A Cost-Effective Scheduling Algorithm for Scientific Workflows in Clouds

Mengxia (Michelle) Zhu
Dept of Computer Science
Southern Illinois University
Carbondale, IL 62901
Email: mzhu@cs.siu.edu

Qishi Wu
Dept of Computer Science
University of Memphis
Memphis, TN 38152
Email: qishiwu@memphis.edu

Yang Zhao
Dept of Computer Science
Southern Illinois University
Carbondale, IL 62901
Email: yzhao@cs.siu.edu

***Abstract*—Cloud computing enables the delivery of computing, software, storage, and data access through web browsers as a metered service. In addition to commercial applications, an increasing number of large-scale workflow-based scientific applications are being supported by cloud computing. In order to meet the rapidly growing and dynamic computing demands of scientific users, the cloud service provider needs to employ efficient and cost-effective job schedulers to guarantee workflow completion time as well as improve resource utilization for high throughput. Based on rigorous cost models, we formulate a delay-constrained optimization problem to maximize resource utilization and propose a two-step workflow scheduling algorithm to minimize the cloud overhead within a user-specified execution time bound. The extensive simulation results illustrate that our approach consistently achieves lower computing overhead or higher resource utilization than existing methods within the execution time bound. Our approach also significantly reduces the total execution time by strategically selecting appropriate mapping nodes for prioritized modules.**

Keywords: Scientific workflow; workflow scheduling; cloud computing

I. INTRODUCTION

Cloud architecture acts as a resource sharing pool that provides services to various clients through the Internet. The cloud computing resources can be allocated, released, and reallocated to meet the needs of on-demand and backfill users [2]. There are three types of cloud services, namely, Infrastructure-as-a-Service (IAAS), Platform-as-a-Service (PAAS) and Software-as-a-Service (SAAS). In IAAS, virtual machines (VMs) residing on a physical node are created with full user control, as exemplified by Amazon's ES2's instants [11]. Similarly, PAAS provides users with an instance whose execution environment has been configured for users to run applications in the specific development environment using particular programming paradigms such as Java and Python, as exemplified by Google's App Engine [12]. SAAS aims to serve a wide range of common users to run some particular software remotely, so there is no need to install it on the user's local machine. One typical example is Salesforce's Database.com, which provides many application programming interfaces (APIs) for users to access the database as though it is a local database.

We focus on the requirements of scientific users whose needs are usually satisfied by IAAS clouds. In a typical IAAS cloud, upon a user's request, the service node calls the system monitor to determine where and what instances of VMs should be deployed, and then the pricing model decides the cost charged to the user. Once determined, user VM images are sent to the physical node to boot up VMs, on which, the user configures and runs computing modules.

Many scientific applications are modeled as a workflow, which can be as simple as a single module or as complex as a Directed Acyclic Graph (DAG). The dependency and parallelism embedded in a workflow requires that the modules be dispatched to a group of distributed VMs in order to maximize the execution efficiency. Although there have been many research efforts on the mapping and scheduling of workflow systems in clouds, our work differs from existing ones in that the available cloud resources are considered to be time-dependent as opposed to static upon the arrival of a user request. In other words, the resource availability of any physical computer node or network link is not a constant but varies over time. We believe that this consideration better reflects the cloud environments in the real world shared by a large group of users, where any user can make reservations in advance or consume VM resources during the scheduling process.

Our scheduling strategy considers two objectives from the perspective of a cloud service provider. First of all, a user's Quality of Service (QoS) requirement, which can be specified as the latest completion time of the entire workflow, must be satisfied. Secondly, the service throughput defined as the total number of served user requests during a certain time period should be maximized. The resource utilization rate, defined as the useful computing cost over the total cost including the overhead on starting up and shutting down VMs as well as the idle VM time, should be minimized. Although most cloud service providers charge users by hours regardless of the time spent on computing or overhead, it is always of the provider's interest to reduce any unnecessary computing cycles spent on overhead since these wasted resources could be allocated elsewhere to meet other users' job requests. This is particularly important for those heavily loaded clouds during the peak time. A scheduling algorithm that does not take resource utilization into careful consideration may lead to an early resource saturation and job request turndown.

This cloud scheduling problem has been proven to be NP-complete [8]. We propose a two-step heuristic workflow mapping approach, referred to as Cost-Effective Virtual Machine Allocation Algorithm within Execution Time Bound (CEVAET). In the first step, we perform topological sorting to divide modules into different layers, which determine the mapping order of the modules starting from the first layer. We then assign each module with a certain priority value based on its computational load and map it to the node that yields the lowest partial end-to-end delay (EED) from the starting module to the current module. This module mapping process is repeated to reduce the total EED until a convergence point is reached. In the second step, we improve the resource utilization rate by reducing the overhead of VM's startup and shutdown time as well as the idle time. The modules may be mapped in such a way that some of them may share the same VM for reduced startup and shutdown overhead, or some VMs may be released early to save the idle time until the next active module arrives.

The rest of the paper is organized as follows. In Section II, we conduct a survey of workflow mapping algorithms. In Section III, we construct the models for scientific workflows and cloud environments to compute the computation cost of the workflow. In Section IV, we prove that the EED and cost are two conflicting objectives, and optimizing both at the same time is not possible. We then formulate the problem to minimize the cost within a given execution time bound. We present the details of the algorithm in Section V and evaluate its performance in Section VI. We conclude our work in Section VII.

II. Related Work

We conduct a survey of existing workflow mapping algorithms in various network environments with focus on those developed specifically for clouds.

The mapping of DAG-structured workflows into underlying distributed computing environments with heterogeneous resources for minimal EED has been studied for years and is well known to be NP-hard [8]. Several heuristic algorithms [1], [15], [16], [8], [17], [22] have been proposed in the literature. In [17], the Heterogeneous Earliest Finish Time (HEFT) heuristic algorithm tries the mapping of each module onto all the nodes first, and then chooses the best one with the earliest finish time. $Streamline$ [8], a workflow scheduling algorithm for streaming data, creates a coarse-grained dataflow graph on available grid resources. In [16], an optimal algorithm is proposed to determine an optimal static allocation of modules among a large number of sensors based on an A^* algorithm. The Recursive Critical Path (RCP) algorithm utilizes a dynamic programming strategy and recursively finds a critical path to minimize the EED [15]. This algorithm is used as the mapping scheme for the Scientific Workflow Automation and Management Platform (SWAMP) [18], a Condor/DAGMan-based workflow system that enables scientists to conveniently assemble, execute, monitor, control, and steer computing workflows in distributed environments via a unified web-based user interface. Condor is a specialized workload management system for compute-intensive jobs [19] and it can be used to serve grid environments such as Globus grid [21]. The Directed Acyclic Graph Manager (DAGMan) is a meta-scheduler for Condor jobs that manages dependencies between jobs at a higher level than the Condor Scheduler. DAGMan submits jobs to Condor in an order represented by a DAG and processes the results [20]. RCP provides a better mapping performance than the mapping scheme currently employed by the Condor Scheduler [18]. However, the aforementioned algorithms either target static homogeneous environments with fully connected networks, or fail to find feasible mapping solutions when the system scales up, or adopts a simple greedy algorithm that oftentimes leads to unsatisfactory performance.

