A Cost-Effective Deadline-Constrained Dynamic Scheduling Algorithm for Scientific Workflows in a Cloud Environment

Jyoti Sahni and Deo Prakash Vidyarthi, *Member, IEEE*

Abstract—Cloud computing, a distributed computing paradigm, enables delivery of IT resources over the Internet and follows the pay-as-you-go billing model. Workflow scheduling is one of the most challenging problems in cloud computing. Although, workflow scheduling on distributed systems like grids and clusters have been extensively studied, however, these solutions are not viable for a cloud environment. It is because, a cloud environment differs from other distributed environment in two major ways: on-demand resource provisioning and pay-as-you-go pricing model. Thus, to achieve the true benefits of workflow orchestration onto cloud resources novel approaches that can capitalize the advantages and address the challenges specific to a cloud environment needs to be developed. This work proposes a dynamic cost-effective deadline-constrained heuristic algorithm for scheduling a scientific workflow in a public cloud. The proposed technique aims to exploit the advantages offered by cloud computing while taking into account the virtual machine (VM) performance variability and instance acquisition delay to identify a just-in-time schedule of a deadline constrained scientific workflow at lesser costs. Performance evaluation on some well-known scientific workflows exhibit that the proposed algorithm delivers better performance in comparison to the current state-of-the-art heuristics.

Index Terms—Cloud computing, quality of service (QoS), resource provisioning, scheduling, scientific workflows

1 Introduction

Scientists in different research domains such as physics, bio-informatics, earth science and astronomy run increasingly complex large scale scientific applications for simulation and analysis of the real-world activities. Many of such large scale applications are usually constructed as workflows [1]. A workflow is a loosely coupled coarse-grained parallel application that consists of a set of computational tasks linked through control and data dependencies. Scientific workflows may vary in size from a few tasks with limited resource needs to millions of tasks requiring tens of thousands of processing hours, terabytes of storage and high bandwidth network resources. Such complex workflows demand a high-performance computing environment and often it is desirable to distribute its tasks amongst multiple computing nodes in order to complete the work in a reasonable time. Traditionally, developers of scientific applications have used local workstations, supercomputers, cluster and grid platforms for running such workflows. Each of these platforms offer various trade-offs in terms of usability, performance and cost. Many Grid projects such as Pegasus [2], ASKALON [3] and GrADS [4] have designed workflow management systems (WMSs) to define, manage and execute workflows on the Grid. Cloud computing, has recently emerged as a promising execution platform for huge and complex scientific applications. Many studies [5], [6], [7], [8], [9], [10] have investigated the use of cloud for scientific applications and have concluded that it offers reasonably good solutions in terms of performance and cost. However, with the emergence of this new computing paradigm, novel scheduling approaches that are able to capitalize the advantages while addressing the challenges specific to a cloud environment needs to be developed.

Cloud provides a utility-oriented computing model that enables delivery of IT resources over the Internet and follows the pay-as-you-go billing model where users are charged based on their resource consumption [11]. Cloud services are majorly categorized as: Infrastructure as a Service (IaaS), which includes raw infrastructure and associated middleware; Platform as a Service (PaaS), which includes APIs for developing applications on an abstract platform and Software as a Service (SaaS) that provides support for remote software Services. PaaS and SaaS based solutions are presently not considered as feasible alternatives for executing scientific workflows. This is because PaaS based solutions involve the overhead of porting legacy applications to new platforms whereas scientific computing SaaS services are currently rarely available for usage [7]. IaaS cloud on the other hand, offers several cost and performance related benefits for executing scientific applications as compared to traditional distributed execution environments like grids and clusters [6], [10]. Some of these benefits are as follows.

a) *Infinite economical resources:* Clouds give an illusion of unlimited computing resources, which may be provisioned on demand in a reasonable time frame and

• *The authors are with the School of Computer and Systems Sciences, Jawaharlal Nehru University, New Delhi 110067, India. E-mail: jyoti92_scs@jnu.ac.in, dpv@mail.jnu.ac.*

Manuscript received 30 Sept. 2014; revised 28 May 2015; accepted 24 June 2015. Date of publication 1 July 2015; date of current version 7 Mar. 2018. Recommended for acceptance by T. Fahringer.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TCC.2015.2451649

charged on a pay-per-use basis. Cloud platforms, thus, offers an alternative for executing scientific applications in which resources are no longer hosted by the research institutions but leased from big data centres as and when required. Outsourcing of scientific computation to cloud platforms may not only help in potentially lowering the financial burden of resource over-provisioning, but also in reducing the effort and cost of operating, maintaining and periodically upgrading local computing infrastructures.

b) *Direct on-demand provisioning:* In grids and clusters, user specifies the amount of time and the resources required for a computation and the responsibility of resource allocation is delegated to a batch scheduler. Thus, requests for resources are queued and served in accordance with the scheduling policies. The allocation of resources and binding of jobs to these resources is tied together and is out of user's control. In cloud, on the other hand, user directly provisions required resources to schedule their computations using a user-controlled scheduler. This helps in decreasing scheduling overheads and hence significantly improves the performance.

c) *Elasticity:* Cloud allows users to acquire and release the resources on demand. This allows the applications to easily grow or shrink its resource pool in accordance with the resource needs. Workflows usually have multiple stages, where the number of resources required for each stage may vary. Cloud-based workflow applications, may thus be allocated the exact amount of resources as and when required instead of reserving a fixed number of resources in advance. This not only ensures efficient resource utilization but also results in also results in good cost savings for the user.

Although, cloud has many advantages there still remains certain specific issues which needs to be addressed by the prospective scheduling policies. Some of these issues are discussed below.

a) *Performance variation in cloud:* Virtualization of resources, shared nature of infrastructure and the heterogeneity of the physical resources in cloud results in performance variability. Schad et al. [12] identified an overall CPU performance variability of 24 percent on Amazon's EC2 cloud. Performance variability may have an adverse effect when deadline constrained workflows are scheduled on clouds. Scheduling policies generally rely on the estimation of task runtimes on different VMs. This estimation is done based on the VMs computing capacity. If this capacity is always assumed to be optimal and the actual task execution takes longer time, the task will be delayed. This delay may then have a cascading effect on the child tasks causing the application to miss its deadline.

b) *Instance acquisition and termination delay:* When a VM is leased, it requires an initial boot time for proper initialization before it is made available to the user. Similarly, when a VM is released it needs some time to shut down. Long start-up time may result in delays leading to missing of deadlines and therefore needs to be accounted for in the schedule generation. On the other hand, VM termination delays do not adversely affect the application deadlines though may incur little extra cost on the user.

c) *Heterogeneous IaaS resources:* When leasing a VM from an IaaS provider, the user has the facility to choose different machines with varying configurations and prices. Any scheduling solution needs to make appropriate provisioning decisions considering the performance and cost trade-off.

The above discussed advantages and issues, dictate the development of innovative resource scheduling algorithms tailored for a cloud environment which may generate cost and performance effective solutions for scientific workflow execution. Workflow execution in a cloud environment involves two main stages. The first stage is the resource provisioning phase which identifies and provisions the computing resources required to run the tasks. In the second stage, a schedule is generated and each task is mapped onto the appropriate computing resources. The decisions taken at each of these stages are guided by the task's precedence constraints and performance requirements as specified by the user. Most of the previous works have focussed on planning workflows on distributed systems e.g. grids and clusters, confining only on the scheduling phase. This is because, grid and cluster environments provide a static pool of resources readily available to execute the tasks and whose configuration is known in advance. Cloud environment, however, requires that both the above-mentioned stages are addressed and combined in order to produce efficient execution plans. Another important characteristic of previous works developed for grids and clusters is that mostly they focus on minimizing the makespan (execution time) of the workflows. Though this is well suited for such environments, however, in cloud an important parameter besides execution time is the economic cost. Faster resources are usually costlier and therefore there is a time-cost trade-off in selecting appropriate services. Hence, scheduling policies developed for clouds should assess the various time/cost alternatives so as to deliver time effective solutions while avoiding unnecessary costs.

This work presents a dynamic cost-minimization deadline constrained heuristic for scheduling scientific applications in a public cloud environment. The proposed technique aims to exploit the advantages offered in cloud environment while taking into account the VM performance variability and instance acquisition delay to identify a just-in-time schedule of a scientific workflow which is able to meet its deadline at lesser costs.

The rest of this paper is organized as follows. Section 2 presents the scientific workflow application model and the architecture for workflow execution on elastic resources. Problem definition is presented in Section 3, followed by the proposed scheduling algorithm and its explanation in Section 4. Experiment based performance evaluation is presented in Section 5. Section 6 reviews the related work and Section 7 concludes the work.

2 Architecture for Workflow Execution in Cloud

This section explains the application model, cloud resource model and the overall architecture of the computing framework for workflow execution in cloud, used for this study.

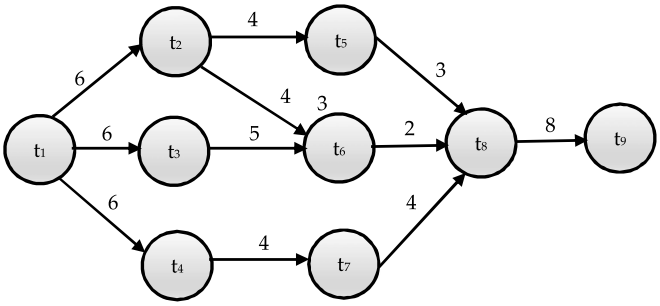


Fig. 1. A sample workflow.

2.1 Application Model

A workflow application W = (T, E) is modelled as a Directed Acyclic Graph (DAG) where $T = \{t_1, t_2, \ldots, t_n\}$, the set of vertices represents tasks and E is a set of directed edges representing data or control dependencies between the tasks. A dependency e_{ij} is a precedence constraint of the form (t_i, t_j), where, $t_i, t_j \in T$, and $t_i \neq t_j$. This implies that task t_j (child task) can start only after task t_i (parent task) has finished its execution and the associated data dependencies are transferred to t_j. Thus, a child task cannot execute until all its parent tasks have finished execution and the required dependencies, both data and control, obliged.

A deadline D is defined as the time limit specified by the user for the execution of the workflow. A sample workflow is shown in Fig. 1. Each node represents a task and the edges show the data transfer time between the tasks.

2.2 Cloud Resource Model

The cloud model, assumed in this work, consists of virtualized resources offered by an IaaS cloud service provider. The services include computation services e.g. Amazon Elastic Cloud Compute (EC2) [13] and storage services e.g. Amazon Elastic Block Store [14] used as a local storage device for storing the input/output files. All computation and storage services are assumed to be in the same data centre or region so that average bandwidth between computation services is roughly equal. Further, computation services are offered in form of different types of virtual machines (VMs). These VM types have varied configurations for CPU type, memory size and are available at different prices. It is assumed that there is no limitation on the number of VM instances leased (used) by an application. Also, when a VM is leased, it requires an initial boot time for its proper initialization before it is made available to the user. Similarly when a VM is released it again requires some time for proper shutdown. The pricing model is based on a pay-as-you-go billing scheme similar to the current commercial clouds and the users are charged for the number of *time intervals* they use (lease) a VM, even if the leased VMs have not been completely used in the last time interval. The *time interval* is specified by the cloud provider. For example cloud providers such as Amazon, charge the users based on the time interval of one hour. Thus, even if a VM is used only for few minutes, one has to pay for the whole hour. However, some service providers like Google Cloud Platform [24], have recently started with short time intervals and have per minute pricing model with a minimum of 10 minutes billing period. Since the internal data transfer is free in most cloud environments, the data transfer cost is assumed to be zero. Further, though real cloud providers charge for the storage services used for storing the input/output data files based on the allocated volume, they are not accounted for in the proposed model since these costs are independent of the scheduling algorithms. In the cloud resource model, VM type (VM_v) is defined by a two-tuple $\{(ET_{t_i})_v, C_v\}$ which specifies its estimated processing time for each task t_i and cost per *time interval* respectively. It is assumed that the estimated processing time for the tasks on different types of VMs can be achieved using some existing performance estimation techniques [15] (e.g. analytical modelling [16], empirical modelling [4] and historical data [18], [19]). The cost of running a task t_i on a VM of type VM_v, is calculated as $\lceil \frac{(ET_{t_i})_v}{time\ interval} \rceil \times C_v$. Further, the data transfer time $TT(e_{ij})$ between tasks scheduled on different VMs is calculated as $\frac{d_{t_i}}{\beta}$, where d_{t_i} is the size of the output data file to be transferred from t_i to t_j and β is the average bandwidth within the cloud datacentre. The only exception is when the parent task t_i and child task t_j both are scheduled on the same VM, in which case, the data transfer time $TT(e_{ij})$ becomes zero.

