG) $W = (T, E)$, where $T = \{t_1, t_2, \dots, t_n\}$ is the set of tasks and $E \subseteq T \times T$ is the set of dependencies between the tasks. For each task $t \in T$, we denote its *multi-resource demands* by a k -tuple $D_t = (r_t^1, r_t^2, \dots, r_t^k)$ and its execution time by RT_t . It is assumed that the tasks are not moldable on any

type of VMs, and their resource usages and execution times can be estimated using some existing performance prediction techniques [35, 36, 37]. If there is a dependency $(t_i, t_j) \in E$, t_i is called a *predecessor* of t_j , and t_j is called a *successor* of t_i . The set of all predecessors and the set of all successors of task t are denoted by $pred(t)$ and $succ(t)$, respectively. A task is called an *entry task* if it has no predecessor and called an *exit task* if it has no successor. The set of all entry tasks and the set of all exit tasks are denoted by T_{entry} and T_{exit} , respectively.

Many existing algorithms assume an idealized communication model where the intermediate data are transferred directly between services without consideration of communication contentions. On the IaaS platforms, if the last task on a VM wants to send data directly to a task whose hosting VM has not been launched, either the former should be kept on for longer time or the latter should be boot up earlier to make the communication possible. Since the VMs are charged for their running times, this may potentially increase the cost of workflow execution.

In practice, instead of using the above peer-to-peer communication model, workflow management systems (e.g., Pegasus [38] and Swift [39]) usually store the intermediate data in centralized storage that all tasks can directly access (e.g., using distributed file systems or remote object stores). In this case, the data accessing time can be included in task execution times. Moreover, with centralized storage, the intermediate data generated will not be lost in the case of VM failures [40, 41]. Thus, in this paper, we assume the centralized storage scheme, by which the data transfer times between tasks do not need to be formulated separately.

3.3. Multiprogrammed Uses of VMs

In this paper, we allow a single VM to be multiprogrammed and shared by more than one tasks when there are sufficient resources. A large number of heuristics and meta-heuristics which schedule workflow tasks in the non-multiprogrammed way have been proposed. Since the IaaS platforms allow users to choose the types of VMs to acquire, one may argue that the multiprogrammed use of VMs is unnecessary even if the tasks are not moldable. An alternative approach is to use small VMs and still allocate a single task per VM using existing algorithms. However, this approach has some disadvantages compared with allowing VMs to be multiprogrammed.

First, small VMs usually have fewer capacities in all types of resources than large VMs. For example, the single-core VM instances on Amazon EC2 can be equipped with at most 2GB memory, and their network performance is officially described as “low to moderate” [42]. Thus, one may have to use a larger, multi-core VM for a single-thread computation which requires large memory or high-throughput networking. In this case, if no other task is allowed to be executed simultaneously on the VM, the idle CPU time of the other cores is wasted. Moreover, some advanced techniques, like *Enhanced Networking* and *Intel AVX instruction set extensions*, are only available on large VMs [42], making them more suitable for scientific computing.

Second, as aforementioned, many IaaS platforms set limits on the maximum number of VMs that a customer can simultaneously use. On these platforms, the multiprogrammed use of

large VMs would archive much higher concurrency for workflow execution than the non-multiprogrammed use of small VMs, thereby reducing the makespan and execution cost. This shall be demonstrated by our experimental results in Section 6.2.

3.4. The Scheduling Model

Schedule: Workflow scheduling maps tasks to services, as illustrated in Fig. 1. On the IaaS platforms, the services are VMs dynamically acquired and released by customers. In other words, the scheduling algorithm needs to decide the number and types of VMs to use for the workflow execution. Therefore, a valid schedule shall contain at least two mappings:

$$PL : T \rightarrow V, \quad (2)$$

$$TYP : V \rightarrow P, \quad (3)$$

where V is the set of VMs used, PL_t indicates where a task t is placed, and TYP_v determines the type of VM v .

Many scheduling algorithms tend to start tasks as early as possible. However, in our problem, sometimes a lazier strategy can achieve better schedules. For example, in Fig. 1b, postponing the task t_6 for 10 min reduces the running time of v_1 , which may consequently cut the total cost of workflow execution. Hence, we introduce an additional mapping

$$ST : T \rightarrow \mathbb{R}^+ \quad (4)$$

to indicate the start times of tasks. Finally, we can define the solution of our scheduling problem as $\mathbb{S} = (PL, TYP, ST)$.

To facilitate discussions, we denote the finish time of a task t by FT_t , where $FT_t = ST_t + RT_t$. To reduce the monetary cost, a VM will not be launched until the first task assigned to it arrives, and it will be terminated as soon as the last task on it finishes. Therefore, we define the *running interval* of a VM v as

$$VT_v = [VT_v^-, VT_v^+) = \left[\min_{t \in T_v} ST_t, \max_{t \in T_v} FT_t \right), \quad (5)$$

where T_v is the set of tasks on v .

Constraints: A set of constraints must be satisfied for a valid solution. To simplify presentation, for two k -tuples $R_i = (r_i^1, r_i^2, \dots, r_i^k)$ and $R_j = (r_j^1, r_j^2, \dots, r_j^k)$, we define:

$$R_i \preceq R_j := r_i^1 \leq r_j^1 \wedge r_i^2 \leq r_j^2 \wedge \dots \wedge r_i^k \leq r_j^k, \quad (6)$$

$$R_i + R_j := (r_i^1 + r_j^1, r_i^2 + r_j^2, \dots, r_i^k + r_j^k). \quad (7)$$

First, to respect task dependencies, there must be:

$$\forall (t_i, t_j) \in E : FT_{t_i} \leq ST_{t_j}. \quad (8)$$

Second, at any moment, the total resource consumption in a VM must not exceed its resource capacity:

$$\forall v \in V, \tau \in VT_v : \sum_{t \in T_v \wedge \tau \in [ST_t, FT_t)} D_t \preceq C_{TYP_v}. \quad (9)$$

¹For simplicity of illustration, we plot only one type of resources in this and later figures in this paper.

Figure 1: An example DAG and a schedule of it.¹

Finally, at any moment, the number of running VMs must not exceed the platform limit:

$$\forall \tau \in \mathbb{R}^+ : |\{v \in V \mid \tau \in VT_v\}| \leq L. \quad (10)$$

Objective: Various optimization criteria such as makespan and execution cost can be used for workflow scheduling. In this paper, we focus on deadline-constrained cost-optimization workflow scheduling, which aims to minimize the total monetary cost of workflow execution within a specified deadline. Formally, the objective can be written as

$$\text{minimizing} \quad \sum_{v \in V} \text{Cost}(TYP_v, |VT_v|), \quad (11)$$

$$\text{subject to} \quad \max_{t \in T} FT_t - \min_{t \in T} ST_t \leq DL, \quad (12)$$

where DL is the given deadline.

4. List Scheduling with Multi-resource Packing

A list-scheduling algorithm assigns the tasks in a workflow to best-fit services in the order of a priority list. In this section, we propose a multiprogrammed list-scheduling framework to schedule tasks in a workflow to the VMs dynamically acquired from IaaS platforms according to their multi-resource demands. By employing different task prioritization and placement comparison methods, this framework can be applied to deal with different scheduling criteria.

4.1. Multi-Resource Packing

For a p -type VM v with resource capacity C_p , a task t with resource demand D_t and execution time RT_t can be placed on it at start time st if and only if

$$\forall \tau \in [st, st + RT_t) : U(v, \tau) + D_t \preceq C_p, \quad (13)$$

where $U(v, \tau)$ denotes the usage of different types of resources in v at time τ .