A number of research efforts have been devoted to scheduling workflows in cloud environments [4], [5], [6], [7], [3]. In [4], Hoffa *et al.* compared the performance and the overhead of running a workflow in a local machine, a local cluster, and a virtual cluster. Haizea is a lease management architecture [6], which implements leases as VMs, leverages the ability to suspend, migrate, and resume computations, and provides the leased resources in a customized application environment. In [7], Vockler *et al.* discussed the experience of running workflows and evaluating their performance as well as the challenges in different cloud environments. In [5], Figueiredo *et al.* made an effort to create a grid-like environment from cloud resources to ensure a higher level of security and flexible resource control. In [3], Yu *et al.* proposed a cost-based workflow scheduling algorithm that minimizes execution cost while meeting the deadline for completing the tasks.

There are several commonly used job scheduling policies including Greedy (First Fit) and Round Robin algorithms in open-source cloud computing management systems such as Eucalyptus [9]. Queuing system, advanced reservation and preemption scheduling are adopted by OpenNebula [10]. Nimbus uses some customizable tools such as PBS and SGE [13]. The Greedy and Round Robin are heuristic approaches that select adaptive physical resources for the VM to deploy without considering the maximum usage of the physical resource. The queuing system, advanced reservation and preemption scheduling do not consider any balanced overall system utilization, either. Pegasus Workflow Manage System is a more advanced workflow scheduling algorithm [14], which maps a workflow onto the cloud to generate an executable workflow using a clustering approach to group short-duration modules as a single module in order to reduce data transfer overhead and the number of VMs created. The rank matching algorithm in [23] features a scheduling strategy that ranks each module's possible mapping nodes and selects the node with the lowest cost as the mapping result.

In this paper, we aim to achieve the maximum utilization of cloud computing resources while guaranteeing the QoS required by individual users. Many different techniques have been proposed to meet these two objectives separately, but the research efforts in tackling both problems at the same time are still very limited.

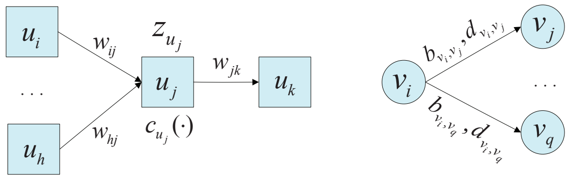

Fig. 1. Models of workflow graph (left) and cloud network (right).

III. Analytical Models

We construct the analytical cost models for the workflow task graph and the underlying cloud computer network graph to facilitate a mathematical formulation of the constrained mapping optimization problem.

A. Workflow and Cloud Models

The left side of Fig. 1 shows the workflow of a distributed computing application constructed as a directed acyclic graph $G_{wf} = (V_{wf}, E_{wf})$, $|V_{wf}| = N$, where the vertices represent computing modules $V_{wf} = \{u_1, u_2, ..., u_N\}$: u_1 and u_N denote the starting and ending modules, respectively. The weight w_{ij} on edge $e_{i,j}$ represents the size of data transferred from module u_i to module u_j. The dependency between a pair of modules is represented as a directed edge. Module u_j receives a data input w_{ij} from each of its preceding modules u_i and performs a predefined computing routine whose complexity is modelled as a function $c_{u_j}(\cdot)$ of the total aggregated input data size z_{u_j}. However, in real scenarios, the complexity of a module is an abstract quantity, which not only depends on the computational complexity of its own procedure but also on the implementation details realized in its algorithm. If module u_j is executed on a particular node, the data output w_{jk} is sent to each of its succeeding modules u_k upon the completion of execution. A module cannot start its execution until all input data required by this module arrive. To generalize our model, if an application has multiple starting or ending modules, we can create a virtual starting or ending module of complexity zero and connect it to all starting or ending modules without any data transfer along the edges.

The right side of Fig. 1 shows the cloud environment where virtual machines are reserved, deployed and executed on physical computer nodes. We consider a general cloud environment where both prior VM reservations and on-demand requests are supported. Thus, our resource allocation status for the cloud network is time dependent, which means that the available computing resources on each node and the bandwidth on each link vary over time. We model the underlying cloud network as an arbitrary directed network graph $G_{cn} = (V_{cn}, E_{cn})$ with $|V_{cn}| = K$, consisting of a set of computing nodes $V_{cn} = \{v_1, v_2, ..., v_K\}$ as well as directed edges between them. Node v_j is featured by its normalized computing power p_{v_j} based on CPU and memory. The communication link $L_{i,j}$ between node v_i to v_j is featured by bandwidth b_{v_i,v_j}, and minimum link delay d_{v_i,v_j}.

Fig. 2 shows several allocable resource graphs for a cloud network at different time points due to resource allocation.

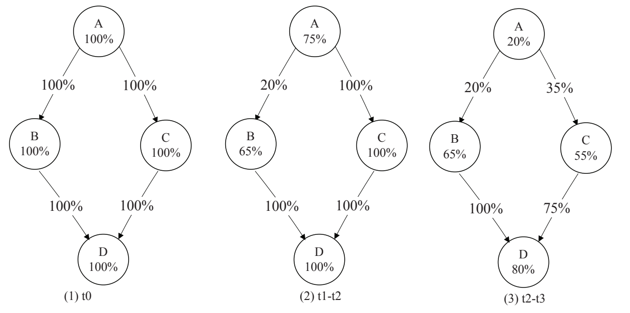

Fig. 2. Allocable resource graph for a cloud network from time point t_0 to t_3.

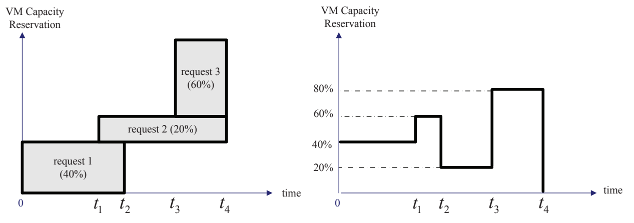

Fig. 3. Reserved requests on a single cloud node from time point t_0 to t_4.

Fig. 3 illustrates an example of three reservation requests made on one cloud node during different time slots. For example, request 1 reserves 40% of the node's general capacity from t_0 to t_2; request 2 reserves 20% from t_1 to t_4; request 3 reserves 60% from t_3 to t_4. The maximal available computing power of this node from t_0 to t_4 is $p_{v_j,t_0,t_4} = \min(60\%, 40\%, 80\%, 20\%)$. The largest VM instance that can be allocated on v_j from time t_1 to t_n, namely $p^{VM}_{v_j,t_1,t_n}$, is computed as the maximum VM instance that can be launched using p_{v_j,t_1,t_n}. The execution time of module u_i on node v_j during time slot t_1 and t_n is then computed as $t_{v_j,t_1}(u_i) = \frac{z_{u_i} * c_{u_j}(\cdot)}{p^{VM}_{v_j,t_1,t_n}}$, where z_{u_i} denotes the aggregated and complexity normalized input data size on module u_i. Similarly, the maximum link bandwidth along L_{v_i,v_j} during time slot t_m and t_n is $min(B_{v_i,v_j,t_m,t_n})$.