2.3 Computing Platform Model

Fig. 2 depicts the computing platform model for workflow execution on cloud resources. The computing platform,

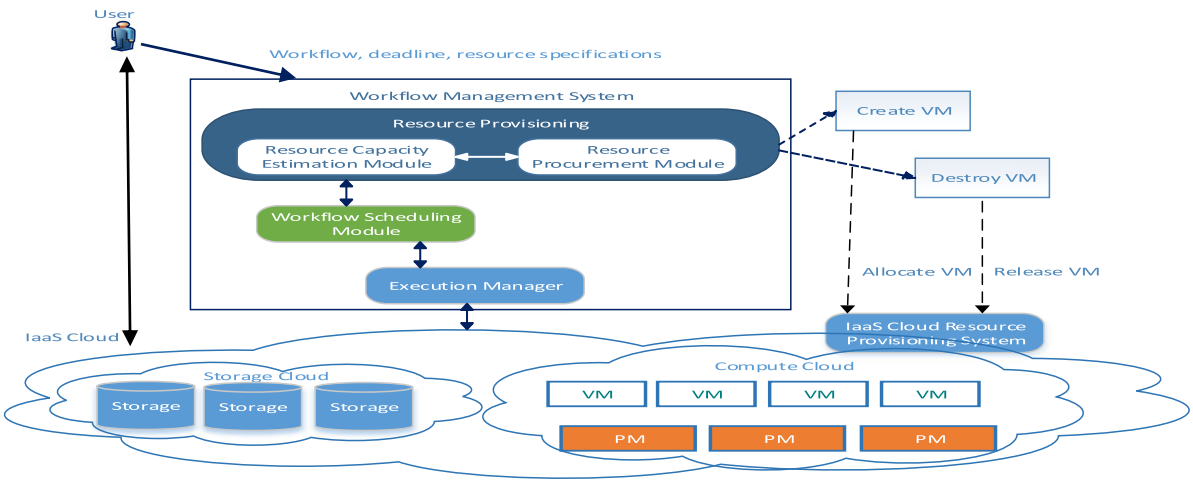


Fig. 2. Computing platform model for executing scientific workflows on cloud resources.

used in this study, is similar to the one used in [20]. Since cloud requires users to provision the appropriate amount of resources to run their applications (by identifying the resource type and the lease period), two main stages are involved when planning the execution of a workflow in a cloud environment. In the first stage resource provisioning is done which involves identifying and provisioning the appropriate computing resources for executing workflow tasks. The second stage involves generating a task execution schedule by mapping tasks onto appropriate resources.

A user submits a workflow along with the associated QoS requirements e.g. deadline constraint and resource specifications to the workflow management system. The deadline constraint specifies the time limit and the resource specifications describe the resource requirements (compute, memory, I/O) of the applications. Given these inputs, the WMS would automatically identify and provision the required resources, schedule tasks onto the provisioned resources and manage the workflow execution. In order to reduce the cost of running the application, the WMS acquires the resources as and when they are needed and releases them immediately after use. WMS consists of three main modules: *Resource Provisioning Module, Workflow Scheduling Module* and *Execution Manager*. Resource Provisioning module consists of two sub-modules: *Resource Capacity Estimation Module* and *Resource Procurement Module*. Resource Capacity Estimation module analyses the workflow structure to determine the amount of resources required. The Resource Procurement Module negotiates with the IaaS resource provisioning system to acquire the identified amount of resources. The Workflow Scheduling Module, in coordination with the Execution Manager, identifies the mapping between the provisioned resources and the tasks of the workflow. The scheduled tasks are then executed by the Execution Manager. The main difference between this computing model and other traditional high performance computing models is that resource allocation is workflow driven and the resource set size may vary during runtime.

3 Problem Definition

Resource provisioning and scheduling heuristics may have a number of objectives. The proposed work focusses on finding a just-in-time schedule to execute a workflow on an IaaS cloud such that total execution cost is minimized while meeting the user defined deadline constraint. In order to identify a schedule bounded by the deadline, it needs to be first ascertained if the user specified deadline is *achievable*. If it is not so, the scheduling problem would not have a feasible solution and therefore the user needs to revise the deadline. Once it is ascertained that the deadline is achievable, the goal is to make an appropriate scheduling decision for each task of the workflow within the given deadline at minimum possible cost. This involves making three decisions at each decision stage, which are as follows.

Cheapest task-VM mapping. The first decision, called as the *cheapest task-VM mapping*, is to determine the cheapest instance type for each task waiting to be scheduled

$$Cheapest\ task-VM\ mapping\ =\ \{(t_i \rightarrow VM_v)\}.$$

Provisioning Plan. The second decision, called as the *Provisioning Plan*, is to identify the number of instances of each VM type required at different stages of the workflow execution based on the status of the running tasks and the *Cheapest task-VM mapping* of the waiting tasks. Each VM, v_k, added to the resource pool of the workflow, has a *type* associated with it besides its start time st_k and end time et_k

$$VM\ Pool\ =\ \{v_k, type(v_k), st_k, et_k\}.$$

Scheduling Plan. The third decision called as the *Scheduling Plan*, is to determine the VM instance v_k of the resource pool on which a task t_i is to be scheduled with its estimated start time and end time

$$Schedule = \{t_i, v_k, estimated\ start\ time,\ estimated\ end\ time\}.$$

It may be noted that each provisioned VM needs to complete the transfer of all the output data files obtained by executing the tasks scheduled on it, to the local storage of the VMs on which the corresponding children tasks are to be scheduled, before it is deprovisioned. However, since cloud providers offer storage services that persist independent of VM lifetime (e.g. EBS [14]), the VMs on which a child task is to be executed need not be active when the input data files are being transferred to the corresponding storage volume. Further, due to performance variation and other delays, the tasks may not execute strictly as per the schedule. Therefore each task t_i, has an associated actual start time (AST) and an actual finish time (AFT) being maintained by the Execution manager. The scheduling algorithm needs to monitor the time difference between the actual and the estimated schedules in order to make appropriate decisions for the subsequent unscheduled tasks.

Based on the above, the problem can formally be defined as follows. Find a schedule *S* for a workflow application *W* that minimizes the total cost *C* of running the workflow and for which the total execution time *(TET)* does not exceed the workflow deadline *D*. It is thus an optimization problem as depicted in Equation (1):

$$\begin{array}{c} Minimize\ C = \sum_{k=1}^{|VMpool|} C_{type(v_k)} * \left\lceil \left(\frac{et_k - st_k}{time\ interval}\right) \right\rceil \\ subject\ to\ TET \leq D \quad Where,\ TET = \max_{t_i}\{(AFT(t_i)\}. \end{array} \tag{1}$$

4 The Proposed Workflow Scheduling Algorithm

The proposed algorithm is named as *just-in-time* (*JIT-C)* workflow scheduling algorithm for a cloud environment, which makes appropriate scheduling decisions just before the workflow tasks are ready for execution. In order to make appropriate scheduling/provisioning decisions, in the event of performance variation, the proposed algorithm uses a monitor control loop. Inside each loop, progress of running tasks is continuously monitored and resource provisioning/scheduling decisions are made based on the most recent information.

In order to accommodate for the VM acquisition delay, the algorithm takes as input, the *expected VM acquisition time* and makes the provisioning decisions accordingly. As discussed in Section 1, VM termination delay does not adversely affect the deadline constraint of the workflow. Moreover, service providers now offering short billing intervals the effect of

TABLE 1
Notation

Symbol	Meaning
$\mathrm{VM_{set}} = \{\mathrm{VM_1}, \mathrm{VM_2}, \ldots \mathrm{VM_m}\}$	Set of all VM types offered by the service provider
D	User defined deadline of the workflow
$\mathrm{ET(t_{i,}VM_v)}$	Execution Time of task $\mathrm{t_i}$ on VM type $\mathrm{VM_v}$
$\mathrm{TT(e_{ij})}$	Data Transfer Time from task $\mathrm{t_i}$ to $\mathrm{t_j}$
$\mathrm{MET(t_{\mathit{i}})}$	Minimum Execution Time of task $\mathrm{t_i}$
$\{\mathrm{t_{entry}}\}$	Tasks without any parent
$\{\mathrm{t_{exit}}\}$	Tasks without any child
$\mathrm{EST(t_i)}$	Earliest Start Time of task $\mathrm{t_i}$
$\mathrm{EFT(t_i)}$	Earliest Finish Time of task $\mathrm{t_i}$
$\mathrm{AST(t_i)}$	Actual Start Time of task $\mathrm{t_i}$
$\mathrm{AFT(t_i)}$	Actual Finish Time of task $\mathrm{t_i}$
$\mathrm{XST(t_i)}$	Expected Start Time of task $\mathrm{t_i}$
$\mathrm{XFT(t_i)}$	Expected Finish Time of task $\mathrm{t_i}$
$\mathrm{LST(t_i)}$	Latest Start Time of task $\mathrm{t_i}$
$\mathrm{LFT(t_i)}$	Latest Finish Time of task $\mathrm{t_i}$
$\mathrm{XET(t_i, VM_v)}$	Expected Execution Time of the tasks of the critical path starting at $\mathrm{t_i}$ on VM type $\mathrm{VM_v}$
$\mathrm{XIST(v_k)}$	Expected Idle Start Time of an active VM $\mathrm{v_k}$
VM_Pool_Status	5-tuple for each VM $\mathrm{v_k}$ in the VM pool $\{(\mathrm{v_k}, \mathrm{type(v_k)}, \mathrm{st}, \mathrm{XIST}, \mathrm{et})\}$
Acquistiondelay	Expected VM acquisition time
MET_W	Minimum Execution Time of the Workflow
$\mathrm{CLI}(v_\mathrm{k})$	Current Lease Interval of VM v_k

termination delay on cost is greatly reduced and hence is not accounted for in the proposed algorithm.

This section first discusses the basic definitions used in this work before elaborating the proposed scheduling algorithm. Finally, its operation is illustrated through an example.

Table 1 lists the notation used in the work.