Algorithm 1 List-scheduling framework.

```

1: Prioritize and sort tasks to a scheduling list
2:  $plt \leftarrow$  an empty set representing the platform
3: for each task  $t$  in the scheduling list do
4:    $EST_t \leftarrow \max_{t' \in pred(t)} FT_{t'}$ 
5:    $\rho_{bst} \leftarrow None$ 
6:   CalcExtendableIntervalsForVMs( $plt$ )
7:   for each VM  $v$  in  $plt$  do
8:      $est \leftarrow \max\{EI_v^-, EST_t\}$ 
9:     for each  $p \in P_v^e$  do  $\triangleright P_v^e$  is defined in (17)
10:       $st \leftarrow \text{EarliestSlot}(v, C_p, D_t, RT_t, est)$ 
11:      if  $st + RT_t \leq EI_v^+$  then
12:        if  $\rho(t, v, p, st)$  is better than  $\rho_{bst}$  then
13:           $\rho_{bst} \leftarrow \rho(t, v, p, st)$ 
14:       $st \leftarrow \text{EarliestSlot}(plt, L, 1, RT_t, EST_t)$ 
15:      for each  $p \in P_v^n$  do  $\triangleright P_v^n$  is defined in (18)
16:        if  $\rho(t, None, p, st)$  is better than  $\rho_{bst}$  then
17:           $\rho_{bst} \leftarrow \rho(t, None, p, st)$ 
18:      Place  $t$  according to  $\rho_{bst}$ 
19:      if  $\rho_{bst}$  is on a new VM then
20:        Add the new VM to  $plt$ 

```

To deal with the platform limit, we can consider an IaaS platform has the capacity L and each VMs on it has the size 1. Denoting $U(plt, \tau)$ as the number of VMs running at time τ on the platform plt , a new VM can be acquired at time st and released at time $(st + l)$ if and only if

$$\forall \tau \in [st, st + l) : U(plt, \tau) + 1 \leq L. \quad (14)$$

As discussed in Section 3, there are different forms of resource limits on the IaaS platforms. The above abstraction also works for the other limit forms. For example, if a platform has specific quantity limits L_{p_a} on p_a -type VMs and L_{p_b} on p_b -type VMs, we can define the platform size as (L, L_{p_a}, L_{p_b}) and consider the sizes of p_a -type VMs, p_b -type VMs and other VMs as $(1, 1, 0)$, $(1, 0, 1)$ and $(1, 0, 0)$, respectively. Also, if a platform sets quotas for different types of resources, we can use a tuple containing these quotas as the platform size and consider VMs having sizes of their resource capacities.

4.2. The List-scheduling Framework

Algorithm 1 gives the multiprogrammed list-scheduling framework on IaaS platforms. The framework first assigns priorities to the tasks in the workflow and sorts them into a scheduling list (Line 1). For different scheduling objectives, different prioritization methods can be employed.

In the order of the sorted scheduling list, a list-scheduling algorithm then places each task to its best-fit service in a set of candidate services. For traditional cluster or grid environments, this set is known and composed of fixed servers. However, on the IaaS platform, the set of VMs used in the final schedule needs to be constructed by the scheduling algorithm. Thus, in Algorithm 1, we first initialize an empty set to represent the initial platform (Line 2). During the scheduling, a task can be

Figure 2: The extendable intervals of two VMs on a platform with a limit $L = 2$.

placed on the VM that has already hosted tasks, or it can be assigned to a new VM. If it is placed on a new VM, this VM will then be added onto the platform.

For each task, the framework first compares its placements on different existing VMs (Lines 6–13), and then, it considers launching a new VM for the task (Lines 14–17). To facilitate discussions, we use $\rho(t, v, p, st)$ to denote a placement of task t on p -type VM v , starting execution at time st . If v is *None*, the placement is on a new VM. Before evaluating task placements, the framework calculates EST_t for t (Line 4), which is the earliest possible start time of t due to the task dependencies.

Placements on existing VMs: The framework first calculates the *Extendable Interval* for all the existing VMs on the platform (Line 6). The extendable interval $EI_v = [EI_v^-, EI_v^+)$ of a VM v is defined as the longest time interval satisfying

$$[VT_v^-, VT_v^+) \subseteq [EI_v^-, EI_v^+) \subseteq [0, +\infty) \quad (15)$$

and

$$\forall \tau \in [EI_v^-, VT_v^-) \cup [VT_v^+, EI_v^+) : U(plt, \tau) + 1 \leq L, \quad (16)$$

which restricts the running interval of v under the platform limit and thus the start time and the finish time of a new task that can be placed on v . If the task starts before EI_v^- or finishes after EI_v^+ , the platform limit will be violated. Fig. 2 illustrates the extendable intervals of some VMs on a platform with a limit $L = 2$.

After that, to search for a good placement, the framework scales each existing VM v to different VM types in a set P_v^e and compares the placement of putting t on v with these types (Lines 9–13). Specifically, P_v^e is the set of p that satisfies

$$R_v^{peak} \preceq C_p \wedge D_t \preceq C_p \wedge \nexists p' : R_v^{peak} + D_t \preceq C_{p'} \preceq C_p, \quad (17)$$

where R_v^{peak} is the peak usage of different types of resources in v over its running interval.

The rationale of conditions (17) is as follows. First, to make the placement of t possible, the available resource capacities of v must not be less than D_t . Second, v cannot be scaled down to smaller than R_v^{peak} ; otherwise, the existing placements on v would be affected. Finally, if a VM type $p' \neq p$ satisfying $R_v^{peak} + D_t \preceq C_{p'} \preceq C_p$ exists, since the resource utilization of v will never go beyond $R_v^{peak} + D_t$ after the placement, p' must be a better choice than p in this case.

Algorithm 2 Compressing a VM.

```

1: procedure COMPRESSVM( $v$ )
2:   Sort  $T_v$  by descending order of  $FT_t$ 
3:   for each task  $t$  in the sorted  $T_v$  do
4:      $EI_t \leftarrow \text{ExtendableIntervalForTask}(v, t)$ 
5:      $ft \leftarrow \min \left\{ EI_t^+, \min_{t' \in \text{succ}(t)} ST_{t'} \right\}$ 
6:     if  $ft > FT_t$  then
7:       Postpone  $t$  to start at  $ft - RT_t$ 
8:   Sort  $T_v$  by ascending order of  $ST_t$ 
9:   for each task  $t$  in the sorted  $T_v$  do
10:     $EI_t \leftarrow \text{ExtendableIntervalForTask}(v, t)$ 
11:     $st \leftarrow \max \left\{ EI_t^-, \max_{t' \in \text{pred}(t)} FT_{t'} \right\}$ 
12:    if  $st < ST_t$  then
13:      Advance  $t$  to start at  $st$ 

```

Given a VM v of a type p , the algorithm looks for the first time slot of length RT_t inside time interval $[\max \{EI_v^-, EST_t\}, EI_v^+)$ that meets the multi-resource demands of the task (condition (13)) (Lines 8, 10–11). In Line 10, EarliestSlot returns the beginning time of the slot found. The placement of putting the task at the beginning of this slot will be compared to the best-known placement ever found (Lines 12–13).

Placement on a new VM: The framework also evaluates placing the task on a new VM. Regardless of the VM type, a newly acquired VM will have a running time of RT_t if task t is placed on it. Thus, in order not to violate the platform limit, the algorithm first finds the earliest feasible slot for acquiring an RT_t -length VM on the platform (condition (14)) after t 's earliest possible start time EST_t (Line 14). The beginning time of the slot found will be the start time of t in its placement on the new VM.

Then, the algorithm decides the best type of the new VM. It traverses all the possible VM types and compares the placements of the task on the new VM with these types (Lines 15–17). Similarly, to avoid resource under/over-provisioning, the VM types to examine is restricted to a set p_v^n of p satisfying

$$D_t \preceq C_p \wedge \nexists p' : D_t \preceq C_{p'} \preceq C_p. \quad (18)$$

Finally, the task is appointed the best placement found on both existing and new VMs (Line 18). If it is on a new VM, this VM is added to the platform (Line 19–20).

4.3. Reducing Resource Fragmentation

Resource fragmentation can be caused by unaligned task placements. For example, in Fig. 1b, the task t_6 can be started later at time 40 min to reduce the running interval (and thus the cost) of the VM v_1 without compromising the makespan of workflow execution. Thus, we propose an adjustment procedure (Algorithm 2) to further improve the schedule produced, which “compresses” the VMs being used.