B. Workflow Execution Cost

The cost of running a workflow in a cloud is measured by the sum of the total time, during which virtual machines are running including idle and overhead time, multiplied by the corresponding VM's capacity. The time spent on deploying VM on a particular node v_j consists of the following components: 1) The startup time for the virtual machine includes selecting a virtual node and transferring a virtual image as well as the boot-up time, and is assumed to be a fixed value of t_{start}. 2) The running time for every assigned module on that VM. Suppose that a set U of modules are assigned on $VM_{v_j,k}$, and start to run from time t_s and end at time t_e in a sequential manner. The running time for these modules is computed as $\frac{\sum_{u_i \in U} z_{u_i} \cdot c_{u_i}(\cdot)}{p_{VM_{v_j,k}}}$. 3) The idle time between the execution time of any two modules. When two modules run on the same VM, there could be some idle time after one module

TABLE I
NOTATIONS USED IN THE ANALYTICAL MODELS.

Parameters	Definitions
$G_{wf} = (V_{wf}, E_{wf})$	the computation workflow
N	the number of modules in the workflow
u_i	the i-th computing module
$e_{i,j}$	the dependency edge from module u_i to u_j
w_{ij}	the data size transferred over dependency edge $e_{i,j}$
z_{u_j}	the aggregated input data size of module u_j
$c_{u_j}(\cdot)$	the computational complexity of module u_i
st_i	the start time of module u_i
et_i	the end time of module u_i
$G_{cn} = (V_{cn}, E_{cn})$	the cloud network
K	the total number of nodes in the cloud
v_j	the j-th computer node
v_s	the source node
v_d	the destination node
p_{v_j}	the total computing power of node v_j
$p^{VM}_{v_j,t_1,t_n}$	the maximal percentage of computing power of VM on node v_j from t_1 to t_n
$L_{i,j}$	the network link between nodes v_i and v_j
b_{v_i,v_j,t_1,t_n}	the bandwidth of link $L_{i,j}$ from t_1 to t_n
d_{v_i,v_j}	the minimum link delay of link $L_{i,j}$
t_{start}	the time spent on setting up a virtual machine for the workflow running environment on a node
t_{shut}	the time spent on shutting down a virtual machine
$t_{v_j}(u_i)$	the execution time of module u_i running on node v_j
K_G	the total number of nodes that have been allocated for the workflow
$VM_{v_j,k}$	the k-th VM on the j-th node
M_{v_j}	the total number of VMs on the j-th node
$p_{VM_{v_j,k}}$	the computing power of $VM_{v_j,k}$

is completed and before the next module starts, calculated as $Idle(VM_{v_j,k}) = \sum_{u_i \in U}(st_i - et_{i-1})$. 4) The time to shut down that virtual machine is assumed to a constant of t_{shut}. Consequently, we can define the total resource cost for this workflow G_{wf} as:

$$TC(G_{wf}) = \sum_{i=1}^{N} z_{u_i} \cdot c_{u_i}(\cdot)$$
$$+ \sum_{j=1}^{K_G} \sum_{k=1}^{M_{v_j}} p_{VM_{v_j,k}} \cdot (t_{start} + Idle(VM_{v_j,k}) + t_{shut}), \quad (1)$$

where N is the total number of modules in a workflow, K_G represents the total number of nodes that have been allocated for the workflow, and M_{v_j} denotes the total number of VMs that have been set up on node v_j.

The Utilization Rate (UR) is defined as:

$$UR = \frac{\sum_{i=1}^{N} z_{u_i} \cdot c_{u_i}(\cdot)}{TC(G_{wf})}, \quad (2)$$

which measures the efficiency of the cloud resource utilization excluding VM overhead. Obviously, the cloud provider always desires to maximize this ratio, i.e. reduce the cost to improve the resource utilization rate, which leads to a higher system throughput. For convenience, we provide a summary of the notations used in the cost models in Table I.

Our mapping objective is to select an appropriate set of virtual nodes to set up VM instances for running computing modules to achieve the Minimum End-to-end Delay (MED) for fast response, which is an important performance requirement in time-critical applications especially for interactive operations. The utilization rate can be improved by cutting down the VM startup, shutdown and idle time. Our approach chooses the mapping scheme that results in a higher UR under the same End-to-End Delay (EED). Once a mapping schedule is determined, EED is calculated as the total time incurred on the critical path (CP), i.e. the longest execution path from the source module to the destination module.

IV. Problem Formulation

A schedule S with the maximum resource utilization rate may be obtained by simply mapping all the modules onto one node. However, in general, such a schedule S has a much longer EED than the optimal one.

We first consider a bi-objective scheduling problem to minimize the EED and maximize the utilization rate (or to minimize the total overhead). However, these are two conflicting objectives and cannot be achieved at the same time, as stated in Theorem 1.

Theorem 1. *The bi-objective problem of minimizing the EED and maximizing the utilization rate is non-approximable within a constant factor.*

Proof: We consider a simple instance of the problem that involves only three modules, u_1, u_2 and u_3, and two computing nodes, v_1 and v_2, whose computing powers have a relationship $p_{v_1} = kp_{v_2}$. We assume a constant VM startup time SA and shutdown time SU. The computational complexity of modules u_1 and u_2 has a relationship $c_{u_1}(\cdot) = kc_{u_2}(\cdot)$, and they provide two input datasets to u_3. The link bandwidth between these two nodes is a constant $b_{v_1,v_2,t} = B_{v_1,v_2}$ without any other transfer task scheduled. The data size transferred from u_2 to u_3 is represented by $z_{12} = mB_{v_1,v_2}$. Assume that $z_{u_1} = z_{u_2}$ and the data transfer time is small enough to be ignored compared with the module running time. Note that data transfer in cloud environments is fast and such cost is typically not included in the user bill. There exist two feasible solutions S1 and S2:

(i) S1 is optimal for EED: The modules u_1 and u_3 are scheduled on node v_1, and module u_2 is scheduled on node v_2. Two independent virtual nodes can start up simultaneously. The EED of S1 is calculated in Eq. 3 where $c_{u_1}(\cdot) = kc_{u_2}(\cdot)$ and $p_{v_1} = kp_{v_2}$. As u_1 and u_2 are independent, they can run in parallel on two different virtual nodes, and thus only the latest running time needs to be counted for the EED. The efficiency resource (ERC), which is the useful cost for running the workflow (i.e. user payload), is computed in Eq. 4 . The utilization rate of S1 is calculated in Eq. 5.

$$EED(S1) = SA + \frac{c_{u_1(\cdot)} \cdot z_{u_1}}{p_{v_1}} + \frac{z_{12}}{b_{v_1,v_2}} + \frac{c_{u_3}(\cdot) \cdot z_{u_3}}{p_{v_1}} + SU. \quad (3)$$

$$ERC = (k+1)c_{u_2(\cdot)} \cdot z_{u_2} + z_{12} + c_{u_3}(\cdot) \cdot z_{u_3}. \quad (4)$$

$$UR(S1) = \frac{ERC}{ERC + (k+1)(SA + SU)p_{v_2}}. \quad (5)$$

(ii) S2 is optimal for the utilization rate: All the modules should be mapped to v_1, which is more powerful, to achieve a better EED with maximized utilization rate. To calculate the EED, the running time of u_1 and u_2 needs to be computed first.