4.1 Basic Definitions

The definition of some of the basic terms, used in the model, are as follows.

a) $MET(t_i)$: Minimum Execution Time of a task t_i is defined as the execution time of t_i on a VM instance type $VM_v \in VM_{set}$, which has the minimum execution time $ET(t_{i,}VM_v)$ among all the available VM types. It is calculated as given in Equation (2):

$$MET(t_i) = \min_{VM_v \in VM_{set}} \{ET(t_{i,}VM_v)\}. \tag{2}$$

b) $EST(t_i)$: Earliest Start Time of a task t_i is defined as the time at which t_i can start its execution, after all its predecessor tasks are scheduled on the fastest VM types and associated data dependencies have been transferred to t_i. It is obtained as given in Equation (3):

$$\left.\begin{array}{l} EST(t_{entry}) = 0 \\ EST(t_i) = \max\limits_{t_p \in t_{i's\, parent}} \{EST(t_p) + MET(t_p) + TT(e_{pi})\} \end{array}\right\}. \tag{3}$$

c) $EFT(t_i)$: Earliest Finish Time of a task t_i is defined as in Equation (4):

$$EFT(t_i) = EST(t_i) + MET(t_i). \tag{4}$$

d) $XFT(t_i)$: Expected Finish Time of a task t_i, scheduled on VM v_k, is defined as in Equation (5):

$$XFT(t_i) = \begin{cases} AST(t_i) + ET(t_i, type(v_k)), \text{ if } t_i \text{ is in execution} \\ \max\limits_{t_p \in t_{i's\, parent}} \{XFT(t_p) + TT(e_{pi})\} + ET(t_i, type(v_k)), \\ \text{if } t_i \text{ is waiting for execution.} \end{cases} \tag{5}$$

e) $XST\ (t_i)$: Expected Start Time of a task t_i is defined as the estimated time at which t_i can start its execution after all its predecessor tasks have been scheduled. It is calculated as in Equation (6):

$$\left.\begin{array}{l} XST(t_{entry}) = acquistiondelay \\ XST(t_i) = \max\limits_{t_p \in t_{i's\ parent}} \{XFT(t_p) + TT(e_{pi})\} \end{array}\right\}. \tag{6}$$

f) $XIST(v_k)$: Expected Idle Start Time of an active VM v_k of the resource pool is the time at which v_k is expected to finish the execution of the most recent task scheduled on it. Thus, if t_p is the most recent task scheduled on v_k then Expected Idle Start Time is obtained as in Equation (7):

$$XIST(v_k) = \begin{cases} AST(t_p) + ET(t_p, type(v_k)), \text{ if } t_p \text{ is in execution} \\ XST(t_p) + ET(t_p, type(v_k)), \text{ if } t_p \text{ is waiting} \\ \quad \text{for execution.} \end{cases} \tag{7}$$

g) $LFT\ (t_i)$: Latest Finish Time of a task t_i is defined as the latest time at which t_i can finish its execution such that all the tasks are executed before the user defined workflow deadline D. It is calculated as in Equation (8):

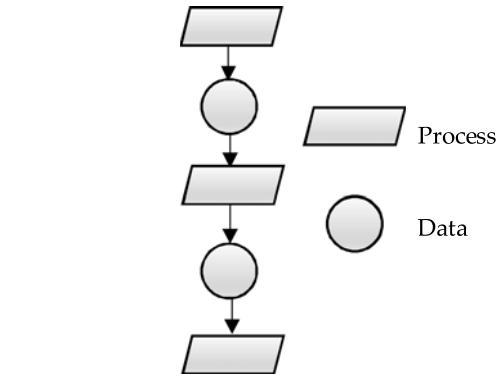


Fig. 3. Pipeline workflow.

$$\left.\begin{array}{l} LFT(t_{exit}) = D \\ LFT(t_i) = \min\limits_{t_c \in t_i's\ childern} \{LFT(t_c) - MET(t_c) - TT(e_{ic})\} \end{array}\right\}. \quad (8)$$

h) $LST\ (t_i)$: Latest Start Time of a task t_i is defined as the latest time at which t_i can start its execution such that all the tasks are executed before the user defined workflow deadline D. It is obtained as in Equation (9):

$$LST(t_i) = LFT(t_i) - MET(t_i). \quad (9)$$

i) $XET(t_i, VM_v)$: Expected Execution Time of critical path (longest execution path) starting at t_i on VM type VM_v is defined as the total time it would take to execute the entire critical path starting at t_i on VM type VM_v. It is calculated as in Equation (10):

$$\left.\begin{array}{l} XET(t_{exit}, VM_v) = ET(t_{exit}, VM_v) \\ XET(t_i, VM_v) = ET(t_i, VM_v) + \max\limits_{t_c \in t_i's\ children} \{XET(t_c, VM_v)\} \end{array}\right\}. \quad (10)$$

j) MET_W: Minimum Execution Time of the Workflow is defined as its critical path length (longest execution path), when all the tasks are executed on the fastest VMs. It is calculated as given in Equation (11):

$$MET\_W = \max_{t_i \in W}(EFT(t_i)). \quad (11)$$

k) CLI (v_k): Current Lease Interval of a leased VM v_k defines its current lease time span for which it will be charged. Given the time at which v_k is provisiond (st_k), unit of time used for billing (*time interval)* and the index of its current *time interval* (n), CLI (v_k) is calculated as given in Equation (12):

$$CLI(v_k) = [st_k, st_k + n \times time\ interval]. \quad (12)$$

4.2 The Proposed Algorithm (JIT-C)

The pseudo code of the proposed *JIT-C* algorithm is listed as Algorithm 1. It begins by verifying the *achievability* of the deadline *D* specified by the user. For a deadline to be *achievable*, it should be greater than the Minimum Execution Time of the Workflow, MET_W [39]. Therefore, the algorithm first evaluates MET_W and compares it with the user specified deadline *D*. If *D* is greater than MET_W, the algorithm continues to find the appropriate schedule, otherwise the user is prompted to revise the deadline *D*.

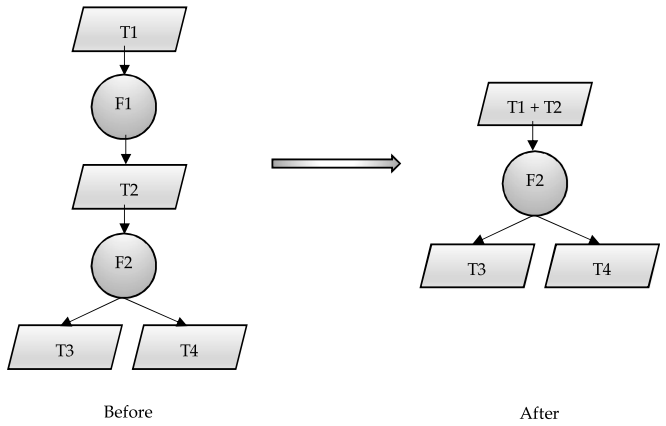


Fig. 4. Pre-processing.

Once the *achievability* of the user specified deadline is ascertained, the algorithm progresses to identify the required schedule by employing a pre-processing step and a monitor control loop. The pre-processing step is intended to reduce the runtime overhead of the algorithm by bundling pipeline tasks into a single task. The monitor control loop maintains the updated information of the running tasks and accordingly makes dynamic provisioning and scheduling decisions for the waiting unscheduled tasks. The remainder of this section explains the proposed algorithm in detail.

4.2.1 Pre-Processing

In order to reduce the runtime overhead of the algorithm, the workflow is pre-processed to combine pipeline tasks (Fig. 3) into a single task. This helps in saving the data transfer time involved in transferring data to the next stage of the pipeline and also accelerates dynamic scheduling/provisioning plan generation. Fig. 4 illustrates an example of pre-processing. Tasks T1 and T2 are combined and treated as one task T1 + T2. This allows T2 to be executed on the same site where T1 is executed enabling T2 to use the temporary results stored locally. It helps in avoiding the data transfer time between T1 and T2 if they would execute on different sites and also reduces the run time overhead of the monitor control loop. The pre-processing steps are elaborated in the Algorithm 2.

4.2.2 Monitor Control loop

After pre-processing, the proposed algorithm provisions the *cheapest applicable* resources required for executing the entry tasks, sends them for execution to the execution manager and enters into a monitor control loop. Since the entry tasks are the first tasks to be scheduled, the Expected Start Time of the entry tasks is set to the *expected VM acquisition time.* Within each monitor control loop, the algorithm updates the Actual Start Time of the scheduled running tasks, identifies the tasks whose parent tasks have been scheduled and are running (*to-be-scheduled* tasks) and then makes appropriate provisioning/scheduling decisions using the *Planand-Schedule* procedure. The monitor control loop continues until all workflow tasks are scheduled. At each stage, the *cheapest applicable* VM type for a task is determined by the *CheapesttaskVMmap* procedure and if a new VM is required

at time *T*, request for the VM is made at time *T- expected VM acquisition time* to ensure that the VM is available when it is required. The entire process is described in Algorithm 1.

Algorithm 1. Just-in-time Workflow Scheduling

Input:
DAG $W(T, E)$ of a job consisting on n tasks
$1 \times m$ cost matrix C of the m VM types offered by the cloud provider
$n \times m$ *ET* matrix of execution time of tasks $t_i (i = 1 \, to \, n)$ on each VM type $VM_v (v = 1 \, to \, m)$
$n \times n$ *TT* matrix of transfer time between tasks
User specified deadline *D* of the job
VM acquisition time *acquistiondelay*
Lease Time interval *interval*

1. Begin
2. Compute MET_W using Equations 2, 3, 4 and 11
3. If D $\geq MET\_W$
4. Call *Pre-processing(W)*
5. Compute *MET*, *LFT* and *XET* matrices using Equations (2), (8) and (10) respectively
6. $\{t_{entry}\} \leftarrow$ Root nodes of the workflow graph W
7. For each $t_e \in \{t_{entry}\}$
8. $To\_Provision \leftarrow CheapesttaskVMMap(t_e)$
9. Procure a VM instance v_e of type *To_Provision* from the cloud
10. Schedule t_e on v_e at $XST(t_e)$
11. Update VM_Pool_Status
12. End for
13. While all tasks in *T* are not completed do
14. Send the scheduled tasks for execution to the execution manager
15. Update *AST*, *XFT* of scheduled tasks
16. $to\_be\_scheduled \leftarrow \{t_i \in T \,|\, \forall t_p \in t_i' \, sparent, t_p$ is scheduled and running$\}$
17. $Planandschedule \, (\, to\_be\_scheduled)$
18. End while
19. Else
20. Prompt user to specify a deadline above MET_W
21. End If
22. End

Algorithm 2. Pre-processing (W)

Input: DAG $W(T, E)$ of a job consisting of n tasks.

1. Begin
2. $tksqueue \leftarrow \{t_{entry}\}$
3. While $tksqueue$ is not empty
4. $t_p \leftarrow tksqueue(front)$
5. $S_c \leftarrow \{t_c \,|\, \text{t}_\text{c} \text{ is the child of } t_p\}$
6. If $Cardinality \, (S_c) = 1$ and t_c has only one parent t_p
7. Replace t_p and t_c with t_{p+c}
8. Set t_{p+c} as the parent of $t_c's$ children tasks
9. Update $ET \, (\, t_{p+c})$
10. Add t_{p+c} to the *front* of $tksqueue$
11. Else
12. Add $t_p's$ children to the *rear* of $tksqueue$
13. End if
14. End While
15. End

PlanandSchedule algorithm. The PlanandSchedule algorithm receives a set of tasks as input and schedules them on *appropriate* VM instances. The algorithm, in coordination with the *CheapesttaskVMmap*, finds a minimum cost schedule in which least possible data transfer time is involved. For this, it tries to schedule a task onto its last parent (parent with maximum XFT) site to avoid delays incurred due to data transfer time if it is scheduled onto a different site. The pseudo-code for *PlanandSchedule* is given in Algorithm 3.