Algorithm 2 can be performed on each VM on the platform after Algorithm 1. First, it postpones all the tasks on a VM v

(i.e., T_v) to finish as late as possible within v 's running interval VT_v in the reverse order of their finish times (Lines 2–7). We define the *Extendable Interval* $EI_t = [EI_t^-, EI_t^+)$ of a task t on v as the longest time interval satisfying

$$[ST_t, FT_t) \subseteq [EI_t^-, EI_t^+) \subseteq [VT_v^-, VT_v^+) \quad (19)$$

and

$$\forall \tau \in [EI_t^-, ST_t) \cup [FT_t, EI_t^+) : U(v, \tau) + D_t \leq C_{\text{TYPE}_v}, \quad (20)$$

which is computed by `ExtendableIntervalForTask`(v, t) (Line 4). EI_t is the interval where task t can be moved without violating the resource capacities of v . The compression procedure moves t only in this interval. On the other hand, t must still finish before the start of all its successors after the movement. Hence, the task t can be postponed to finish at $\min \left\{ EI_t^+, \min_{t' \in \text{succ}(t)} ST_{t'} \right\}$ (Lines 5–7) at the latest. After all the tasks on v are postponed, the running interval VT_v might be shortened due to the possible delay of VT_v^- .

After that, the tasks on v are moved backward again to start as early as possible within the updated VT_v in the order of their current start times (Lines 8–13). Likewise, the movement of a task t can only be conducted inside its current extendable interval EI_t and after $\max_{t' \in \text{pred}(t)} FT_{t'}$ (Lines 10–11). These movements might shorten the running interval VT_v again because of the potential update of VT_v^+ .

Fig. 3 illustrates an example of compressing a VM which hosts five tasks. We assume there is no dependency among these tasks. At first, these tasks are placed from time 0 to 80 min, which makes the VM have a total running time of 80 min (Fig. 3a). The procedure first postpones the tasks in the order of t_5, t_4, t_3, t_2, t_1 . As a result, the acquisition of the VM is delayed to time 15 min (Fig. 3b). Then, the procedure moves each task backward again in the order of t_1, t_2, t_3, t_4, t_5 , which advances the VM's release to time 75 min (Fig. 3c). Eventually, the running time of the VM is decreased to 60 min, and the resource fragmentation is substantially reduced.

4.4. Complexity

We assume the number of resource types k is a small constant so that all operations on k -tuples (e.g., $R_i \preceq R_j$) take $O(1)$ times. Also, we assume comparing two placements of a task can be done in $O(1)$ time. Besides, we consider prioritizing tasks having a typical complexity of $O(n^2)$ and sorting tasks by the priorities has the complexity of $O(n \log n)$, where n is the number of tasks.

Due to space limitations, we do not discuss the implementation details of the platform and VMs. With proper data structure to represent the platform and the VMs, the procedures `CalcExtendableIntervalsForVMs` (Line 6 in Algorithm 1), `ExtendableIntervalForTask` (Lines 4, 10 in Algorithm 2) and `EarliestSlot` (Lines 10, 14 in Algorithm 1) all can run in linear times with respect to the number of VMs or tasks.

Consider the procedure of placing the i -th task t in the scheduling list. First, calculating EST_t for t takes $O(i)$ time. Then,

Figure 3: An example of VM compression.

the framework finds the best task placements on existing VMs. At this moment, there are at most $(i - 1)$ VMs on the platform. Thus, `CalcExtendableIntervalsForVMs` takes at most $O(i)$ time. On each VM v , `EarliestSlot` is invoked at most m times, where m is the number of VM types. Let n_v be the number of tasks already placed on v , there is $\sum_{v \in V} n_v = i - 1$. Therefore, the complexity of finding the best placement on existing VMs for the i -th task is

$$O(i) + \sum_{v \in V} (m \cdot O(n_v)) = O(i \cdot m). \quad (21)$$

The framework also tries to place the task on a new VM. Since there are at most $(i - 1)$ VMs used at this moment, the invocation of `EarliestSlot` to determine the acquisition time of the new VM takes $O(i)$ time. Then, deciding the best type for the new VM takes at most $O(m)$ time. Therefore, this part of the framework has a total complexity of $O(i + m)$.

Thus, the overall complexity of placing the i -th task is

$$O(i) + O(i \cdot m) + O(i + m) = O(i \cdot m). \quad (22)$$

`CompressVM` is performed on each VM v after scheduling. In this procedure, sorting all tasks on v has an $O(n_v \log n_v)$ complexity, and searching each task's extendable interval and moving the task both take at most $O(n_v)$ times. Thus, the complexity of compressing a VM v is

$$O(n_v \log n_v) + n_v \cdot O(n_v) = O(n_v^2). \quad (23)$$

In summary, the overall time complexity of the proposed list-scheduling framework is

$$O(n^2) + \sum_{i=1}^n O(i \cdot m) + \sum_{v \in V} O(n_v^2) = O(m \cdot n^2). \quad (24)$$

5. Deadline-constrained Workflow Scheduling

In this section, we develop a scheduling algorithm to optimize the cost of workflow execution while meeting a given

Algorithm 3 Assigning latest start times to tasks.

```

1: Calculate  $rank_d$  for all tasks in  $T$ 
2: Sort  $T$  in descending order by the  $rank_d$  values
3:  $c^* \leftarrow$  the capacity of the largest VM type in  $P$ 
4:  $V \leftarrow$  a set of  $L$  largest VMs
5: for each task  $t$  in the sorted  $T$  do
6:    $LST_t \leftarrow 0, v_{bst} \leftarrow None$ 
7:    $lft \leftarrow \begin{cases} \min_{t' \in succ(t)} LST_{t'}, & t \notin T_{exit} \\ DL, & t \in T_{exit} \end{cases}$ 
8:   for each VM  $v \in V$  do
9:      $ft \leftarrow \text{LatestSlot}(v, c^*, D_t, RT_t, lft)$ 
10:    if  $LST_t < ft - RT_t$  then
11:       $LST_t \leftarrow ft - RT_t, v_{bst} \leftarrow v$ 
12:    Insert  $t$  into  $v_{bst}$  (starting at  $LST_t$ )
```

execution deadline based on the framework proposed in Section 4.

The main idea of this algorithm is as follows. First, the algorithm estimates a *latest start time* for each task in the workflow. The difference between the given workflow deadline and the latest start time of a task indicates the minimal amount of time needed for executing this task and all its successive tasks with consideration of the platform limit. The tasks are then prioritized and sorted in the order their latest start times. After that, the algorithm appoints tasks the placements which can meet their latest start times and incur minimal *cost increases*.

5.1. Latest Start Times

Given a workflow, Algorithm 3 determines the latest start times (denoted by LST_t for task t) for all tasks assuming that the largest VMs are used.

First, the tasks are sorted in descending order of their *downward ranks* (Lines 1–2), which are defined as

$$rank_d(t) = \begin{cases} RT_t + \max_{t' \in pred(t)} rank_d(t'), & t \notin T_{entry}, \\ RT_t, & t \in T_{entry}. \end{cases} \quad (25)$$

The rank estimates the minimum amount of time needed from the start of the workflow execution to the finish of each task.

Then, the algorithm constructs a set containing L VMs of the largest type in P (with the capacity of c^*) to host the tasks (Lines 3–4). Let DL denote the deadline of workflow execution. Obviously, when placing the tasks, an exit task must finish before DL , and a non-exit task t must finish before time $\min_{t' \in succ(t)} LST_{t'}$. Thus, given a task t , the algorithm searches for the latest time slot on each VM that ends before the time lft (calculated in Line 7) and can accommodate the task (Lines 8–11). In Line 9, `LatestSlot` finds the latest available time slot for a task on a VM before time lft and returns the end time of the slot found. Eventually, t is placed in the slot which has the latest end time among all the VMs, and its start time is taken as LST_t (Line 12).

Note that, if the platform limit is not less than the number of tasks ($L \geq n$), a task t can always be placed on a new VM. Thus, its latest start times can be simply derived by

$$LST_t = \begin{cases} \min_{t' \in succ(t)} LST_{t'} - RT_t, & t \notin T_{exit}, \\ DL - RT_t, & t \in T_{exit} \end{cases} \quad (26)$$

without actually performing the above procedure.

5.2. Task Prioritization

Our algorithm uses the value LST_t as the priority of a task t . The tasks with lower LST_t will be scheduled earlier.