In the beginning, two modules need to share the computing power of v_1 until u_2 is finished. The running time for u_2 is $\frac{2c_{u_2(\cdot)} \cdot z_{u_2}}{p_{v_1}} = \frac{2c_{u_1(\cdot)} \cdot z_{u_1}}{kp_{v_1}}$ when u_i is still in execution. When u_2 releases the node, the running time for the remaining portion of u_1 is $\frac{c_{u_1(\cdot)} * z_{u_1} - \frac{c_{u_1(\cdot)} * z_{u_1}}{k}}{p_{v_1}}$. Thus the EED(S2), ERC' and UR(S2) can be calculated as follows:

$$EED(S2) = SA + \frac{c_{u_1}(\cdot) \cdot z_{u_1}}{p_{v_1}} + \frac{z_{12}}{b_{v_1,v_2}} + \frac{c_{u_3}(\cdot) \cdot z_{u_3}}{p_{v_1}} + SU. \tag{6}$$

$$ERC' = (k+1)c_{u_2(\cdot)} \cdot z_{u_2} + z_{12} + c_{u_3}(\cdot) \cdot z_{u_3} = ERC. \tag{7}$$

$$\begin{aligned} UR(S2) &= \frac{ERC}{ERC + (SA + SU) \cdot p_{v_1}} \\ &= \frac{ERC}{ERC + k(SA + SU) \cdot p_{v_2}} > UR(S1). \end{aligned} \tag{8}$$

Since the transfer time is much faster than the running time, S1 has a smaller EED and also a smaller utilization rate, which contradicts our assumption on its optimality. Therefore, it is impossible to optimize both objectives at the same time. Thus, we attempt to maximize the utilization rate within the constraint of the largest acceptable EED. ■

We consider the following delay-constrained utilization maximization problem:

Definition 1. *Given a DAG-structured computing workflow $G_{wf} = (V_{wf}, E_{wf})$, and an arbitrary computer network in a cloud environment $G_{cn} = (V_{cn}, E_{cn})$ with time-dependent link bandwidth and node computing power, we wish to find a workflow mapping schedule such that the utilization rate is maximized within the largest acceptable end-to-end delay constraint, i.e. the execution time bound (ETB):*

$$\max_{all\ possible\ mappings} (UR_{G_{cn}}(G_{wf})),\ such\quad that\ EED \leq ETB. \tag{9}$$

Here, $UR_{G_{cn}}(G_{wf})$ is the product of the utilization rate of all the resources that are assigned to either run a module or transfer data as shown in Eq. 2. Apparently, a smaller number of resources yield a higher combined UR in the product.

V. Algorithm Design

We propose a two-step heuristic workflow mapping approach, referred to as Cost-Effective Virtual Machine Allocation Algorithm within Execution Time Bounds (CEVAET). In the first step, modules are assigned with different priority values based on a combined consideration of their complexities and whether or not they are on the critical path (CP). Topological sorting is performed to separate modules into different layers, which determines the mapping order of modules starting from the first layer. Each module is mapped to the underlying node, which results in the lowest partial EED from the starting module to the current one. Such module mapping steps are conducted iteratively until the difference in EED between two contiguous rounds falls below a certain threshold. In the second step, the resource utilization rate is improved by reducing the overhead of the virtual machine's startup and shutdown time as well as the idle time. Some strategies, including resource release to cut the idle time and module combination on the same VM, are used to reduce the startup, shutdown cost and idle time of VMs. The pseudocode of CEVAET is provided in Algorithm 1.

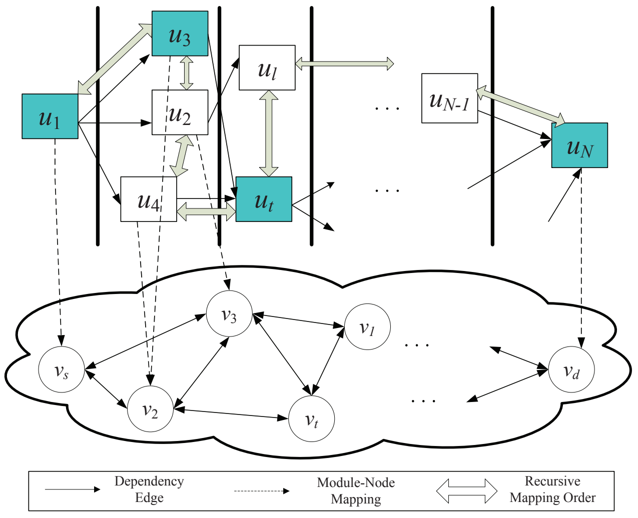

Fig. 4. Layer-ordered prioritized modules mapped to the underlying cloud.

Algorithm 1 CEVAET(G_{wf}, G_{cn}, t_s, ETB)

```
Input: workflow task graph $G_{wf}$, cloud network graph $G_{cn}$, workflow's earliest start time $t_s$, the execution time bound ETB
Output: a task scheduling scheme with the minimum resource cost within the given execution time bound
1: TimeDependentEEDMapping($G_{wf}, G_{cn}, t_s$);
2: ReduceResourceCost($G_{wf}, G_{cn}, M_{tm}, t_s, ETB$).
```

A. *Step 1: Minimized End-to-End Delay (MED)*

1) Construct a computing environment ${G_{cn}}^*$ with homogenous computing nodes and communication links to calculate the initial Critical Path (CP). Since our cloud environment supports in-advance resource reservations in addition to on-demand requests, the available resource capacity graph is time dependent and a set of time stamps are used to represent and track the resource non-changing periods.
2) **TimeDependentEEDMapping()** function as shown in Alg. 2 is called to map all modules to underlying cloud nodes. We first compute the CP by employing the well-known polynomial-time Longest Path (LP) algorithm, namely **FindCriticalPath()**, and then run the prioritized module mapping algorithm **PModulesMapping()** to map the workflow to the network graph until the convergence of EED is reached, as shown in Fig. 4.