Algorithm 3. Planandschedule (task_list)

1. $Active\_VMs \leftarrow$ List of active VMs in the VM pool
2. For each $t_i \in$ task_list do
3. $vmmap \leftarrow CheapesttaskVMMap(t_i)$
4. $Find\{v_k\} \in Active\_VMs$ s.t. $type(v_k) = vmmap$ and $XST(t_i) \leq \text{CLI}(v_k)$ end time and $XFT \, (t_i) \leq LFT(t_i)$ and for no child t_c of $t_i \;\; XST(t_c) \; > \; LST(t_c)$
5. If $\{v_k\}$ exists
6. Find the VM v_k, such that the difference between $XIST(v_k)$ and $XST(t_i)$ is minimum
7. Schedule $t_i \; on \; v_k$ and update $XST(t_i)$
8. Update VM_Pool_Status
9. *Else*
10. Find$\{v_j\} \in Active_VMs$ s.t. $type(v_j) > vmmap$ and $XFT(t_i) \leq \text{CLI}(v_j)$ end time and $\text{XFT}(t_i) \leq LFT(t_i)$ and for no child t_c of $t_i \; XST(t_c) \; > \; LST(t_c)$
11. If $\{v_j\}$ exists
12. Find the VM v_j such that the difference between $XIST(v_j)$ and $XST(t_i)$ is minimum
13. Schedule $t_i \; on \; v_j$, update $XST(t_i)$
14. Update VM_Pool_Status
15. Else
16. Procure a new VMv of type $vmmap$ from the cloud at $(XST \;\; (t_i) - acquistiondelay)$
17. Schedule $t_i \; on \; v$ at $XST \;\; (t_i)$
18. Update VM_Pool_Status
19. End if
20. End if
21. End for
22. Deprovision the idle VMs
23. Return

For each task t_i in the task_list, *PlanandSchedule* calls the *CheapesttaskVMmap* procedure (line 3) which returns the *cheapest applicable* VM type $vmmap$ on which t_i can be scheduled. Since VMs are charged for entire *time intervals* even if they are partially utilized, therefore in order to use VMs as densely as possible, the *PlanandSchedule* algorithm tries to find if t_i can be scheduled on an already leased VM before its Current Lease Interval ends. For this, it first tries to find the set of active VMs $\{v_k\}$ of the same type as $vmmap$ such that the following conditions are satisfied.

i) t_i can be scheduled on v_k, such that t_i uses a portion of the remaining time of $\text{CLI}(v_k)$
ii) Expected Finish Time of $t_i \leq$ Latest Finish Time of t_i
iii) Assuming that t_i is scheduled on v_k, for no t_c such that $t_c \in \; t_i's$ children, Expected Start Time of $t_c >$ Lastest Start Time of t_c

If such a set of active VMs exist, the VM v_k which has the least difference between its Expected Idle Start Time and the Expected Start Time of t_i is selected from this set for

scheduling t_i. $XST(t_i)$ and VM_Pool_Status are updated accordingly (lines 4-8). Otherwise, *PlanandSchedule*, tries to find the set of active VMs $\{v_j\}$, such that $type(v_j) > vmmap$ and the following conditions are met.

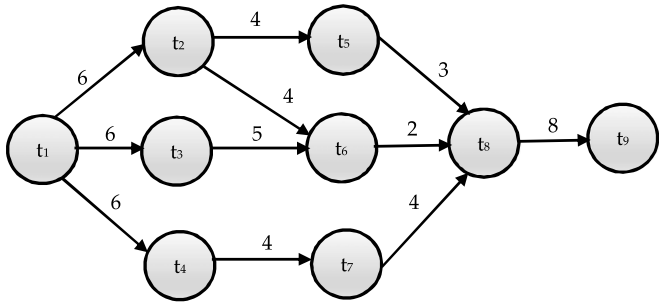
i) t_i can be scheduled on v_j such that t_i can finish its execution within the remaining time of $\mathrm{CLI}(v_j)$
ii) Expected finish time of $t_i \leq$ Latest Finish Time of t_i
iii) Assuming that t_i is scheduled on v_j, for no t_c such that $t_c \in t_i's$ children, Expected Start Time of $t_c >$ Latest Start Time of t_c

If such a set of VMs exist, the VM v_j which has the least difference between its Expected Idle Start Time (XIST) and the Expected Start Time of t_i is selected from this set for scheduling t_i. The $XST(t_i)$ and VM_Pool_Status are updated accordingly (lines 10-14). If none of the active VMs can be utilized for scheduling t_i, a new VM of type $vmmap$ is procured from the cloud to schedule t_i (16-18). After all the tasks in the task_list are scheduled, *PlanandSchedule* deprovisions the idle VMs on which no task is scheduled and which have completed the transfer of all the required output files (line 22).

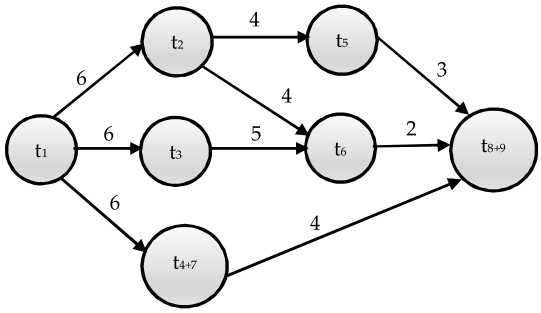
Algorithm 4. CheapesttaskVMMap(t)

1. Begin
2. $taskvmmap = \phi$
3. If *t* is not an entry task then
4. $lastParent \leftarrow arg\,(\max_{t_p \in t's\ parent}\{XFT(t_p)\})$
5. $v_p \leftarrow VM\ on\ which\ lastParent\ is\ running$
6. $temp \leftarrow \max(XFT(lastParent), \max_{t_p \in t's\ parent\ and\ t_p \neq lastParent}\{XFT(t_p) + TT(t_p, t)\})$
7. If $((temp \geq XIST(v_p))\mathrm{and}(temp + XET(t, type(v_p))) \leq D)$ then
8. $XST(t) \leftarrow temp$
9. $taskvmmap \leftarrow type(v_p)$
10. Return $taskvmmap$
11. Else
12. $XST(t) \leftarrow \max_{t_p \in t's\ parent}\{XFT(t_p) + TT(t_p, t)\}$
13. End if
14. Else
15. $XST(t) \leftarrow acquisitiondelay$
16. End if
17. Find $\{VM_k\} \in VM_{\mathrm{set}}$ for which $(XST(t) + XET(t, VM_k)) \leq D$
18. $VM_j = arg\,(\min_{VM_k}(XET(t, VM_k)/\mathrm{interval}) \times Cost(VM_k))$
19. $taskvmmap \leftarrow VM_j$
20. Return $taskvmmap$

CheapesttaskVMmap Algorithm. The *CheapesttaskVMmap* algorithm, receives a task t as input and returns its *cheapest applicable* VM type $taskvmmap$. It is observed that though the *cheapest* VM type may be able to finish a task before its latest finish time, however it may not be the best choice. This is because choosing the cheapest VM type for a task, without considering its effect on the children tasks, may force the children tasks to execute on faster VMs thus increasing the total cost. Therefore, this work defines *cheapest applicable* VM type for a task as the cheapest type single VM, which if used to schedule all the tasks of the critical path (longest path) beginning at t, can finish the execution of the entire critical path before the deadline *D*. Also, as discussed, the objective is to find a minimum cost schedule



5 (a) Sample workflow



5 (b) After Pre-processing

Fig. 5. An example of workflow pre-processing.

with least possible data transfer time. Therefore, assuming that the task t cannot wait for a VM to get idle (free) the algorithm first tries to identify if task t can be scheduled on the same VM v_p on which its *last parent* (parent with the maximum Expected Finish Time) is scheduled. For this $\mathrm{XST}(t)$, if t is scheduled on v_p, is first evaluated and compared with the $\mathrm{XIST}(v_p)$. If $\mathrm{XIST}(v_p)$ is less than $\mathrm{XST}(t)$ (which means v_p will be idle at the Expected Start Time of t and hence is available for scheduling t) and the critical path (longest path) starting at t, if scheduled on v_p, can finish its execution before the deadline *D*,then the $taskvmmap$ is set to $type(v_p)$ and the $\mathrm{XST}(t)$ is updated accordingly (lines 4-10). The *PlanandSchedule* procedure then schedules t on v_p. Otherwise, $XST(t)$ is updated and the cheapest VM type $taskvmmap$ for t is identified using the following rules (lines 12-18).

i) Identify the set $\{VM_k\}$ of VM types such that the critical path starting at t, if scheduled on a single VM of type VM_k, can finish its execution before the deadline *D*.
ii) From the set $\{VM_k\}$ identify the VM type VM_j for which the total execution cost of this critical path is minimum.

4.3 An Illustrative Example

An example has been illustrated, in this section, for better understanding of the algorithm. The steps of the algorithm has been traced on a sample workflow shown in Fig. 5a. The workflow consists of nine tasks t_1 to t_9. The number on each arc shows the estimated data transfer time between the corresponding tasks. Three types of VMs are assumed

$$ExecTime = \begin{array}{c} \\ t_1 \\ t_2 \\ t_3 \\ t_4 \\ t_5 \\ t_6 \\ t_7 \\ t_8 \\ t_9 \end{array} \begin{array}{c} \begin{array}{ccc} v_s & v_m & v_l \end{array} \\ \begin{bmatrix} 4 & 2 & 1 \\ 6 & 4 & 2 \\ 16 & 9 & 6 \\ 12 & 7 & 4 \\ 11 & 8 & 5 \\ 7 & 3 & 2 \\ 18 & 12 & 8 \\ 13 & 9 & 5 \\ 15 & 12 & 9 \end{bmatrix} \end{array}$$

(a)

$$TransferTime = \begin{array}{c} \\ t_1 \\ t_2 \\ t_3 \\ t_4 \\ t_5 \\ t_6 \\ t_7 \\ t_8 \\ t_9 \end{array} \begin{array}{c} \begin{array}{ccccccccc} t_1 & t_2 & t_3 & t_4 & t_5 & t_6 & t_7 & t_8 & t_9 \end{array} \\ \begin{bmatrix} 0 & 6 & 6 & 6 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 4 & 4 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 5 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 4 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 3 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 2 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 4 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 8 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \end{array}$$

(b)

Fig. 6. (a) Execution time matrix and (b) Data transfer time matrix of the sample workflow.

TABLE 2
The values of MET, EST and EFT for the workflow of Fig. 5a

Tasks	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	t_9
MET	1	2	6	4	5	2	8	5	9
EST	0	7	7	7	13	18	15	27	40
EFT	1	9	13	11	18	20	23	32	49

TABLE 3
The values of MET, LFT and XET for the workflow of Fig. 5b

Deadline D = 50							
Tasks	t_1	t_2	t_3	t_{4+7}	t_5	t_6	t_{8+9}
MET	1	2	6	12	5	2	14
LFT	14	24	27	32	33	34	50
XET(V_s)	62	45	51	58	39	35	28
XET(V_m)	42	33	33	40	29	24	21
XET(V_l)	27	21	22	26	19	16	14

$\{VM_s, VM_m, VM_l\}$ (*s-small, m-medium, l-large*), for the workflow tasks' execution. Fig. 6 depicts the corresponding execution time and data transfer time matrix. The time interval of the minimum lease period is assumed to be 10 minutes and the acquisition delay is assumed to be 1 minute. Also, the cost of each time interval is assumed to be \$ 0.01 for small, \$ 0.02 for medium and \$0.04 for large VM instance types. The user defined deadline is set to be 50 minutes. Although, the proposed algorithm is capable of making appropriate provisioning/scheduling decisions in the event of performance variation, for simplicity this example assumes that there is no performance variation and the Actual Start Time of all the tasks are same as their Expected Start Time obtained during the schedule planning.