Many existing list heuristics follow the task prioritization approach in HEFT [3], where an upward rank is assigned to each task in the workflow. Essentially, the upward rank of a task is the minimum time of executing the task and all its successive tasks when there are sufficient computing resources. It can be easily shown that, when $L \geq n$, our prioritization method is equivalent to using the upward ranks as priorities. However, when $L < n$, our priority LST_t can provide a more precise estimation of the minimal time needed for executing t and its successors. Thus, prioritizing tasks with these values can lead to a better scheduling result.

5.3. Placement Comparison

To optimize the total monetary cost of the workflow execution, we hope each task placement can incur minimal cost increase to the current cost of the schedule. The cost increase caused by a placement of a task t on a VM v is defined as the difference between v 's running costs after and before the placement:

$$\Delta Cost = Cost(p', |VT_v \cup [ST_t, FT_t]|) - Cost(p, |VT_v|), \quad (27)$$

where ST_t and FT_t are the start time and finish time of t in the placement, respectively, and p and p' are the VM type of v before and after the placement, respectively. If the placement is on a new VM, $\Delta Cost$ is simply $Cost(p', RT_t)$.

Two placements of a task t are compared as follows:

1. If both placements start before LST_t , the one with lower $\Delta Cost$ is considered better. If both of them have the same $\Delta Cost$ value, the earlier placement is preferred.
2. Otherwise, the earlier placement is always considered better. In this case, reserving more time for the remaining tasks might improve the scheduling success rate to meet the execution deadline.

5.4. The Complete Algorithm

The complete process of the deadline-constrained workflow scheduling algorithm (named as DyDL) is given as follows.

1. Determine the latest start times of tasks by Algorithm 3.
2. Apply the list-scheduling framework in Algorithm 1 with the task prioritization method in Section 5.2 and the placement comparison method in Section 5.3.

Figure 4: A sample workflow

Table 1: The execution times and resource demands of the tasks in Fig. 4

	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8
RT_t (min)	15	20	5	20	25	25	5	5
CPU cores	1	1	2	1	2	2	1	2

3. Compress each VM used by Algorithm 2.

Given a workflow containing n tasks, determining the latest start times for tasks takes $O(n^2)$ time. Sorting the tasks by their priorities takes $O(n \log n)$ time. Comparing two placements of a task takes $O(1)$ time. All these procedures do not affect the overall complexity of the list-scheduling framework. Thus, the complexity of DyDL is $O(m \cdot n^2)$.

5.5. An Illustrative Example

In this section, we show a detailed example of how the proposed algorithms work. The structure of the sample workflow is given in Fig. 4. This workflow consists of eight tasks, whose execution times and resource demands are given in Table 1. For simplicity of illustration, we consider only one type of computing resources, the CPU cores. We assume there are two available VM types, p_1 and p_2 , on the IaaS platform, which are equipped with one and two CPU cores, respectively. The charging units of the VM types are both set to one hour, and we assume the prices of p_1 and p_2 are \$0.05/hr and \$0.1/hr, respectively. The deadline for the workflow execution is set to 100 minutes.

The first step of the proposed algorithm is to calculate the latest start times (LST values) for tasks using Algorithm 3 (Section 5.1). The $rank_d$ values of the tasks are first calculated recursively by formula (25), and the results are listed in Table 2. Then, eight p_2 -type (the larger type) VMs are prepared and the tasks are placed onto these VMs backward from the deadline in descending order of their $rank_d$ values, that is, in the order of $t_8, t_6, t_7, t_5, t_4, t_3, t_2, t_1$. The resultant task placements are shown in Fig. 5, which eventually make uses of three VMs. Then, according to these placements, the LST values of the tasks are assigned, which are shown in Table. 2.

Having calculated the latest start times, the algorithm prioritizes the tasks as discussed in Section 5.2, where the tasks

Table 2: The values of $rank_d$ and LST of the tasks in Fig. 4

	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8
$rank_d(t)$	15	20	25	35	45	65	50	70
LST_t	35	45	65	50	65	70	90	95

Figure 5: The task placements generated by Algorithm 3 for the workflow in Fig. 4.

Table 3: The values of $\Delta Cost$ (in \$) and ST (in minutes) on different VMs and with different VM types for each task in Fig. 4.

		v_1		v_2		v_3		decision
		p_1	p_2	p_1	p_2	p_1	p_2	
t_1	$\Delta Cost$	0.05	-	-	-	-	-	launch v_1 with type p_1 assign t_1 to v_1
	ST	0	-	-	-	-	-	
t_2	$\Delta Cost$	0	-	0.05	-	-	-	assign v_2 to v_1
	ST	15	-	0	-	-	-	
t_4	$\Delta Cost$	0	-	0.05	-	-	-	assign v_4 to v_1
	ST	35	-	15	-	-	-	
t_3	$\Delta Cost$	$\times$	0.05	$\times$	0.1	-	-	scale v_1 to p_2 -type assign v_3 to v_1
	ST	$\times$	55	$\times$	35	-	-	
t_5	$\Delta Cost$	$\times$	0.1	$\times$	0.1	-	-	launch v_2 with type p_2 assign v_5 to v_2
	ST	$\times$	60	$\times$	35	-	-	
t_6	$\Delta Cost$	$\times$	0.1	$\times$	0	$\times$	0.1	assign t_6 to v_2
	ST	$\times$	60	$\times$	60	$\times$	60	
t_7	$\Delta Cost$	$\times$	0.1	$\times$	0	$\times$	0.1	assign t_7 to v_2
	ST	$\times$	60	$\times$	85	$\times$	60	
t_8	$\Delta Cost$	$\times$	0.1	$\times$	0	$\times$	0.1	assign v_8 to v_2
	ST	$\times$	90	$\times$	90	$\times$	90	

with higher LST values get the lower priorities. Therefore, the final scheduling order of the tasks is $t_1, t_2, t_4, t_3, t_5, t_6, t_7, t_8$. Following this order, the tasks are then placed onto the best VMs decided by the procedure shown in Algorithm 1. For each task, different placements on different VMs or with different VM types are compared according to their corresponding $\Delta Cost$ and ST values, as discussed in Section 5.3. Table 3 lists the $\Delta Cost$ and ST values of different placements for every task. The $-$ marks in the cells mean that the VMs have not been provisioned or the VM types are too large to be considered by the current task, while the $\times$ marks indicate that the VM can no longer be scaled down to such types as they have already been of larger types. The detailed procedure is explained as follows.

At the beginning, since there has been no VM on the platform yet, a new p_1 -type VM v_1 is provisioned for the first task t_1 . As shown in the table, the monetary cost incurred is \$0.05,

Figure 6: Task placements generated by DyDL for the workflow in Fig. 4.

even though t_1 actually uses the VM for only 15 minutes. After that, since there is still charged-but-not-used fragmentation on this VM, assigning t_2 to v_1 does not cause any additional cost. If so, it has to wait for 15 more minutes until start, compared to being put on a new VM; however, the ST_{t_2} value of 15 min is still earlier than the LST_{t_2} value we calculated above (45 min, Table 2). Thus, according to the comparison rules discussed in Section 5.3, the algorithm will eventually assign t_2 still onto the VM v_1 . Likewise, the next task, t_4 , is placed onto v_1 as well, without increasing the execution cost.

Now, the algorithm decides the placement for task t_3 . At this moment, the VM v_1 is of the type p_1 . This type can provide only one CPU core, but t_3 has the demand of two CPU cores. Thus, the choices for t_3 can be either scaling v_1 to p_2 type or launching a new p_2 -type VM v_2 . The trade-offs between $\Delta Cost$ and ST values for these two placements are given in the table. Since $ST_{t_3} = 55$ min can still meet task t_3 's latest start time (65 min), the algorithm decides to scale the VM v_1 up and assigns t_3 to it again.

After task t_3 's execution finishes, the first charging unit of VM v_1 is exhausted. Therefore, whether placing task t_5 still on v_1 or launching a new VM for it causes the same cost increase of \$0.1. However, executing t_5 on the new VM make it starts earlier at the time 35 min. Hence, according to the second rule listed in Section 5.3, the v_2 is provisioned for it with the type of p_2 , as task t_5 has the CPU demand of two cores. At last, analogously, tasks t_6, t_7 and t_8 are all assigned to VM v_2 as well, which causes additional workflow's execution cost and at the same time can still fulfill their latest start times.