The pseudocode of **PModulesMapping()** algorithm is provided in Algorithm 3. The algorithm first conducts topological sorting to sort modules into different layers. Each module is assigned a priority value depending on their computing and

Algorithm 2 TimeDependentEEDMapping (G_{wf}, G_{cn}, t_s)

Input: workflow task graph G_{wf}, cloud network graph G_{cn}, workflow's earliest start time t_s

Output: the temporary mapping scheme with the minimum end-to-end delay (MED)

```
 1: $i = 1$;
 2: ${G_{wf}}^*$ = mapped workflow based on ${G_{cn}}^*$;
 3: $CP_i$=**FindCriticalPath**$({G_{wf}}^*)$;
 4: call $MED_i$ = **PModulesMapping**$(G_{wf}, G_{cn}, t_s)$;
 5: update ${G_{wf}}^*$;
 6: **while** $|MED_i - MED_{i-1}| \geq Threshold$ **do**
 7:     $CP_i$=**FindCriticalPath**$({G_{wf}}^*)$;
 8:     call $MED_i$ =**PModulesMapping**$(G_{wf}, G_{cn}, t_s)$;
 9:     update ${G_{wf}}^*$;
10:     $i++$;
11: **end while**
12: return $MED_i$.
```

Algorithm 3 PModulesMapping(G_{wf}, G_{cn}, t_s)

Input: workflow task graph G_{wf}, cloud network graph G_{cn}, workflow's earliest start time t_s

Output: the temporary mapping scheme with the best EED namely MED

```
 1: **for all** $u_j \in$ CP **do**
 2:     set $u_j$.flag = 1;
 3: **end for**
 4: conduct topological sorting and assign the priority value to each module;
 5: dMinMED = $\infty$;
 6: **for all** $u_i \in$ SortedArray **do**
 7:     set PreModIDArr = the list of $u_i$'s pre-modules;
 8:     set PreMappedNodeArr = PreModIDArr's modules mapped node
 9:     CommonNodeArr = the nodes which has direct links to all the nodes
        in PreMappedNodeArr;
10:     **for all** $v_j \in$ CommonNodeArr **do**
11:         calculate the start running time for $u_i$ run on $v_j$
12:         call **GetPartialMED**() to calculate the partial EED for $u_i$ mapped
            on $v_j$;
13:         **if** EED in this round is smaller than previous round **then**
14:             update mapping result for current module;
15:         **end if**
16:     **end for**
17: **end for**
```

communication requirements. The module on the CP is given the highest priority value in the same layer. The modules starting from the first layer are mapped onto the appropriate node with the lowest partial execution time from the starting module. A backtracking strategy is adopted to adjust the mapping of the preceding modules (pre-module) of each newly mapped module's in order to further reduce its partial EED. The remapping of any of these pre-modules also triggers their succeeding modules (suc-module) to be remapped if necessary. Such back and forth remapping is only limited to one layer, i.e. confined within the affected area to control the algorithm complexity. The CP in Fig. 4 shown in shaded modules is given the highest priority in its layer. The forward order to map those modules could be, for instance u_1, u_3, u_2, u_4, u_t,..., u_{N-1}, u_N, as indicated by the shallow arrows.

We compute a new CP after each round of mapping of all modules and such mapping is repeated until the improvement of EED compared with the previous round is below a certain threshold.

The complexity of this iterative module mapping algorithm is $O(k \cdot l \cdot N \cdot |E_{cn}|)$, where l represents the number of layers in the sorted task graph, N represents the number of modules in the task graph, E_{cn} denote the number of links the cloud network graph, and k is the number of iterations where the obtained EED meets a certain requirement.

In the mapping algorithm, there are two different mapping cases. We either map a module to the same node as one of its mapped pre-modules (Case I), or map it to a different node which has direct links with all nodes onto which its pre-modules are mapped (Case II). We use Fig. 4 to illustrate an example for possible mapping choices. The upper figure represents the DAG-structured workflow with shaded modules along the CP. The lower figure shows the network graph. After the topological sorting, u_1 falls in layer 1; u_2, u_3 and u_4 fall in layer 2. The modules from layer 1 are mapped onto v_s first, then the modules from layer 2, and so on. For example, u_t has its pre-modules as u_3 and u_4, which are mapped onto v_3 and v_2, respectively. In Case I, u_t could be mapped either onto v_2 or v_3 since v_2 and v_3 are directly connected. In Case II, u_t could be mapped to v_s or v_t since v_s or v_t has direct links with both v_2 and v_3. The mapping strategy that leads to the lowest partial EED is chosen for that module. We assume that the inter-module communication cost within the same node is negligible as the data transfer within the same memory is typically much faster than that across a network. Since the resource capacity is time dependent in a cloud environment, instead of calculating one partial EED for each possible mapping, we calculate K (i.e. the number of time slots for one cloud node) possible partial EED.

After we map the downstream layer, we adjust its upstream layer's modules depending on its current mapping result. For example, in the above case u_t is mapped to v_t. We need to adjust its pre-modules u_3 and u_4. We first find out all possible nodes which have direct links with the mapping nodes of both its pre-modules and suc-modules. Take module u_3 as an example: its pre-module is u_1 which has been mapped to v_1, and its suc-module is u_t which has been mapped to v_t. So its possible mapping nodes would be either v_2 or v_3 which have direct links to these two nodes. During the adjustment process, we also need to calculate the partial EED. Instead of calculating the EED from the source module to the adjusted module, we calculate the partial EED from the source module to its latest finished suc-module.

This module mapping process is essentially a dynamic programming process. Let us define $u_j \in pre(u_i)$ as the set of pre-modules of our current mapping module u_i, $MN(u_j)$ as u_j's mapping node, $MNL(pre(u_i))$ as the list of mapping nodes of all the u_i's pre-modules, and $DLN(MNL(pre(u_i)))$ as the nodes which have direct links to the mapping nodes of all the pre-modules. We have the following recursive Eq. 10 leading to the minimal $EED(u_i, v_k)$ for the forward mapping.

Similarly, we define $u_l \in suc(u_i)$ as the set of u_i's suc-modules. The possible mapping nodes are limited to those nodes which have direct links to all the nodes onto which its pre-modules and suc-modules are mapped, denoted as

$$EED(u_i) = \min \begin{cases} \min\limits_{v_k \in MNL(pre(u_i))} (\max\limits_{u_j \in pre(u_i)} (EED(u_j, MN(u_j)) + \frac{w_{ij}}{b_{j,k}}) + \frac{z_{u_i} \cdot C(\cdot)}{p_k}) \\ \min\limits_{v_j \in DLN(MNL(pre(u_i)))} (\max\limits_{u_j \in pre(u_i)} (EED(u_j, MN(u_j)) + \frac{w_{ij}}{b_{j,k}}) + \frac{z_{u_i} \cdot C(\cdot)}{p_k}) \end{cases} \tag{10}$$

$$EED(u_i) = \min_{v_k \in DLN(MNL(pre(u_i), suc(u_i)))} (\max_{u_j \in pre(u_i)} (EED(u_j, MN(u_j)) + \frac{w_{ij}}{b_{j,k}}) + \frac{z_{u_i} \cdot C(\cdot)}{p_k} + \max_{u_l \in suc(u_i)} (\frac{w_{kMN(u_l)}}{b_{k,MN(u_l)}} + \frac{z_{u_l} \cdot C(\cdot)}{p_{MN(u_l)}})) \tag{11}$$