The algorithm begins by evaluating the Minimum Execution Time, Earliest Start Time and Earliest Finish Time for each task of the workflow using Equations (2), (3) and (4) respectively. The values of these parameters are shown in Table 2. It then compares the Minimum Execution Time of the Workflow, $MET\_W(\max_{t_i \in W}(EFT(t_i) = 49$ minutes) with the deadline D (50 minutes). Since MET_W is greater than D, the deadline is achievable and the algorithm proceeds to identify the required schedule.

Step 1: *Pre-processing the workflow using Algorithm 2*. Algorithm 2 pre-processes the given workflow graph (Fig. 5a) by combining the sequential tasks t_4, t_7 and t_8, t_9 into a single task as t_{4+7} and t_{8+9} respectively. The workflow graph obtained after pre-processing is shown in Fig. 5b. The modified execution time and data transfer time matrix is shown in Fig. 7.

After pre-processing, the *Just-in-time workflow scheduling* algorithm (Algorithm 1) computes Minimum Execution Time, Latest Finish Time and Expected Execution Time for each task of the workflow using Equations 2, 8 and 10 respectively. The values of these parameters are shown in Table 3.

The algorithm then calls the procedure *CheapesttaskVMmap* to identify the most cost effective VM type to schedule the entry task t_1. *CheapesttaskVMmap* sets the XST (t_1) to $acquisitiondelay$ and finds the cheapest applicable VM type for its execution. Since out of the three VM types, XET (t_1, VM_m) and XET (t_1, VM_l) are less than the deadline D, the algorithm compares the cost of executing the critical path starting at t_1 on both these VM types and identifies VM_m as the *cheapest applicable* VM for scheduling t_1. It then procures a VM of type VM_m, schedules t_1 on it and updates the VM pool status. The algorithm enters the while loop at step 13 and executes the loop until all the tasks are scheduled. A trace of the algorithm is given in Table 4 which lists the

$$ExecTime = \begin{array}{c} \\ t_1 \\ t_2 \\ t_3 \\ t_{4+7} \\ t_5 \\ t_6 \\ t_{8+9} \end{array} \begin{array}{c} \begin{array}{ccc} v_s & v_m & v_l \end{array} \\ \begin{bmatrix} 4 & 2 & 1 \\ 6 & 4 & 2 \\ 16 & 9 & 6 \\ 30 & 19 & 12 \\ 11 & 8 & 5 \\ 7 & 3 & 2 \\ 28 & 21 & 14 \end{bmatrix} \end{array}$$

(a)

$$TransferTime = \begin{array}{c} \\ t_1 \\ t_2 \\ t_3 \\ t_{4+7} \\ t_5 \\ t_6 \\ t_{8+9} \end{array} \begin{array}{c} \begin{array}{ccccccc} t_1 & t_2 & t_3 & t_{4+7} & t_5 & t_6 & t_{8+9} \end{array} \\ \begin{bmatrix} 0 & 6 & 6 & 6 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 4 & 4 & 0 \\ 0 & 0 & 0 & 0 & 0 & 5 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 4 \\ 0 & 0 & 0 & 0 & 0 & 0 & 3 \\ 0 & 0 & 0 & 0 & 0 & 0 & 2 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \end{array}$$

(b)

Fig. 7. (a) Modified execution time matrix and (b) Data transfer time matrix of the sample workflow in Fig. 5b after pre-processing.

TABLE 4
Values of the Parameters during the Planning and Execution of the Workflow Tasks of Fig. 5b

Entry Tasks

t_1

$XST(t_1) = 1$
To_Provision=VM_m

VM Pool Status

VM id	VM Type	Start Time	Expected Idle Start Time	End Time
v_1	VM_m	0	3	-

Schedule

Task	VM id	XST	XFT
t_1	v_1	1	3

Iteration 1

to_be_scheduled

t_2	t_3	t_{4+7}

$XST(t_2) = 3$
taskvmmap=VM_m

$XST(t_3) = 9^*$
taskvmmap=VM_m

$XST(t_{4+7}) = 9$
taskvmmap=VM_m

VM Pool Status

VM id	VM Type	Start Time	Expected Idle Start Time	End Time
v_1	VM_m	0	16	-
v_2	VM_m	(9-1)=8	28	-

Schedule

Task	VM id	XST	XFT
t_2	v_1	3	7
t_3	v_1	7*	16
t_{4+7}	v_2	9	28

*While evaluating the most applicable VM type for task t_3, *CheapesttaskVMmap* procedure observes that t_2 is already scheduled on t_1 (the *lastparent* of t_3) and $EIST(v_1) = 7$, which is greater than $temp$ (= 3) and therefore it updates the $XST(t_3)$ to 9. The *PlanandSchedule* module however, while identifying the best possible plan, finds that t_3 can be scheduled on v_1 and subsequently updates the $XST(t_3)$ to 7.

Iteration 2

to_be_scheduled

t_5	t_6

$XST(t_5) = 11$
taskvmmap=VM_s

$XST(t_6) = 16$
taskvmmap=VM_m

VM Pool Status

VM id	VM Type	Start Time	Expected Idle Start Time	End Time
v_1	VM_m	0	19	-
v_2	VM_m	8	28	-
v_3	VM_s	(11-1)=10	22	-

Schedule

Task	VM id	XST	XFT
t_5	v_3	11	22
t_6	v_1	16	19

Iteration 3

to_be_scheduled

t_{8+9}

$XST(t_{8+9}) = 28$
taskvmmap=VM_m

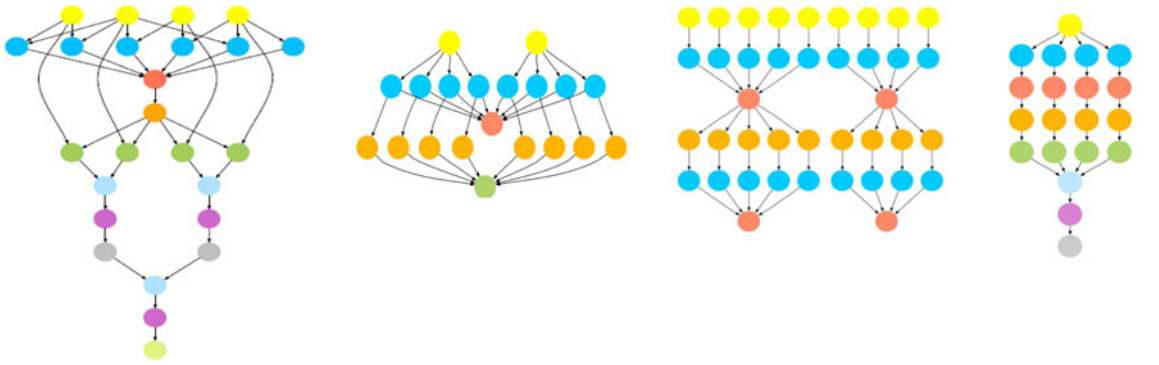
VM Pool Status

VM id	VM Type	Start Time	Expected Idle Start Time	End Time
v_1	VM_m	0	19	21
v_2	VM_m	8	49	49
v_3	VM_s	10	22	25

Schedule

Task	VM id	XST	XFT
t_{8+9}	v_2	28	49

Total Execution Time = 49 minutes
Total Cost = $ 0.18



(a) Montage (b) CyberShake (c) LIGO (d) Epigenomics

Fig. 8. Structure of the workflows used in the experiment [21].

values of different parameters during the planning and execution of the workflow tasks.

5 Performance Evaluation

This section lists the experiments conducted to evaluate the performance of the proposed algorithm.

5.1 Experimental Workflows

The proposed algorithm was evaluated on the following four real application workflows used in diverse scientific domains.

a) *Montage:* Montage, is an astronomical application which is used to generate custom mosaics of the sky based on a set of images. Most of its tasks are characterized as I/O intensive which do not require much processing capacity.
b) *CyberShake:* CyberShake is used in earthquake science to characterize earthquake hazards in a region by generating synthetic seismograms. It may be classified as a data intensive workflow with large memory and CPU requirements.
c) *LIGO:* LIGO workflow is used in gravitational physics for detecting gravitational waves produced by various events in the universe. This workflow is characterized as having CPU intensive tasks that consume large memory.
d) *Epigenomics:* Epigenomics workflow is used in bioinformatics to map the epigenetic state of human cells on genome-wide scale. Most of the tasks in this workflow have high CPU and low I/O utilization.

A detailed description of these workflows is presented by Juve et al. [21]. Fig. 8 shows the structure of small size workflows for each of these applications. It can be seen that these workflows have different composition and structural properties (pipeline, data aggregation, data distribution and data redistribution).

In order to facilitate evaluation of workflow algorithms and systems, Bharti et al. [22] developed a workflow generator to create synthetic workflows of arbitrary size similar to the real world scientific workflows. The generated workflows are represented in form of Directed Acyclic Graph in XML (DAX) format and are available in [33]. These DAX files contain information such as list of tasks, dependencies between tasks, their computation time and size of the input/output files generated by the tasks. In order to evaluate the proposed algorithm, experiments were conducted for each of the above applications on three workflow sizes: small (approximately 30 tasks), medium (approximately 100 tasks) and large (approximately 1,000 tasks).

5.2 Baseline Algorithms

Two recent contributions to the workflow scheduling problem in a cloud environment proposed in [23] and [40] have been used as the baseline algorithms. The IaaS Cloud Partial Critical Paths (IC-PCP) algorithm, proposed in [23], is one of the most cited algorithms for the same problem addressed in this work; Schedule a workflow in an IaaS cloud while minimizing the execution cost and meeting the application's deadline. On the other hand, authors in [40] proposed a robust and fault-tolerant workflow scheduling algorithm that handles performance variations of cloud resources and failures in the environment. They proposed three multi-objective resource selection policies to schedule workflows in a cloud environment that minimizes the makespan and cost. Both these algorithms consider many characteristics typical of a cloud environment. For example, they account for the heterogeneous VM instance types which can be provisioned on demand and are charged based on pay-as-you-go billing model. They also consider data transfer time in addition to the computation time of each task. However, *IC-PCP*, does not account for the performance variation of VMs and the acquisition delay involved in provisioning a VM which has been taken into consideration both in the proposed work and in [40].

The *IC-PCP* algorithm is based on workflow's partial critical paths (PCPs). It begins by identifying a critical path, associated with each exit node of the workflow. The tasks on each critical path are scheduled on the cheapest available VM, preferably to an already leased VM instance, which can meet the latest finish time requirements of all the tasks in the critical path. If none of the already leased VM instances can meet the latest finish time constraints of the tasks, the cheapest instance which can finish all the tasks in the critical path while meeting the latest finish time constraint is provisioned and the path is assigned to it. This process is repeated until all the tasks of the

TABLE 5
Types of VMs Used in the Experiments

VM Type	ECUs(cores)	Processing Capacity (GFLOPS)	Cost per hour ($/h)
m1.small	1(1)	4.4	$.04
m1.large	4(2)	17.6	$.16
m1.xlarge	8(4)	35.2	$.32
c1.medium	5(2)	22.0	$.2
c1.xlarge	20(8)	88	$.8

workflow are scheduled. At the end, each task has a VM assigned with the associated start and end time. Also, each leased VM has a start time determined by the start time of its first scheduled task and an end time determined by the end time of its last scheduled task.