Fig. 6a shows the final task placements on the VMs. The schedule has the overall execution makespan of 95 minutes, which is within the deadline of 100 minutes, and the total ex-

Table 4: Time Complexities of the Compared Algorithms

	Greedy	IC-PCP	PSO	CGA2	DyDL
$T(n)$	$O(n^2)$	$O(mn^2)$	$O(kgn^2)$	$O(kgn^2)$	$O(mn^2)$

ecution cost of \$0.2. The `CompressVM` function is then performed on each used VM, and the result is shown in Fig. 6b. As we can see from the figure, the resource fragmentation on VM v_1 has been massively reduced by the compression procedure. As a result, the overall makespan of workflow execution is further reduced to 80 minutes.

6. Experiments

6.1. Experiment Setup

6.1.1. Algorithms Compared

In this section, we experimentally compare our DyDL algorithm with several existing algorithms including Greedy, IC-PCP [29], PSO [5] and CGA² [7]. The Greedy algorithm uses a simple strategy which always schedules a task onto the cheapest VM that can meet its latest start time. IC-PCP is a well-known deadline-constrained workflow scheduling algorithm for IaaS clouds. Although originally proposed for non-multiprogrammed resources and moldable tasks, it can be applied to our problem with some modifications. Specifically, we check the multi-resource demands of all tasks on a partial critical path (PCP) before assigning them to a VM, and we allow a VM to be scaled up when the platform limit is reached. PSO and CGA² are two meta-heuristics, whose search operators can be applied to our problem without any modification.

The time complexities of the compared algorithms are given in Table 4. The Greedy algorithm has the complexity of $O(n^2)$. The original IC-PCP algorithm also takes $O(n^2)$ time; however, when applying to the multiprogrammed task scheduling problem and allowing VMs to be scaled to different VM types, its complexity is increased to $O(mn^2)$. This is the same as our proposed DyDL algorithm. On the other hand, as meta-heuristics, PSO and CGA² both have significantly higher complexity of $O(kgn^2)$, where k and g are the population size of particles/individuals and the number iterations/generations, respectively.

To verify the benefits of the multiprogrammed use of VMs for workflow execution, we also compare the schedules produced by the proposed algorithm with non-multiprogrammed schedules. The non-multiprogrammed schedules are produced by Greedy and IC-PCP as well; however, in these schedules, the VMs are used in a dedicated manner. That is, a VM can execute only one task at a time, even if there are unused computing resources. We use Greedy/D and IC-PCP/D to denote the non-multiprogrammed versions of Greedy and IC-PCP, respectively.

6.1.2. Workflows and IaaS Characteristics

We include both real-world workflow applications and randomly generated DAGs in the experiments. The real-world

Figure 7: Sample structures of the real-world workflows tested.

workflows include Montage, Epigenomics, LIGO Inspiral Analysis, CyberShake and SIPHT [9, 11], which are widely used for evaluating workflow scheduling algorithms. Fig. 7 shows the sample DAG structures of these applications. We consider two types of computing resources: CPU and Memory. The task resource demands in these real-world workflows are set according to [11]. The random DAGs are generated by a task graph generator called DAGGEN [43] with the parameters $n \in \{30, 90, 200\}$, $jump \in \{1, 3, 5\}$, $fat \in \{0.2, 0.4, 0.6, 0.8\}$, $regularity \in \{0.2, 0.4, 0.6, 0.8\}$ and $density \in \{0.2, 0.4, 0.6, 0.8\}$, where n is the number of tasks in the DAG, $jump$ is the maximum number of levels spanned by task dependencies, fat indicates the maximum number of tasks that can be executed concurrently, $regularity$ defines the regularity of task distribution between different levels of the DAG, and $density$ determines the number of dependencies between the tasks of two consecutive DAG levels. The parameter values to generate each DAG are randomly chosen from the above sets, and the resource demands of the tasks in these DAGs vary from $\langle 1 \text{ CPU}, 100 \text{ MB Mem} \rangle$ to $\langle 8 \text{ CPU}, 9 \text{ GB Mem} \rangle$. The total number of the DAGs generated and tested is 2000.

In the experiments, we simulate the scheduling of the workflows on Amazon EC2's c4-type VMs. The configurations and prices of the VM types used are listed in Table 5.

Table 5: VM types used in experiments.

Type	vCPU	Memory (GB)	Price
c4.large	2	3.75	\$0.1 per hour
c4.xlarge	4	7.5	\$0.199 per hour
c4.2xlarge	8	15	\$0.398 per hour
c4.4xlarge	16	30	\$0.796 per hour
c4.8xlarge	36	60	\$1.591 per hour

6.1.3. Experiment Parameters

Given a workflow, we introduce a factor $k_{dl} \in [0, 1]$ to define its execution deadline:

$$DL = Span_{HEFT} + k_{dl} \times \left(\sum_{t \in T} RT_t - Span_{HEFT} \right), \quad (28)$$

where $Span_{HEFT}$ is the makespan achieved by the HEFT algorithm [3] and $\sum_{t \in T} RT_t$ is the total execution time of all tasks.

The HEFT algorithm aims to minimize the makespan of workflow execution under a given amount of resources. In our case, when computing $Span_{HEFT}$, we assume there are L VMs of c4.8xlarge type (the largest type) available where L is the platform limit. A smaller k_{dl} value indicates a tighter deadline. We evaluate the performance of the proposed algorithms in the cases where $k_{dl} = 0.1, 0.2, 0.3$ and 0.5 . The costs of workflow execution are affected by the values of the charging units and the platform limits. Thus, we conduct experiments with different CU and L values, which are chosen from $\{1s, 60s, 3600s\}$ and $\{5, 20, 100, \infty\}$, respectively.

6.2. Experimental Results

6.2.1. Real-world Workflows

Table 6 shows the execution costs produced by different algorithms on the real-world workflows when $CU = 3600s$ and $L = 20$ (the defaults on Amazon EC2). As we can see, DyDL produces the cheapest schedules in most cases. For example, for the large-size Montage workflow (1000 tasks), the schedules produced by DyDL can save up to 94% cost compared with the schedules produced by the other algorithms. For middle-size Inspiral workflow (100 tasks), although failing to beat IC-PCP under the tightest deadline ($k_{dl} = 0.1$), DyDL produces substantially cheaper schedules than IC-PCP under more relaxed deadlines. For large-size Epigenomics and Inspiral workflows (1000 tasks), the Greedy strategy is unexpectedly effective, which achieves slightly less cost (2%–5%) than DyDL. However, this simple strategy does not always work well, even for the same application. For example, for Epigenomics, when the workflow size is reduced to 100 tasks, the schedules produced by Greedy are the most expensive among all the algorithms compared. On the other hand, PSO and CGA² perform the worst among the multiprogrammed algorithms in most cases, which indicates that the search operators designed in these meta-heuristics are not suitable for our problem, although they can be directly applied without any modification.

The table also shows that DyDL produces significantly better schedules than all the non-multiprogrammed algorithms in

all cases. It can be seen that IC-PCP/D fails to find valid schedules meeting deadlines in more than one-third cases. As we have mentioned in Section 3.3, the non-multiprogrammed use of VMs may suffer from the platform limit on concurrently used VMs. Recalls that IC-PCP and IC-PCP/D are critical-path-based algorithms which always schedule all tasks on a PCP onto a single VM. That is, when no new VM can be acquired, with the multiprogrammed resource model, IC-PCP can still place a PCP in parallel to another PCP on an existing VM, but IC-PCP/D cannot. The failures of IC-PCP/D suggest that the multiprogrammed use of VMs may be especially useful when the number of concurrently used VMs is limited. The table also shows that Greedy/D performs considerably worse than Greedy on large-size workflows, which may be due to the increased resource fragmentation. However, for most small/middle-size workflows, the schedules produced by Greedy/D are noticeably better than those produced by its multiprogrammed cousin. This suggests that, without proper packing strategies, multiprogrammed sharing of VMs among tasks may not always archive better scheduling results than non-multiprogrammed use of VMs.