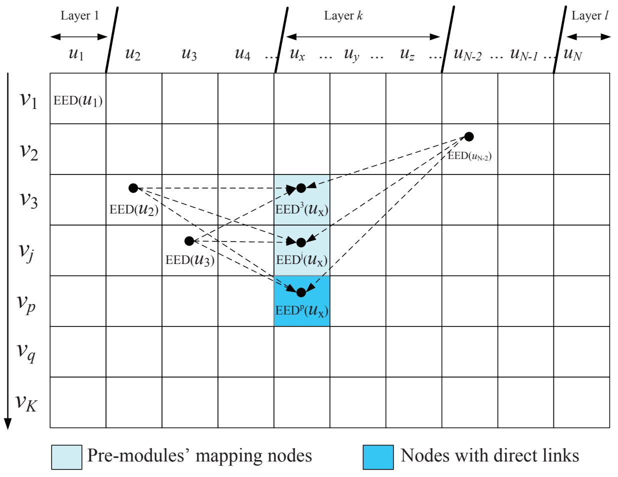
Layer 1
Layer k
Layer l
u_1 u_2 u_3 u_4 … u_x … u_y … u_z … u_{N-2} … u_{N-1} … u_N
v_1 v_2 v_3 v_j v_p v_q v_K
EED(u_1)
EED(u_{N-2})
EED(u_2)
EED^3(u_x)
EED(u_3)
EED^j(u_x)
EED^p(u_x)
Pre-modules' mapping nodes
Nodes with direct links

Fig. 5. 2D dynamic programming matrix for module mapping.

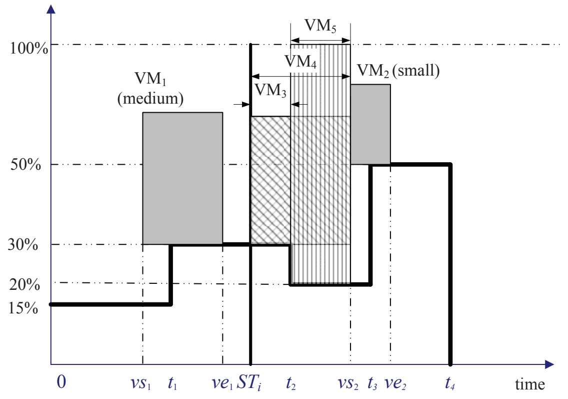

Fig. 6. Map module u_i with start running time ST_i on a cloud node with three possible VMs instances in forward mapping.

$DLN(MNL(pre(u_i), suc(u_i)))$. We also define a recursive equation to update $EED(u_i)$ as in Eq. 11 for the backward mapping. In Fig. 5, the updates of each partial EED entry in a 2D matrix are computed from several previous entries using a dynamic programming approach.

Fig. 6 illustrates how the partial EED is calculated for a module to be mapped on a cloud node. VM_1 and VM_2 are virtual machines that have been deployed to run some pre-modules. We can calculate the execution start time (ST_i) for module u_i, and then find out the time slot where ST_i is located. We check all the possible VM strategies, and select the one with the lowest partial EED. In this example, there are three possible VMs that can be allocated for u_i, namely, VM_3, VM_4 and VM_5. We calculate the execution time of u_i on VM_3 to obtain a partial EED, then check if the execution time is shorter than the life time of VM_3. If not, we calculate the execution time on VM_4; otherwise, we calculate the execution time on VM_5. We compare the partial EED on each VM, and select the one with the lowest partial EED.

B. Step 2: Reduce VM Overhead

In the second step of this algorithm, we want to reduce the VM overhead for the workflow while still meeting the user-specified execution time bound (ETB). The overheads in a cloud include setting up, shutting down and releasing a virtual machine as well as the virtual machine's idle time. The goal of this step is to reduce the unnecessary overheads and improve the resource utilization for better throughput that can be provided to users.

Algorithm 4 ReduceResourceCost($G_{wf}, G_{cn}, M_{tm}, t_s, ETB$)

Input: workflow task graph G_{wf}, cloud network graph G_{cn}, the mapping result from step 1, earliest start time t_s, execution time bound ETB.
Output: mapping result with the lowest cost UR within ETB.

```
1: Calculate the maximal acceptable running time for each module $i$ as $MART_i$;
2: SortedArray = topological and priority sort;
3: for all $u_i \in$ SortedArray do
4:     Set SucModIDArr = the list of $u_i$'s suc-modules;
5:     SucMappedNodeArr = the module SucModIDArr mapped node;
6:     CommonNodeArr = all nodes that have direct links with nodes in SucMappedNodeArr;
7:     SET findReuse = false;
8:     for all $v_j \in$ CommonNodeArr do
9:         if $v_j$ has allocated VM then
10:            call ReuseVM() to see whether we can reuse a VM on $v_j$ ;
11:            if $v_j$ has reusable VM then
12:                update mapping result;
13:                break;
14:            end if
15:        end if
16:        call AllocateNewVM() to allocate a new VM on $v_j$;
17:    end for
18: end for
```

We provide below a brief description of **ReduceResource-Cost()**, which is presented in Algorithm 4.

1) We combine the user-specified execution time bound (ETB) with the MED we calculate from Step 1 to compute the initial maximal acceptable running time (MART) for each module. The running time is calculated as $MART_i = RT_i \cdot \frac{ETB}{MED}$.

2) Perform topological sorting in a reverse direction starting from the destination module and assign the corresponding priority value for each module similar to Step 1.

3) For each module u_i from the last module to the first module in the reverse topological sorting list:

- Determine the list of u_i's suc-modules as SucMappedNodeArr.
- Determine the list of mapping nodes for all suc-modules as SucMappedNodeArr.
- Determine the list of all nodes that have direct links with nodes in SucMappedNodeArr as CommonNodeArr, including the candidate mapping nodes for the current module.
- Compare the mapping result for each possible mapping node. There would be two cases:
  i) If the mapping node has some allocated VMs, we then call **ReuseVM()** method to check whether or not we can reuse one of these VMs on that node. Two conditions must be satisfied if we reuse a module if possible: a) The available VM resource should be sufficient to run the module. b) Any possible idle time should be less than the time to shut down a VM and start up a new one. If both conditions are satisfied and the partial EED to this module is less than previously found one, we update the mapping information.
  ii) If the mapping node has no VMs or those VMs can not be reused, we call **AllocateNewVM()** to allocate a new VM for this module. The **AllocateNewVM()** is similar to **getPartialEED()**. We create a VM with the maximal allocable resource. Taking Fig. 7 as an example, we can calculate the end time of the module as ET_i. We have 3 different strategies to deploy a VM as VM_1, VM_2 or VM_3. Let ve_x be the VM's end time and vs_x be its start time. We calculate the running time for that module to be mapped on each VM as $\frac{z_{u_i} \cdot c_{u_i}(\cdot)}{p^{VM}_{v_j,vs_x,ve_x}}$. The allocable resource cost on a VM is $p^{VM}_{v_j,vs_x,ve_x} \cdot (ve_x - vs_x)$. We then compare the running time with the maximal running time of $MART_i$. If the running time is less than $MART_i$, this VM is acceptable and may be created. For all acceptable VMs, we compare their allocable amount of resources, and select the VM with the maximum amount of allocable resource. In this example, we would select VM_1 which has the largest area.
- Select the node and its corresponding VM which incurs the lowest VM overhead as the final mapping node/VM for this module.