The robust scheduling algorithm proposed in [40] is also based on partial critical paths. In order to incorporate performance variability of VMs, a certain amount of slack time defined by the *robustness type* is added to the PCP execution time which dictates the amount of execution time fluctuations a PCP can tolerate. From the set of all possible VM types, a feasible solution set FS for each PCP is created using the budget and time constraints. For each PCP and a given *robustness type,* appropriate VM type is selected from the FS based on certain resource selection policies. Each of the policies have three objectives; *robustness*, *time* and *cost*. The priorities among these objectives change for each of the policies: (a) robustness-cost-time (RCT) policy gives priority to robustness, followed by cost and time (b) robustness-time-cost (RTC) gives priority to robustness, followed by time and cost and (c) weighted policy allows users to define their own objective function using the three parameters (robustness, time and cost) and assign weights to each of them. Each of these policies sorts the feasible solution set based on the first parameter and the solutions with the same first parameter are sorted in the increasing order of second parameter. Solutions with the same first and second parameters are sorted in increasing order of third parameter. The best solution from this sorted list is picked and the corresponding VM type is mapped to the tasks of the PCP. Further, for fault-tolerance, the authors employed checkpointing at regular intervals. When a task fails, the algorithm resumes the task from the last checkpoint. The comparison of the proposed work with the robust scheduling algorithm is done only for the parameters which caters for performance variation and the acquisition delays of VMs. Further, the RCT and the RTC resource selection policies, in which the objective functions are specified distinctly, have been selected as the baseline algorithms.

5.3 Experimental Setup

The cloud service provider is assumed to provide five different types of VMs. The VM configurations and their processing capacity are based on the performance analysis of EC2 cloud offering [7] and are presented in Table 5. An ECU is the equivalent CPU power of 1.0-1.2 GHz Opteron or Xeon processor. VM pricing is based on the current pricing schemes of Amazon EC2 [29]. The average bandwidth between VMs is set to 20 MBps, which is the approximate average bandwidth offered in Amazon web services [25].

Billing interval is set to 10 minutes. Processing time of workflow tasks on different VMs were estimated on the basis of their processing capacity. Performance variation was modelled in accordance with the Schad et al. [12]. Similar to [26], the performance of each VM is reduced by at most 24 percent based on a normal distribution with mean 12 percent and standard deviation of 10 percent. In addition, a data transfer time variation of 19 percent [12] is modelled, based on a normal distribution with mean 9.5 percent and a standard deviation of 5 percent. Boot time of a VM is set to 97 seconds based on the results obtained by Mao and Murphy [43] for Amazon EC2 cloud.

In order to evaluate the proposed algorithm, a deadline needs to be defined for each workflow. If the deadline is generously relaxed, there is enough slack time to accommodate for the VM acquisition delay and the performance variation. Therefore, a comprehensive evaluation requires performance analysis on all possible deadlines: *Strict*, *Moderate* and *relaxed*. To this end, the deadlines were set using the rule as specified in Equation (13):

$$Deadline\ D\ = (1 + \mu) \times MET\_W, \tag{13}$$

Where, MET_W.is the minimum execution time of the workflow. μ is the deadline factor defined as follows.

$$For\ Strict\ Deadlines: \quad 0 \leq \mu\ < 1.5$$

$$For\ Moderate\ Deadlines: \quad 1.5 \leq \mu\ < 3$$

$$For\ Relaxed\ Deadlines: \quad 3 \leq \mu\ < 4.5$$

For the experiments, the value of μ is varied with a step length of 0.4.

5.4 Results and Analysis

A comprehensive evaluation of the proposed algorithm and its comparison with the baseline algorithms is performed, to identify its ability in meeting workflow deadline constraints at reduced costs. To this end, a cloud environment with performance variation and acquisition delays are simulated in accordance with the findings of Schad et al. [12] as discussed in Section 5.3 and experiments are conducted to analyse the algorithms in terms of meeting the deadlines, makespan and the cost incurred in executing the experimental workflows. Each experiment is executed 10 times and the mean of the results obtained for large workflows are reported.

5.4.1 Deadline Constraint Evaluation

To analyse the algorithms in terms of meeting the user defined deadlines, the percentage of deadlines met for each workflow with the different deadline factors are evaluated. The results are displayed in Table 6.

Strict deadlines. It can be observed from the results in Table 6 that *IC-PCP* fails to meet all the strict deadlines for all the experimental workflows. *RCT* displays a better performance with 52.5 percent hit rate for the LIGO workflow, 47.5 percent hit rate for Montage workflow, 40 percent hit rate for CyberShake workflow and 37.5 percent hit rate for Epigenomics workflow. *RTC* exhibits much better

TABLE 6
Percentage of Deadline Met for Each Workflow and Deadline Factor

Deadline Factor		MONTAGE	CYBERSHAKE	EPIGENOMICS	LIGO
	PCP	0	0	0	0
	RCT	47.5	40	37.5	52.5
STRICT	RTC	80	77.5	72.5	77.5
	JIT	88	84	80	84
	PCP	0	0	0	0
	RCT	52.5	47.5	52.5	55
MODERATE	RTC	100	100	100	100
	JIT	100	100	100	100
	PCP	0	0	0	0
	RCT	55	52.5	60	57.5
RELAXED	RTC	100	100	100	100
	JIT	100	100	100	100

performance with 80 percent of the deadline constraints met for Montage workflow, followed by CyberShake, LIGO and Epigenomics workflows with 77.5, 77.5 and 72.5 percent hit rate respectively. The proposed *JIT-C* outperforms the other three algorithms at strict deadlines with a hit rate of 88 percent for Montage workflow, 84 percent hit rate for CyberShake and LIGO workflows and 80 percent hit rate for Epigenomics workflow.

Moderate deadlines. Results obtained for moderate deadlines (Table 6) show that *IC-PCP* does not improve its performance and fails to meet any of the moderate deadline constraints. *RCT* slightly improves its performance as compared to its performance on strict deadlines with a hit rate of 55 percent for LIGO workflows, 52.5 percent for Montage & Epigenomics workflows, and 47.5 percent for CyberShake workflow. Both *RTC* and *JIT-C* are best performing algorithms at moderate deadlines with 100 percent hit rate.

Relaxed deadlines. As can be seen from the results in Table 6, IC-*PCP*, fails to meet all the relaxed deadlines and registers a 0 percent hit rate. *RCT* registers a marginal improvement with 60 percent hit rate for Epigenomics workflow, 57.5 percent hit rate for LIGO, 55 percent hit rate for Montage, and 52.5 percent hit rate for CyberShake workflows. Both *RTC* and *JIT-C* algorithms again exhibit 100 percent performance at relaxed deadlines.

It is observed, from the above, that IC-PCP performs poorly over all other three algorithms. This is because *IC-PCP* algorithm fails to capture the performance unpredictability and startup delays of VMs in a cloud environment. On the other hand, as discussed in Section 5.2, *RCT* and *RTC* policies are designed to bear a certain degree of uncertainty in VM performance, which is denoted by its *robustness type. RCT* policy gives highest priority to robustness followed by cost and makespan in order while *RTC* gives highest priority to robustness followed by makespan and cost in order. Evidently, *RTC* exhibits a better performance in comparison to *RCT* in terms of meeting the deadline constraints. The inability of *RCT* to improve its performance substantially with increase in deadline factors may be attributed to the following reasons. Given a deadline, a *robustness type* and processing time matrix of the VMs, *RCT* schedules PCPs on cheapest possible VM services such that the deadline constraints are met. Consequently, as deadline increases, *RCT* schedules PCPs on cheaper VMs which in the event of performance variation, exceeds the execution time of the PCP beyond the fluctuation limits it can tolerate, thus missing the deadlines.

Although, with moderate and relaxed deadlines, both *RTC* and *JIT*-C exhibit 100 percent hit rate however at strict deadline factors the performance of *JIT-C* is better than *RTC*. It may be noted that since *RTC* schedules the entire PCPs on VMs, therefore, at strict deadlines, performance variation of a VM not only adversely effects the execution time of all the tasks of the PCP running on it but also the execution start time of tasks of other PCPs. This cascading effect, at times, results in missing the strict deadlines. On the other hand, *JIT-C* algorithm schedules *individual tasks* just before they are ready for execution taking into account the performance variation of the predecessor tasks and hence is able to exhibit a better hit rate.

5.4.2 Makespan and Cost Evaluation

Since, it is intended that the algorithms should generate a cost effective schedule but not at the expense of a longer execution time, therefore for a holistic comparison, the average makespan and cost needs to be observed simultaneously. Fig. 9 shows the average execution costs (in $) and the average makespan (in seconds) for each workflow. The reference line above each deadline factor corresponds to its associated deadline value.

It can be seen that for all the workflows and all the deadline factors, *IC-PCP* generates cheapest schedules but it takes a longer execution time than the workflow's deadline and hence fails to meet any of the deadline constraints. Since the objective is to generate cheaper schedule while meeting the deadlines, therefore a cheaper schedule obtained at the cost of deadline constraint violation is not of any use. The comparison is therefore made among the other three algorithms that manage to meet the deadlines.

As may be seen in Fig. 9, among *RCT, RTC* and *JIT-C* algorithms, *RCT* generates the cheapest schedules with maximum makespan at all the deadline factors, however, it is able to register an average hit rate of only 50 percent. At strict deadline factors of 0 and 0.4, *JIT-C* generates the most expensive schedules with minimum makespan and registers a better hit rate in comparison to the other algorithms. This is because *JIT-C* takes appropriate scheduling decisions to adapt to the VM performance variation in order to limit

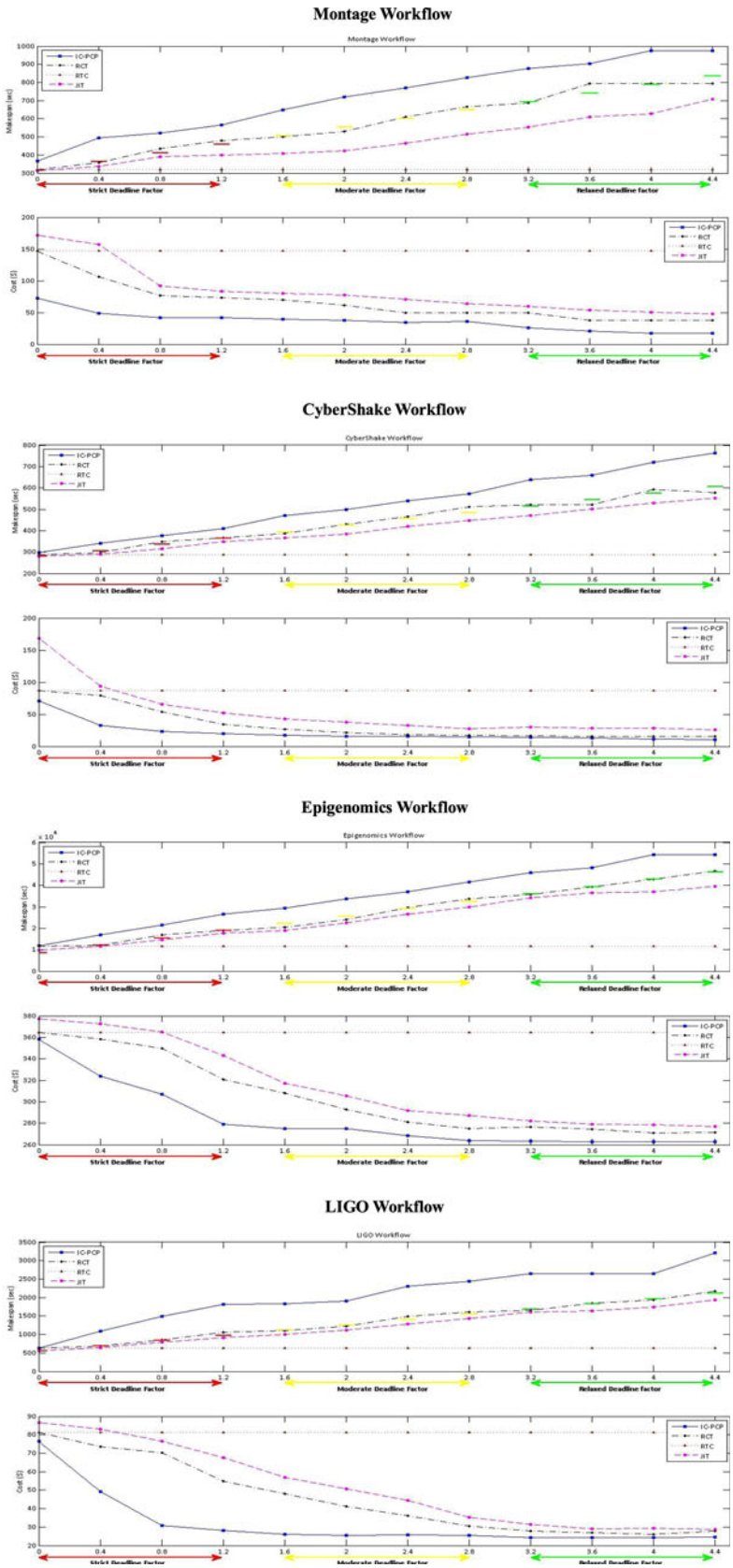


Fig. 9. Makespan and Cost of each Workflow for a given deadline factor.