To evaluate the impacts of different charging units and platform limits, for each application under each CU (or L) value, we compare the average cost produced by each algorithm over different k_{dl} , L (or CU) and workflow sizes. Before taking the average, the costs produced are first normalized by

$$Cost_{norm} = \frac{Cost - Cost_{min}}{Cost_{max} - Cost_{min}}, \quad (29)$$

where $Cost_{max}$ and $Cost_{min}$ are the maximum and minimum costs achieved among all the seven algorithms compared, respectively.

Fig. 8a compares the average costs produced by different algorithms under different charging units. It can be seen that DyDL has stable performance with various charging units, and it significantly outperforms all the other algorithms compared under all the charging units tested. For SIPHT workflows, when $CU = 3600s$, IC-PCP and the two non-multiprogrammed algorithms archive worse but close performance compared with DyDL. However, when the charge units are smaller, their performance drops rapidly. Fig. 8b presents the results for different platform limits². It is shown that, when the computing resources are relatively abundant (L is large), Greedy can achieve comparable performance to DyDL, while DyDL still produces significantly less average cost than all the other algorithms in all cases. However, as we have seen in Table 6, the performance of Greedy is not stable—when L is not large, the average costs produced by Greedy are substantially higher than those produced by both IC-PCP and DyDL in most cases. IC-PCP performs better than the other algorithms for SIPHT and Epigenomics workflows, but it is still notably worse than DyDL.

The success rates archived by different algorithms to meet the given deadlines on the set of real-world workflows are summarized in Table 7. For almost all the parameter settings and

²For CyberShake, Inspiral and Montage workflows under the platform limit $L = 5$, the missing bars in the figures indicate that IC-PCP/D cannot find schedules meeting the deadlines in these cases.

(a) With different CU settings.

(b) With different L settings.

Figure 8: Average normalized costs produced by different algorithms for real-world workflows under different CU and L values.

Figure 9: Cumulative distributions of normalized costs produced by different algorithms on random DAGs ($CU = 3600s$).

Table 6: Costs achieved by different algorithms for real-world workflows when $CU = 3600s$ and $L = 20$ (in \$).

Applications		CyberShake				Epigenomics				Inspirial				Montage				SIPHT			
Number of tasks		30	50	100	1000	24	46	100	997	30	50	100	1000	25	50	100	1000	30	60	100	1000
$k_{dl} = 0.1$	Greedy	0.50	1.00	1.99	1.99	1.00	2.99	17.94	63.78	0.50	1.00	1.99	3.99	0.50	0.50	1.99	1.99	3.99	3.99	3.99	3.99
	IC-PCP	0.50	0.62	1.12	8.59	0.50	1.36	7.94	84.68	0.50	0.74	0.87	4.98	0.37	0.50	1.12	11.96	0.37	0.62	0.62	3.84
	PSO	0.50	1.00	1.99	11.96	1.00	2.49	16.31	75.73	0.50	1.00	1.99	12.96	0.50	0.50	1.99	10.46	0.25	1.12	3.99	35.87
	CGA ²	0.50	1.00	1.99	11.96	1.00	2.12	14.20	72.00	0.50	1.00	1.99	12.96	0.50	0.50	1.99	10.46	2.12	3.99	5.98	35.87
	Greedy/D	0.50	0.62	1.12	1.24	0.87	1.86	14.26	134.04	0.74	1.36	2.36	8.43	0.74	1.86	2.48	2.48	0.37	0.50	0.74	6.32
	IC-PCP/D	0.99	1.36	2.23	<i>failed</i>	0.87	2.23	<i>failed</i>	<i>failed</i>	0.87	1.36	1.24	<i>failed</i>	1.12	<i>failed</i>	<i>failed</i>	<i>failed</i>	0.50	0.87	0.99	6.70
	DyDL	0.25	0.37	0.50	0.62	0.50	0.99	7.32	67.58	0.37	0.50	0.99	4.34	0.25	0.37	0.50	0.99	0.25	0.37	0.50	3.35
$k_{dl} = 0.2$	Greedy	0.50	1.00	1.99	1.99	1.00	2.99	17.94	63.78	0.50	1.00	1.99	3.99	0.50	0.50	1.99	1.99	3.99	3.99	3.99	3.99
	IC-PCP	0.50	0.62	1.12	8.59	0.50	1.36	7.94	84.68	0.50	0.74	0.87	4.98	0.37	0.50	1.12	11.96	0.37	0.62	0.50	3.84
	PSO	0.50	1.00	1.99	11.96	1.00	0.87	17.94	75.73	0.50	1.00	1.99	12.96	0.50	0.50	1.99	10.46	1.24	3.99	5.98	35.87
	CGA ²	0.50	1.00	1.99	11.96	1.00	2.12	14.20	72.00	0.50	1.00	1.99	12.96	0.50	0.50	1.99	10.46	2.11	8.34	5.98	35.87
	Greedy/D	0.50	0.62	0.62	0.99	0.74	1.86	14.26	133.30	0.62	1.24	1.36	8.43	0.74	0.99	1.98	2.48	0.37	0.50	0.74	6.32
	IC-PCP/D	0.99	1.36	2.23	<i>failed</i>	0.87	2.23	<i>failed</i>	<i>failed</i>	0.87	1.36	1.24	<i>failed</i>	1.12	<i>failed</i>	<i>failed</i>	<i>failed</i>	0.50	0.87	0.87	6.70
	DyDL	0.25	0.25	0.25	0.50	0.50	0.87	7.32	67.21	0.37	0.62	0.87	4.34	0.12	0.12	0.12	0.12	0.25	0.37	0.37	3.35
$k_{dl} = 0.3$	Greedy	0.50	1.00	1.99	1.99	1.00	2.99	17.94	63.78	0.50	1.00	1.99	3.99	0.50	0.50	1.99	1.99	3.99	3.99	3.99	3.99
	IC-PCP	0.50	0.62	1.12	8.59	0.50	1.36	7.94	84.68	0.50	0.74	0.87	4.98	0.37	0.50	1.12	11.96	0.37	0.62	0.74	3.84
	PSO	0.50	1.00	1.99	11.96	1.00	2.99	17.94	75.73	0.50	1.00	1.99	12.96	0.50	0.50	1.99	10.46	2.24	3.99	5.98	35.87
	CGA ²	0.50	1.00	1.99	11.96	1.00	2.12	14.20	72.00	0.50	1.00	1.99	12.96	0.50	0.50	1.99	10.46	2.24	3.99	5.98	35.87
	Greedy/D	0.37	0.37	0.50	0.99	0.74	1.74	14.01	133.42	0.50	1.24	1.49	8.43	0.62	0.99	1.98	2.60	0.37	0.50	0.74	6.32
	IC-PCP/D	0.99	1.36	2.23	<i>failed</i>	0.87	2.23	<i>failed</i>	<i>failed</i>	0.87	1.36	1.24	<i>failed</i>	1.12	<i>failed</i>	<i>failed</i>	<i>failed</i>	0.50	0.87	0.87	7.94
	DyDL	0.25	0.25	0.25	0.50	0.50	0.87	7.19	67.08	0.25	0.50	0.50	4.34	0.12	0.12	0.12	0.12	0.25	0.37	0.37	3.35
$k_{dl} = 0.5$	Greedy	0.50	1.00	1.99	1.99	1.00	2.99	17.94	63.78	0.50	1.00	1.99	3.99	0.50	0.50	1.99	1.99	3.99	3.99	3.99	3.99
	IC-PCP	0.50	0.62	1.12	8.59	0.50	1.36	7.94	84.68	0.50	0.74	0.87	4.98	0.37	0.50	1.12	11.96	0.37	0.62	0.74	3.84
	PSO	0.50	1.00	1.00	11.96	1.00	2.99	15.95	75.73	0.50	1.00	1.99	12.96	0.50	0.50	1.99	10.46	2.24	3.99	5.98	35.87
	CGA ²	0.50	1.00	1.99	11.96	1.00	2.37	14.20	72.25	0.50	1.00	1.99	12.96	0.50	0.50	1.99	10.46	1.74	3.99	5.98	35.87
	Greedy/D	0.25	0.25	0.25	0.99	0.74	1.61	14.01	133.67	0.74	0.62	0.74	8.43	0.62	0.99	1.98	0.62	0.37	0.50	0.74	6.32
	IC-PCP/D	0.99	1.36	2.23	<i>failed</i>	0.87	2.23	<i>failed</i>	<i>failed</i>	0.87	1.36	1.24	<i>failed</i>	1.12	<i>failed</i>	<i>failed</i>	<i>failed</i>	0.50	0.62	1.12	7.94
	DyDL	0.12	0.12	0.12	0.50	0.50	0.87	7.19	66.84	0.12	0.25	0.50	4.34	0.12	0.12	0.12	0.12	0.25	0.37	0.37	3.35

Table 7: Success rates of different algorithms to meet given deadlines on real-world workflows.