4) Repeat Step 3 until all modules from this workflow have been mapped.

VI. Performance Evaluation

We implement the proposed CEVAET algorithm in C++ on a Windows 7 desktop PC equipped with Intel Core i5 CPU of 2.4GHz and 4.0GB memory. In the experiments, we compare our algorithm's utilization rate and end-to-end delay with Pegasus Workflow Manage System and Rank Match algorithm.

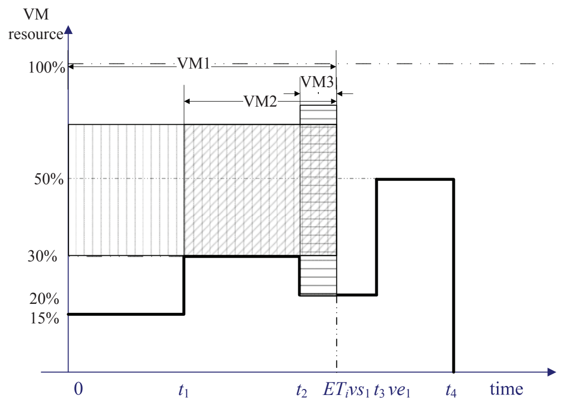

Fig. 7. Three different VMs to execute module u_i with end running time of ET_i in backward mapping.

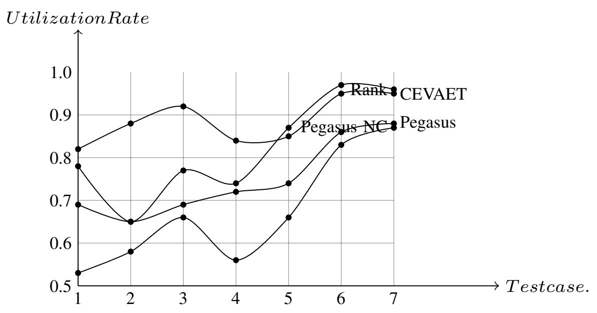

Fig. 8. Comparison of the utilization rate among different scheduling algorithms.

For the Pegasus workflow mapping, we run both the Clustering and non-Clustering versions. In the Rank Match algorithm, we use the cost of each possible mapping result as the rank value. We run seven tests on a set of randomly generated workflows and networks. The workflow mapping results in terms of utilization rate and EED are presented in Table II and further plotted in Figs. 8 and 9 for a visual comparison. These results demonstrate that our algorithm achieves a better mapping performance compared to Pegasus and Rank Match in terms of EED and utilization rate.

In each of the first three test cases, we map a workflow in a small cloud of 6 nodes. The Pegasus clustering algorithm (Pegasus) clusters tasks together, and achieves reduced overheads and therefore a better utilization rate compared to the Pegasus non-clustering one (Pegasus NC). Since we define the rank value based on the cost, the Rank Match always yields a better utilization rate compared to Pegasus. Since neither of these two algorithms considers EED during the mapping process, their EED increases significantly as the utilization rate increases. In our algorithm, we define the user-specified end-to-end delay as the EED of the cluster's mapping, and we observe that it produces the best utilization rate.

In each of the last four cases, we map a larger workflow in a cloud with 15 or more nodes. The Pegasus non-clustering algorithm still yields a lower utilization rate due to a random selection without considering reuse first. As the number of

TABLE II
WORKFLOW MAPPING EXPERIMENTAL RESULTS.

Index of Test Case	1	2	3	4	5	6	7
# of Modules	5	9	15	15	20	25	50
# of Nodes	6	6	6	15	15	15	30
UR (Pegasus NC)	0.53	0.58	0.66	0.56	0.66	0.83	0.87
UR (Pegasus)	0.69	0.65	0.69	0.72	0.74	0.86	0.88
UR (Rank)	0.78	0.65	0.77	0.74	0.84	0.97	0.96
UR (CEVAET)	0.82	0.88	0.92	0.84	0.85	0.95	0.95
EED (Pegasus NC)	28.61	23.4	35.78	34.56	94.7	336.99	529.46
EED (Pegasus)	18.23	42.8	48.38	32.98	88.96	335.29	512.19
EED (Rank)	35.2	27.46	42.8	52.69	213.2	733.1	1241.12
EED (CEVAET)	18.23	40	40	32.98	60	200	350

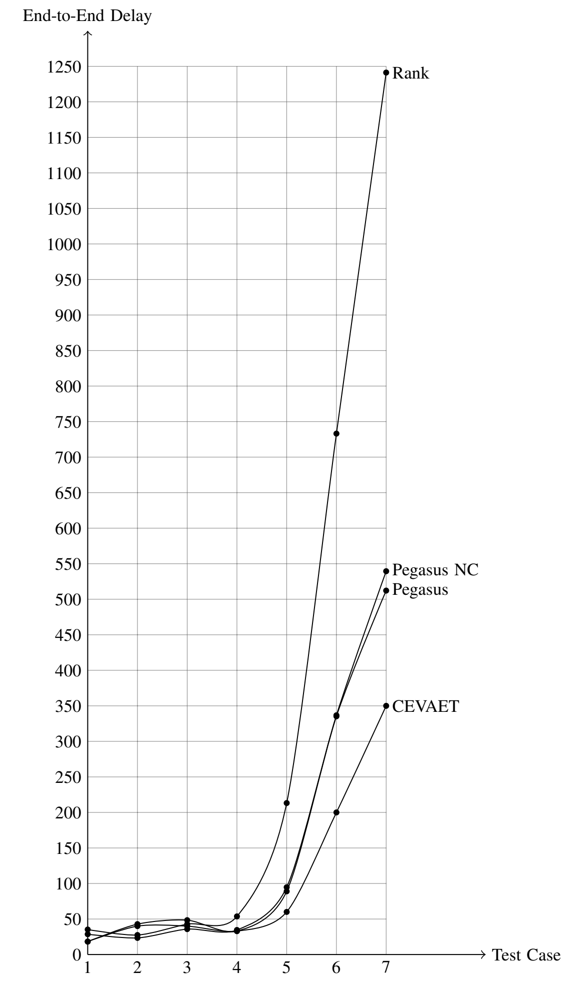

Fig. 9. Comparison of EED among different scheduling algorithms.

modules increases, the number of modules in the same layer would also increase, and therefore even a random selection would have a better chance to reuse a VM. The Pegasus clustering algorithm clusters modules together to force them to reuse a VM collectively, and hence exhibits a better mapping performance than the non-clustering one.