deadline violations albeit at higher costs. It can be seen that for all the deadline factors other than 0 and 0.4, *RTC* generates the most expensive schedules with minimum makespan. On the contrary, as deadline factor increases, *JIT-C* takes advantage of the increased slack time and generates cheaper schedules which are able to meet the deadline constraints. Further, it is observed that on an average for all deadline factors, *JIT-C* algorithm has 34 percent lower cost

than *RTC* and 28 percent higher cost than *RCT*. *JIT-C*, generates 46 percent higher makespan than *RTC* and 16 percent lower makespan than *RCT*. It may therefore be concluded from the experimental results that *JIT-C* delivers better performance in terms of meeting the deadlines at reduced costs in comparison to the baseline algorithms. At strict deadlines, it is able to deliver highest hit rates for all the workflows, though at higher costs. However, as deadlines get relaxed, it is able to capitalize the increased slack time available so as to reduce the cost.

5.4.3 Computational Complexity

To compute the time complexity, suppose the workflow *W (T, E)* to be scheduled consists of n tasks and e edges. Also, let the maximum number of VM types offered by the cloud provider be m. Since *W* is a DAG, the maximum number of edges in *W* is $\frac{(n-1)(n-2)}{2} \cong O(n^2)$. First, we compute the time complexity for the overall actions of the different modules used in the algorithm (*Preprocessing, PlanandSchedule* and *CheapesttaskVMmap*) instead of entering into details. The algorithm *JIT-C* first establishes the *achievability* of the user defined deadline by evaluating the values of *MET, EST, EFT* and *MET_W* (lines 2-3) which involves the time complexities of $O(n.m)$, $O(n+e)$, $O(n)$ and $O(n)$ respectively. Once it is established that the user defined *deadline D* is achievable, the *Preprocessing* module (line 4) is called to combine pipeline tasks into a single task, which involves a time complexity of $O(n)$. Next, *MET, LFT* and *XET* are evaluated for the *preprocessed* workflow (line 5) with time complexities $O(n.m), O(n+e)$ and $O(n^2.m)$ respectively. The algorithm then schedules the entry tasks and enters into a monitor control loop, in which, it iteratively schedules the *to_be_scheduled tasks* of the workflow (lines 6-18). For each *to_be_scheduled task,* the algorithm first identifies its cheapest applicable VM type, *taskvmmap,* with the *CheapestVMmap* module which implies a time complexity of $O(n+m)$. Next, the *Planandchedule* module evaluates the applicability of the available *Active_VMs* to schedule the *to_be_scheduled task* in $O(n)$ time (since the number of active VMs is bounded by the number of tasks (n), which represents the case when each task is scheduled on a separate VM). If an appropriate *Active_VM* does not exist, a new VM of type *taskvmmap* (*vmmap)* is initiated to schedule the task. Subsequently, the *AST* and *XFT* of the scheduled task is updated in a constant time. Further, since each task may have atmost $(n-1)$ sucessors, the list of *to_be_scheduled tasks* is updated with time complexity $O(n)$. The above process is repeated once for each task of the workflow until all the tasks are scheduled. Accordingly, the complexity of the scheduling steps (6-18) of the *JIT-C* algorithm is $O((n+m)n)$. Thus, the overall time complexity of the proposed algorithm is $O(n^2.m + ((n+m)n)) = O(n^2.m)$. Since the number of VM types, offered by a cloud provider, is constant and small enough to be ignored the overall time complexity of the proposed algorithm is $O(n^2)$. Further deliberating on the internal steps and dependencies involved in the VM assignment procedure for a task t, it may be observed that for each of the *Active_VMs,* the *Planandschedule* module verifies that selecting a VM for scheduling t does not lead to violation of the LSTs of any of its children tasks. Since a task may have a maximum of (*n–1*) children tasks, therefore, the complexity obtained while considering all the dependencies is $O(n^3)$.

It may be noted that the computational complexity of the baseline algorithms, computed to be equal to $O(n^2)$ in [23] and [40], does not include the time complexity involved in identifying the partial critical paths. Thus, if the complexity of identifying the partial critical paths is taken into account, the time complexity of these algorithms adds up to $O(n^3)$, as has been computed in a previous work by Abrishami et al. [17]. Thus, the proposed model incurs same computational complexity as the baseline algorithms while producing much better results.

6 Related Work

As deliberated in [28], workflow scheduling on distributed resources is an NP-hard problem. Therefore, it warrants to apply heuristics/meta-heuristics based techniques for near-optimal or approximate solutions. Most of the existing work on workflow scheduling focusses on distributed environments such as grids and clusters [15], [34], [35], [36], [37], [38] and very few models are proposed for cloud based environments. This section surveys some relevant contributions to the field.

Malawski et al. proposed various dynamic and static algorithms for resource provisioning and scheduling workflow ensembles in cloud [30]. These algorithms aim to maximize the number of executed workflows while meeting the QoS constraints of deadline and budget. The proposed solutions acknowledge different delays involved in leasing VM resources from the IaaS cloud, such as VM acquisition and termination delays. Also, the issue of performance variation of VMs is addressed by assuming that a task's execution time may vary based on a uniform distribution. They, however, consider only a single VM type ignoring the heterogeneous nature of IaaS clouds.

Mao and Humphrey proposed a dynamic approach for scheduling workflow ensembles on clouds such that all workflows are finished within their deadlines at minimum cost [27]. They acknowledge different types of VMs, available at different prices, which can be leased dynamically in accordance with the needs. They proposed a set of heuristics such as task bundling and instance consolidation aimed at minimizing the execution cost of the workflow ensemble. The issues of acquisition delays and performance variation of VM instances were addressed by making dynamic scheduling decisions. Their approach however does not consider data transfer time between tasks which is of significant importance and affects both performance and the cost in scientific workflows especially for data intensive applications.

The algorithms presented by Mao and Humphrey [27] and Malawski [30] were designed for workflow ensembles and not for single workflow instances. More in line, with this work, are the algorithms proposed by Abrishami et al. [23] and Poola et al. [40] for scheduling a single workflow instance on an IaaS cloud. Both of these works are based on workflow's partial critical paths. The algorithm proposed in [23] calculates the latest finish time for each task, based on its estimated minimum execution time and the workflow's deadline. It then schedules all the tasks of a PCP onto a single VM instance which can finish all its tasks before their

latest finish time. [23] also considers the characteristic features of cloud such as VM heterogeneity, elastic provisioning and interval based pay-as-you-go billing model. However, it does not consider the performance variation and instance acquisition delays which may be encountered in a cloud environment. The authors in [40] proposed robust scheduling algorithm that handles performance variation and failures in a cloud environment. They also proposed resource allocation policies that schedules PCPs of a workflow on heterogeneous cloud resources while minimizing the makespan and the cost. The algorithm presented in [40] though considers all the characteristic features of cloud, however it caters only for a certain degree of performance variability.

Byun et al. [20] proposed the Partitioned Balanced Time Scheduling (PBTS) algorithm to estimate the minimum number of computing resources required at each time interval such that a workflow finishes its execution within the user specified finish time. Their algorithm also generates a task to resource mapping and is designed to run online. However, they do not consider the heterogeneous nature of computing resources and assumes only one VM type.

Some authors have used meta-heuristic techniques for scheduling workflows. Pandey et al. [42] proposed a particle swarm optimization (PSO) based heuristic to minimize the total cost of execution of a single workflow on a cloud while balancing the load on the available resources. Rodriguez and Buyya [26] developed a particle swarm optimization based algorithm to minimize the execution cost of a workflow while meeting the user defined deadline constraints. Yu and Buyya [31] used Genetic Algorithm for cost optimization under deadline constraint and execution time optimization under budget constraint. Chen and Zhang [32] proposed an Ant Colony Optimization algorithm with three QoS parameters: time, cost and reliability. Szabo at al. [41] proposed a multi-objective algorithm based on evolutionary approach, for execution of data-intensive scientific applications in a cloud environment such that the data transferred between tasks and the workflow execution time are minimized. Though, these methods exhibit good performance they usually are more time consuming than other heuristic based approaches.

Thus, other existing heuristic based algorithms for scheduling deadline constrained workflows in a cloud at reduced costs [20], [23], [30] [40] either fail to completely incorporate the basic characteristics of cloud computing (e.g. heterogeneous computing resources, VM performance variation and acquisition delays) or fail to incorporate the characteristics of scientific workflows (e.g. data transfer time between tasks) [27]. As a result, these solutions are either unable to meet the user defined deadline or generate costly schedules. The proposed work incorporates all the essential characteristic features of cloud and scientific workflows, and presents a just-in-time resource provisioning and scheduling algorithm for executing scientific workflows in a cloud environment that meets the user specified deadline with reduced cost.

7 Conclusion

Cloud computing environment offers tremendous opportunities and alternatives to execute large scale scientific workflows. Executing scientific applications in cloud involves making appropriate provisioning and scheduling decisions such that the overall execution cost is minimized while meeting a user defined deadline. Towards this, a dynamic cost-minimization and deadline constrained heuristic, *JIT-C*, for scheduling scientific applications in a cloud environment has been proposed in this work. In order to maintain low execution cost, resources are provisioned just before they are needed. The objective of meeting the deadline is achieved through continuous monitoring of the running tasks and dynamically making cost effective scheduling decisions for subsequent tasks such that the deadline constraint is not violated. The simulation experiments conducted on four well known workflows show that in comparison to the other state of the art heuristics, *IC-PCP*, *RCT* and *RTC*, the proposed algorithm displays the highest hit rate in meeting the deadline. Further, it exploits the slack time available with relaxed deadlines to produce cheaper schedules with lower execution costs. In comparison to the best performing baseline algorithm *RTC* for the similar purpose, the proposed algorithm *JIT-C* generate schedules with an average of 34 percent lower costs.

The proposed scheduling algorithm addresses three major issues of cloud platforms: VM performance variation, resource acquisition delays and heterogeneous nature of cloud resources. It has the potential to act as a good candidate for its incorporation in cloud resource management. It is proposed that in future, this work has the potential to include robustness against the task and the VM failures which may adversely affect the overall workflow execution time. Another future work may include the querying ability such as the effect on the cost by changing the deadline and revising it accordingly.