By k_{dl}	$k_{dl} = 0.1$	$k_{dl} = 0.2$	$k_{dl} = 0.3$	$k_{dl} = 0.5$
Greedy	100.00%	100.00%	100.00%	100.00%
IC-PCP	98.75%	98.75%	98.75%	98.75%
PSO	100.00%	100.00%	100.00%	100.00%
CGA ²	100.00%	100.00%	100.00%	100.00%
Greedy/D	98.75%	99.17%	100.00%	100.00%
IC-PCP/D	67.08%	67.50%	67.92%	68.33%
DyDL	100.00%	100.00%	100.00%	100.00%
By L	$L = 5$	$L = 20$	$L = 100$	$L = \infty$
Greedy	100.00%	100.00%	100.00%	100.00%
IC-PCP	95.00%	100.00%	100.00%	100.00%
PSO	100.00%	100.00%	100.00%	100.00%
CGA ²	100.00%	100.00%	100.00%	100.00%
Greedy/D	97.92%	100.00%	100.00%	100.00%
IC-PCP/D	20.83%	65.00%	85.00%	100.00%
DyDL	100.00%	100.00%	100.00%	100.00%

real-world workflows tested, the schedules produced by different multiprogrammed algorithms can all meet the given deadlines of workflow execution. The only exception is IC-PCP which fails to find valid schedules in some cases of the 1000-task Montage workflow with a platform limit $L = 5$. On the other hand, if the VMs can only be used in the non-multiprogrammed mode, Greedy/D fails to produce valid schedules in some cases under strict platform limit ($L = 5$) and tight deadlines ($k_{dl} = 0.1$ or 0.2). In comparison, IC-PCP/D fails in many more cases. Its success rate drops dramatically with the decrease of L value. In particular, when the platform limit L is set to 5, it can find valid schedules to meet the deadlines in about only one-fifth cases.

Finally, we remark that our proposed DyDL algorithm can complete scheduling within 4 seconds for all the cases of real-world workflows, similar to the Greedy and IC-PCP algorithms.

6.2.2. Random DAGs

Fig. 9 presents the cumulative distributions of the costs produced by different algorithms for the random DAGs. Due to the diversity of the DAG structures generated, the costs produced are also normalized by (29). It is shown that PSO and CGA² perform the worst among all the multiprogrammed algorithms compared in all cases, and they even fail to beat the non-multiprogrammed Greedy/D and IC-PCP/D algorithms when there is no platform limit on concurrently used VMs or when the deadlines are loose. This suggests again that existing meta-heuristics do not work well for our scheduling problem.

Note that a normalized cost of 0 indicates the algorithm per-

Table 8: Success rates of different algorithms to meet given deadlines on random DAGs ($CU = 3600s$).

		$k_{dl} = 0.1$	$k_{dl} = 0.2$	$k_{dl} = 0.3$	$k_{dl} = 0.5$
$L = 20$	Greedy	100.00%	100.00%	100.00%	100.00%
	IC-PCP	98.60%	98.50%	98.50%	98.50%
	PSO	100.00%	100.00%	100.00%	100.00%
	CGA ²	99.85%	99.80%	99.70%	100.00%
	Greedy/D	100.00%	99.95%	100.00%	100.00%
	IC-PCP/D	96.20%	94.90%	94.75%	94.60%
	DyDL	100.00%	100.00%	100.00%	100.00%
$L = \infty$	Greedy	100.00%	100.00%	100.00%	100.00%
	IC-PCP	98.60%	98.50%	98.50%	98.50%
	PSO	100.00%	100.00%	100.00%	100.00%
	CGA ²	99.85%	99.80%	99.70%	100.00%
	IC-PCP/D	96.20%	94.90%	94.75%	94.60%
	Greedy/D	100.00%	100.00%	100.00%	100.00%
	DyDL	100.00%	100.00%	100.00%	100.00%

forms best among all those tested and a normalized cost of 1 indicates the algorithm performs the worst. In general, the performance of Greedy and IC-PCP is similar. When the platform limit is $L = 20$, Greedy achieves the lowest costs for more DAGs than IC-PCP under the tightest deadline ($k_{dl} = 0.1$), but IC-PCP works better under more relaxed deadlines. When there is no platform limit ($L = \infty$), Greedy performs slightly better than IC-PCP. On the other hand, Greedy/D and IC-PCP/D perform notably worse than Greedy, IC-PCP and our DyDL algorithm, which demonstrates again the advantages of multiprogrammed use of VMs for workflow execution. Finally, irrespective of whether there is a platform limit or not, DyDL remarkably outperforms all the other algorithms under all the k_{dl} settings. In particular, under the looser deadlines ($k_{dl} = 0.3$ and 0.5), DyDL produces the cheapest schedules for almost all the random DAGs, and under the tightest deadline ($k_{dl} = 0.1$), it can still perform the best in around 75% cases.

Table 8 shows the success rates of different algorithms to meet the given deadlines for the random DAGs. It can be seen that CGA² and Greedy/D occasionally experience failures when the platform limit L is set at 20, while IC-PCP and IC-PCP/D fails in relatively more cases under every deadline constraint, no matter whether there is a platform limit or not. In contrast, the schedules produced by our DyDL algorithm can always meet the deadlines in all cases.

7. Conclusions

In this paper, we have presented a workflow scheduling model which considers both the multiprogrammed use of computing resources on heterogeneous IaaS platforms and the multi-resource demands of tasks. A generic list-scheduling framework for the new model is proposed. To make the best use of cloud resources, this framework can efficiently pack tasks onto VMs and pack the VMs to the platform. Also, to fully exploit the heterogeneity of clouds, the framework allows the VMs to be dynamically scaled up during the scheduling process. Based

on this framework, a deadline-constrained workflow scheduling algorithm which optimizes the monetary cost of workflow execution is proposed. The experimental results have shown that it can produce significantly better schedules than several existing heuristics and meta-heuristics.

Acknowledgments

This work is supported by Singapore Ministry of Education Academic Research Fund Tier 2 under Grant MOE2013-T2-2-067, and Academic Research Fund Tier 1 under Grant 2017-T1-002-024.