Our algorithm outperforms the Pegasus clustering algorithm. We use a much smaller EED for our algorithm and still obtain a much better mapping result. Different from the first three cases, we do not use the EED from the clustering algorithm because when the utilization rate approaches the value of 1, it does not help to increase the maximum allowed EED. In some cases, we observe that Rank Match achieves a better utilization rate; however, its EED is much higher than ours. These experimental results show that our algorithm has a better control of a workflow's running time compared to Pegasus and Rank Match, with a higher resource utilization rate by reducing the overhead of the workflow.

VII. CONCLUSION

We formulated a workflow scheduling problem in cloud environments. In general, it is of the cloud service provider's best interest to improve the system throughout to satisfy as many user requests as possible using the same hardware resources. Hence, the resource utilization rate is a very important performance metric, which, however, has not been sufficiently considered in the existing workflow scheduling algorithms developed specifically for clouds. Also, it is a common goal in scientific applications to minimize the execution time of each individual workflow to meet a certain Quality of Service requirement.

Our approach aims to achieve the dual goals of end-to-end delay performance and low overhead using a two-step approach. In the first step, modules are mapped from the first level to the last level in order to identify the best mapping strategy to minimize the execution time. If the final finish time is earlier than the latest finish time specified by the user, the extra allowed time delay is used to relax the mapping of modules to reduce the cost on VM setup and shutdown as well as the idle time. A backward remapping procedure for modules from the last layer toward the first layer is conducted to cut down the overhead. One strategy is to maximize the allocable volume of a VM to open the window for more modules to reuse it. After this backward mapping, any unused VM volume in terms of extra time is not requested. The simulation experiment results have demonstrated that our algorithm significantly reduces the VM cost compared with other representative cloud scheduling algorithms with comparable or lower total execution time. It is of our future interest to implement and test this scheduling algorithm in local cloud testbeds and production cloud environments to support real-life large-scale scientific workflows.

ACKNOWLEDGMENTS

This research is sponsored by U.S. DOE's Office of Science under Grant No. DE-SC0002400 with University of Memphis and Grant No. DE-SC0002078 with Southern Illinois University at Carbondale.

REFERENCES

[1] A. Bala and I. Chana. A Survey of Various Workflow Scheduling Algorithms in Cloud Environment. In *Proc. of the 2nd National Conference on Information and Communication Technology*, pp. 26-30, 2011.

[2] S. Zhang, X. Chen, and X. Huo. Cloud Computing Research and Development Trend. In *Proc. of the 2nd International Conference on Future Networks* (ICFN'10), pp. 93-97, 2010.

[3] J. Yu, R. Buyya, and C.K. Tham. Cost-based Scheduling of Scientific Workflow Applications on Utility Grids. In *Proc. of the 1st IEEE International Conference on e-Science and Grid Computing* (e-Science 2005), Dec. 5-8, 2005, Melbourne, Australia.

[4] C. Hoffa, G. Mehta, T. Freeman, E. Deelman, K. Keahey, B. Berriman, and J. Goo. On the Use of Cloud Computing for Scientific Workflows. In *Proc. of the IEEE 4th International Conference on eScience*, pp. 640-645, 2008.

[5] R. J. Figueiredo, P. A. Dinda, and J. A.B. Fortes. *A Case for Grid Computing On Virtual Machines*. In *Proc. of Distributed Computing Systems*, pp. 550-559, 2003.

[6] B. Sotomayor, K. Keahey, and I. Foster. Combining batch execution and leasing using virtual machines. In *Proc. of the 17th International Symposium on High Performance Distributed Computing* (HPDC'08), Boston, Massachusetts, USA, June 23-27, 2008.

[7] J. Vockler, G. Juve, E. Deelman, M. Rynge, and B. Berriman. Experiences using cloud computing for a scientific workflow application. In *Proc. of the 2nd International Workshop on Scientific Cloud Computing* (ScienceCloud'11), pp. 15-24, 2011.

[8] B. Agarwalla, N. Ahmed, D. Hilley, and U. Ramachandran. Streamline: a scheduling heuristic for streaming application on the grid. In *Proc. of the 13th Multimedia Computing and Networking Conf.*, San Jose, CA, 2006.

[9] D. Nurmi, R. Wolski, C. Grzegorczyk, G. Obertelli, S. So-man, L. Youseff, and D. Zagorodnov.The Eucalyptus open-source cloud-computing system. In *Proc. of IEEE International Symposium on Cluster Computing and the Grid* (CCGrid'09), 2009.

[10] *Open Nebular, http://www.opennebula.org*.

[11] *http://aws.amazon.com/ec2/*.

[12] *https://developers.google.com/appengine/*.

[13] *Nimbus, http://nimbusproject.org.*.

[14] E. Deelman, G. Singh, M. H. Su, J. blythe, and Y. e.a. Gil. Pegasus: A framework for mapping complex scientific workflows onto distributed systems. *Scientific Programming Journal*, vol. 13, pp. 219-237, July 2005.

[15] Q. Wu and Y. Gu. Supporting distributed application workflows in heterogeneous computing environments. In *Proc. of the 14th IEEE Int. Conf. on Parallel and Distributed Systems*, Melbourne, Australia, pp. 3-10, 2008.

[16] A. Sekhar, B. Manoj, and C. Murthy. A state-space search approach for optimizing reliability and cost of execution in distributed sensor networks. In *Proc. of Int. Workshop on Distributed Computing*, pp. 63-74, 2005.

[17] S. Topcuoglu and M. Wu. Task scheduling algorithms for heterogeneous processors. In *Proc. of the 8th IEEE Heterogeneous Computing Workshop* (HCW'99), pp. 3-14, 1999.

[18] Q. Wu, M. Zhu, X. Lu, P. Brown, Y. Lin, Y. Gu, F. Cao, and M. Reuter. Automation and management of scientific workflows in distributed network environments. In *Proc. of the 6th Int. Workshop on Sys. Man. Tech.*, pp. 1-8, 2010.

[19] Condor, *http://www.cs.wisc.edu/condor*.

[20] DagMan, *http://www.cs.wisc.edu/condor/dagman*.

[21] Globus, *http://www.globus.org*.

[22] T. Ma and R. Buyya. Critical-path and priority based algorithms for scheduling workflows with parameter sweep tasks on global grids. In *Proc. of the 17th Int. Symp. on Computer Architecture on High Performance Computing*, pp. 251-258, 2005.

[23] A. Mandal, K. Kennedy, C. Koelbel, G. Marin, J. Mellor-Crummey, B. Liu, and L. Johnsson. Scheduling Strategies for Mapping Application WorMows onto the Grid. In *Proc. of the IEEE International Symposium on High Performance Distributed Computing* (HPDC), pp. 125-134, 2005.