References

- [1] M. Wiecek, A. Hoheisel, R. Prodan, Towards a general model of the multi-criteria workflow scheduling on the grid, *Future Generation Computer Systems* 25 (3) (2009) 237–256 (2009).
- [2] S. Smach, K. Viriyapant, Taxonomies of workflow scheduling problem and techniques in the cloud, *Future Generation Computer Systems* 52 (2015) 1–12 (2015).
- [3] H. Topcuoglu, S. Hariri, M.-Y. Wu, Performance-effective and low-complexity task scheduling for heterogeneous computing, *IEEE Transactions on Parallel and Distributed Systems* 13 (3) (2002) 260–274 (2002).
- [4] H. Arabnejad, J. G. Barbosa, List scheduling algorithm for heterogeneous systems by an optimistic cost table, *IEEE Transactions on Parallel and Distributed Systems* 25 (3) (2014) 682–694 (2014).
- [5] M. A. Rodriguez, R. Buyya, Deadline based resource provisioning and scheduling algorithm for scientific workflows on clouds, *IEEE Transactions on Cloud Computing* 2 (2) (2014) 222–235 (2014).
- [6] J. J. Durillo, R. Prodan, Workflow scheduling in amazon ec2, *Proceedings of the Workshop on Large Scale Distributed Virtual Environments on Clouds and P2P* 17 (2013) 374–383 (2013).
- [7] L. Liu, M. Zhang, R. Buyya, Q. Fan, Deadline-constrained coevolutionary genetic algorithm for scientific workflow scheduling in cloud computing, *Concurrency and Computation: Practice and Experience* 29 (5) (2017) e3942 (2017).
- [8] J. Sahni, D. Vidyarthi, A cost-effective deadline-constrained dynamic scheduling algorithm for scientific workflows in a cloud environment, *IEEE Transactions on Cloud Computing* (2015).
- [9] S. Bharathi, A. Chervenak, E. Deelman, G. Mehta, M. H. Su, K. Vahi, Characterization of scientific workflows, in: *Proceedings of the Workshop on Workflows in Support of Large-Scale Science*, 2008, pp. 1–10 (2008).
- [10] L. Ramakrishnan, D. Gannon, A survey of distributed workflow characteristics and resource requirements, *Tech. rep.*, Indiana University (2008).
- [11] G. Juve, A. Chervenak, E. Deelman, S. Bharathi, G. Mehta, K. Vahi, Characterizing and profiling scientific workflows, *Future Generation Computer Systems* 29 (3) (2013) 682–692 (2013).
- [12] Y. C. Lee, H. Han, A. Y. Zomaya, On resource efficiency of workflow schedules, *Procedia Computer Science* 29 (2014) 534–545 (2014).
- [13] Y. C. Lee, A. Y. Zomaya, Stretch out and compact: Workflow scheduling with resource abundance, in: *Proceedings of the 13th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, 2013, pp. 219–226 (2013).
- [14] Y. C. Lee, H. Han, A. Y. Zomaya, M. Yousif, Resource-efficient workflow scheduling in clouds, *Knowledge-Based Systems* 80 (2015) 153–162 (2015).
- [15] R. Grandl, S. Kandula, S. Rao, A. Akella, J. Kulkarni, Graphene: Packing and dependency-aware scheduling for data-parallel clusters, in: *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation*, 2016, pp. 81–97 (2016).
- [16] A. Ghodsi, M. Zaharia, B. Hindman, A. Konwinski, S. Shenker, I. Stoica, Dominant resource fairness: Fair allocation of multiple resource types, in: *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, Vol. 167, 2011, pp. 24–24 (2011).
- [17] W. Wang, B. Liang, B. Li, Multi-resource fair allocation in heterogeneous cloud computing systems, *IEEE Transactions on Parallel and Distributed Systems* 26 (10) (2015) 2822–2835 (2015).

- [18] M. Chowdhury, Z. Liu, A. Ghodsi, I. Stoica, Hug: Multi-resource fairness for correlated and elastic demands, in: Proceedings of the 13th USENIX Symposium on Networked Systems Design and Implementation, 2016, pp. 407–424 (2016).
- [19] W. Wang, B. Li, B. Liang, J. Li, Multi-resource fair sharing for datacenter jobs with placement constraints, in: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, 2016, pp. 1–17 (2016).
- [20] R. Grandl, G. Ananthanarayanan, S. Kandula, S. Rao, A. Akella, Multi-resource packing for cluster schedulers, SIGCOMM Computer Communication Review 44 (4) (2014) 455–466 (2014).
- [21] D. Huang, P. Du, C. Zhu, H. Zhang, X. Liu, Multi-resource packing for job scheduling in virtual machine based cloud environment, in: Proceedings of the 2015 IEEE Symposium on Service-Oriented System Engineering, 2015, pp. 216–221 (2015).
- [22] M. Sheikhalishahi, R. M. Wallace, L. Grandinetti, J. L. Vazquez-Poletti, F. Guerriero, A multi-dimensional job scheduling, Future Generation Computer Systems 54 (2016) 123–131 (2016).
- [23] G. Xu, C. Z. Xu, S. Jiang, Prophet: Scheduling executors with time-varying resource demands on data-parallel computation frameworks, in: Proceedings of the 2016 IEEE International Conference on Autonomic Computing, IEEE, 2016, pp. 45–54 (2016).
- [24] R. Grandl, M. Chowdhury, A. Akella, G. Ananthanarayanan, Altruistic Scheduling in Multi-resource Clusters, in: Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation, 2016, pp. 65–80 (2016).
- [25] R. Sakellariou, H. Zhao, E. Tsiakkouri, M. D. Dikaiakos, Scheduling workflows with budget constraints, in: Proceedings of the Integrated Research in GRID Computing CoreGRID Integration Workshop, 2007, pp. 189–202 (2007).
- [26] H. Arabnejad, J. G. Barbosa, A budget constrained scheduling algorithm for workflow applications, Journal of Grid Computing 12 (4) (2014) 665–679 (2014).
- [27] Z. Zhu, G. Zhang, M. Li, X. Liu, Evolutionary multi-objective workflow scheduling in cloud, IEEE Transactions on Parallel and Distributed Systems 27 (5) (2016) 1344–1357 (2016).
- [28] S. Abrishami, M. Naghibzadeh, D. H. J. Epema, Cost-driven scheduling of grid workflows using partial critical paths, IEEE Transactions on Parallel and Distributed Systems 23 (8) (2012) 1400–1414 (2012).
- [29] S. Abrishami, M. Naghibzadeh, D. H. J. Epema, Deadline-constrained workflow scheduling algorithms for infrastructure as a service clouds, Future Generation Computer Systems 29 (1) (2013) 158–169 (2013).
- [30] A. Ghodsi, V. Sekar, M. Zaharia, I. Stoica, Multi-resource fair queueing for packet processing, SIGCOMM Computer Communication Review 42 (4) (2012) 1–12 (2012).
- [31] Amazon, Amazon ec2 faqs, <https://goo.gl/iB5IUg>.
- [32] Microsoft, Azure subscription limits and quotas, <https://goo.gl/Vxi9TD>.
- [33] Rackspace, Quotas - rackspace developer portal, <https://goo.gl/pND86w>.
- [34] Google, Google cloud platform - compute engine documentation - resource quotas, <https://goo.gl/XSYuPx>.
- [35] R. F. da Silva, G. Juve, M. Rynge, E. Deelman, M. Livny, Online task resource consumption prediction for scientific workflows, Parallel Processing Letters 25 (03) (2015) 1541003 (2015).
- [36] A. M. Chirkin, A. S. Belloum, S. V. Kovalchuk, M. X. Makkes, M. A. Melnik, A. A. Visseratin, D. A. Nasonov, Execution time estimation for workflow scheduling, Future Generation Computer Systems 75 (2017) 376–387 (2017).
- [37] T. P. Pham, J. J. Durillo, T. Fahringer, Predicting workflow task execution time in the cloud using a two-stage machine learning approach, IEEE Transactions on Cloud Computing (99) (2017) 1–1 (2017).
- [38] E. Deelman, K. Vahi, G. Juve, M. Rynge, S. Callaghan, P. J. Maechling, R. Mayani, W. Chen, R. F. da Silva, M. Livny, K. Wenger, Pegasus, a workflow management system for science automation, Future Generation Computer Systems 46 (2015) 17–35 (2015).
- [39] M. Wilde, M. Hategan, J. M. Wozniak, B. Clifford, D. S. Katz, I. Foster, Swift: A language for distributed parallel scripting, Parallel Computing 37 (9) (2011) 633–652 (2011).
- [40] E. Vairavanathan, S. Al-Kiswani, L. B. Costa, Z. Zhang, D. S. Katz, M. Wilde, M. Ripeanu, A workflow-aware storage system: An opportunity study, in: 2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (ccgrid 2012), 2012, pp. 326–334 (2012).
- [41] V. Singh, I. Gupta, P. K. Jana, A novel cost-efficient approach for deadline-constrained workflow scheduling by dynamic provisioning of resources, Future Generation Computer Systems 79 (2018) 95–110 (2018).
- [42] Amazon, Ec2 instance types, <https://goo.gl/DU2D7J>.
- [43] F. Suter, Daggen: A synthetic task graph generator, <https://github.com/frs69wq/daggen>.