Acknowledgments

Authors would like to accord their sincere thanks to the anonymous reviewers for their useful suggestions resulting in the quality improvement of this paper. It is also to acknowledge UPE-II for the financial support to this work.

References

[1] E. Deelman, D. Gannon, M. Shields, and I. Taylor, "Workflows and e-Science: An overview of workflow system features and capabilities," *Future Gener. Comput. Syst.*, vol, 25, no. 5, pp. 528–540, 2009.

[2] E. Deelman, G. Singh, M. H. Su, J. Blythe, Y. Gil, C. Kesselman, and D. S. Katz, "Pegasus: A framework for mapping complex scientific workflows onto distributed systems," *Sci. Program.*, vol, 13, no. 3, pp. 219–237, 2005.

[3] M. Wieczorek, R. Prodan, and T. Fahringer, "Scheduling of scientific workflows in the ASKALON grid environment," *ACM SIGMOD Rec.*, vol, 34, no. 3, pp. 56–62, 2005.

[4] F. Berman, H. Casanova, A. Chien, K. Cooper, H. Dail, A. Dasgupta, and A. YarKhan, "New grid scheduling and rescheduling methods in the GrADS project," *Int. J. Parallel Program.*, vol. 33, no. 2/3, pp. 209–229, 2005.

[5] C. Hoffa, G. Mehta, T. Freeman, E. Deelman, K. Keahey, B. Berriman, and J. Good, "On the use of cloud computing for scientific workflows," in *Proc. IEEE 4th Int. Conf. eScience*, Dec. 2008, pp. 640–645.

[6] G. Juve, E. Deelman, K. Vahi, G. Mehta, B. Berriman, B. P. Berman, and P. Maechling, "Scientific workflow applications on Amazon EC2," in *Proc. 5th IEEE Int. Conf. E-Science Workshops*, Dec. 2009, pp. 59–66.

[7] S. Ostermann, A. Iosup, N. Yigitbasi, R. Prodan, T. Fahringer, and D. Epema, "A performance analysis of EC2 cloud computing services for scientific computing," in *Proc. 1st Int. Conf. Cloud Comput.*, 2010, pp. 115–131.

[8] E. Deelman, G. Singh, M. Livny, B. Berriman, and J. Good, "The cost of doing science on the cloud: The montage example," in *Proc. ACM/IEEE Conf. Supercomput.*, Nov. 2008, p. 50.
[9] G. Juve, E. Deelman, G. B. Berriman, B. P. Berman, and P. Maechling, "An evaluation of the cost and performance of scientific workflows on amazon ec2," *J. Grid Comput.*, vol. 10, no. 1, pp. 5–21, 2012.
[10] G. Juve and E. Deelman, "Scientific workflows in the cloud," in *Grids, Clouds and Virtualization*. London, U.K.: Springer, pp. 71–91, 2011.
[11] P. Mell and T. Grance , "The NIST definition of cloud computing, " NIST, Gaithersburg, MD, USA, Tech. Rep. SP 80-145, 2011.
[12] J. Schad, J. Dittrich, and J. A. Quiané-Ruiz, "Runtime measurements in the cloud: Observing, analyzing, and reducing variance," *Proc. VLDB Endowment*, vol. 3, no. 1–2, pp. 460–471, 2010.
[13] Amazon elastic compute cloud (Amazon EC2) [Online] Available: http://aws.amazon.com/ec2/, 2015.
[14] Amazon elastic block store (Amazon EBS) [Online] Available: http://aws.amazon.com/ebs/, 2015.
[15] J. Yu, R. Buyya, and C. K. Tham, "Cost-based scheduling of scientific workflow applications on utility grids," in *Proc. 1st Int. Conf. e-Sci. Grid Comput.*, Jul. 2005, pp. 140–147.
[16] G. R. Nudd, D. J. Kerbyson, E. Papaefstathiou, S. C. Perry, J. S. Harper, and D. V. Wilcox, "PACE—A toolset for the performance prediction of parallel and distributed systems," *Int. J. High Perform. Comput. Appl.*, vol. 14, no. 3, pp. 228–251, 2000.
[17] S. Abrishami, M. Naghibzadeh, and D. H. Epema, "Cost-driven scheduling of grid workflows using partial critical paths," *IEEE Trans. Parallel Distrib. Syst.*, vol. 23, no. 8, pp. 1400–1414, Aug. 2012.
[18] S. Jang, X. Wu, V. Taylor, G. Mehta, K. Vahi, and E. Deelman, "Using performance prediction to allocate grid resources," Texas A&M Univ., College Station, TX, USA, GriPhyN Tech. Rep. 2004-25, 2004.
[19] W. Smith, I. Foster, and V. Taylor, "Predicting application run times using historical information," in *Proc. Workshop Job Scheduling Strategies Parallel Process.*, Jan. 1998, pp. 122–142.
[20] E. K. Byun, Y. S. Kee, J. S. Kim, and S. Maeng, "Cost optimized provisioning of elastic resources for application workflows," *Future Gener. Comput. Syst.* vol. 27, no. 8, pp. 1011–1026, 2011.
[21] G. Juve, A. Chervenak, E. Deelman, S. Bharathi, G. Mehta, and K. Vahi, "Characterizing and profiling scientific workflows," *Future Gener. Comput. Syst.*, vol. 29, no. 3, pp. 682–692, 2013.
[22] S. Bharathi, A. Chervenak, E. Deelman, G. Mehta, M. H. Su, and K. Vahi, "Characterization of scientific workflows," in *Proc. 3rd Workshop Workflows Support Large-Scale Sci.*, Nov. 2008, pp. 1–10.
[23] S. Abrishami, M. Naghibzadeh, and D. H. Epema, "Deadline-constrained workflow scheduling algorithms for Infrastructure as a service clouds," *Future Gener. Comput. Syst.*, vol. 29, no. 1, pp. 158–169, 2013.
[24] Google Cloud Platform. [Online] Available: https://cloud.google.com/compute/, 2015.
[25] M. R. Palankar, A. Iamnitchi, M. Ripeanu, and S. Garfinkel, "Amazon S3 for science grids: A viable solution?" in *Proc. Int. Workshop Data-Aware Distrib. Comput.*, Jun. 2008, pp. 55–64.
[26] M. A. Rodriguez and R. Buyya, "Deadline based resource provisioning and scheduling algorithm for scientific workflows on clouds," *IEEE Trans. Cloud Comput.*, vol. 2, no. 2, pp. 222–235, Apr.–Jun. 2014.
[27] M. Mao and M. Humphrey, "Auto-scaling to minimize cost and meet application deadlines in cloud workflows," in *Proc. Int. Conf. High Perform. Comput., Netw., Storage Anal.*, Nov. 2011, p. 49.
[28] D. S. Johnson and M. Garey, *Computers and Intractability: A Guide to the Theory of NP-Completeness*. San Francisco, CA, USA: Freeman, 1979.
[29] Amazon EC2 Pricing [Online] Available: https://aws.amazon.com/ec2/pricing/, 2015.
[30] M. Malawski, G. Juve, E. Deelman, and J. Nabrzyski, "Cost-and deadline-constrained provisioning for scientific workflow ensembles in iaas clouds," in *Proc. Int. Conf. High Perform. Comput., Netw., Storage Anal.*, Nov. 2012, p. 22.
[31] J. Yu and R. Buyya, "Scheduling scientific workflow applications with deadline and budget constraints using genetic algorithms," *Sci. Program.*, vol. 14, no. 3, pp. 217–230, 2006.
[32] W. N. Chen and J. Zhang, "An ant colony optimization approach to a grid workflow scheduling problem with various QoS requirements," *IEEE Trans. Syst., Man, Cybern., C: Appl. Rev.*, vol. 39, no. 1, pp. 29–43, Jan. 2009.
[33] Workflow Generator [Online] Avaliable: https://confluence.pegasus.isi.edu/display/pegasus/WorkflowGenerator, 2015.
[34] Y. Yuan, X. Li, Q. Wang, and X. Zhu, "Deadline division-based heuristic for cost optimization in workflow scheduling," *Inf. Sci.*, vol. 179, no. 15, pp. 2562–2575, 2009.
[35] R. Sakellariou, H. Zhao, E. Tsiakkouri, and M. D. Dikaiakos, "Scheduling workflows with budget constraints," in *Proc. CoreGRID Integr. Workshop Integr. Res. GRID Comput.*, 2007, pp. 189–202.
[36] R. Prodan and M. Wieczorek, "Bi-criteria scheduling of scientific grid workflows," *IEEE Trans. Autom. Sci. Eng.*, vol. 7, no. 2, pp. 364–376, Apr. 2010.
[37] R. Duan, R. Prodan, and T. Fahringer, "Performance and cost optimization for multiple large-scale grid workflow applications," in *Proc. ACM/IEEE Conf. Supercomput.*, Nov. 2007, pp. 1–12.
[38] A. Afzal, J. Darlington, and A. McGough., "Qos-constrained stochastic workflow scheduling in enterprise and scientific grids," in *Proc. 7th IEEE/ACM Int. Conf. Grid Comput.*, Sep. 2006, pp. 1–8.
[39] A. Saifullah, D. Ferry, C. Lu, and C. Gill, "Real-time scheduling of parallel tasks under a general dag model," 2012.
[40] D. Poola, S. K. Garg, R. Buyya, Y. Yang, and K. Ramamohanarao, "Robust scheduling of scientific workflows with deadline and budget constraints in clouds," in *Proc. IEEE 28th Int. Conf. Adv. Inf. Netw. Appl.*, May 2014, pp. 858–865.
[41] C. Szabo, Q. Z. Sheng, T. Kroeger, Y. Zhang, and J. Yu, "Science in the cloud: Allocation and execution of data-intensive scientific workflows," *J. Grid Comput.*, vol. 12, no. 2, pp. 245–264, 2014.
[42] S. Pandey, L. Wu, S. M. Guru, and R. Buyya, "A particle swarm optimization-based heuristic for scheduling workflow applications in cloud computing environments," in *Proc. 24th IEEE Int. Conf. Adv. Inf. Netw. Appl.*, Apr. 2010, pp. 400–407.
[43] M. Mao and M. Humphrey, "A performance study on the VM startup time in the cloud," in *Proc. IEEE 5th Int. Conf. Cloud Comput.*, Jun. 2012, pp. 423–430.

Jyoti Sahni received the MTech degree in computer science from Jawaharlal Nehru University, New Delhi, India in 2013. She is currently working toward the PhD degree in the School of Computer and System Sciences, Jawaharlal Nehru University, New Delhi, India. Her research interests include resource management in distributed systems and cloud computing.

Deo Prakash Vidyarthi is a professor in the School of Computer & Systems Sciences, Jawaharlal Nehru University, New Delhi. He has published around 65 research papers in various peer reviewed International Journals and Transactions (including IEEE, Elsevier, Springer, Wiley, World Scientific, etc.) and around 35 research papers in proceedings of various peer-reviewed conferences in India and abroad. He has two books (research monograph) to his credit. One entitled *Technologies and Protocols for the Future Internet Design: Reinventing the Web* published by IGI-Global (USA) released in February 2012, and another entitled *Scheduling in Distributed Computing Systems: Design, Analysis and Models* published by Springer, USA released in 2009. He also has contributed chapters in many edited books. He is in the editorial board and in the reviewer's panel of many International Journals. His research interest includes parallel and distributed system, grid and cloud computing, mobile computing, and evolutionary computing. He is a member of the IEEE, senior member of the International Association of Computer Science and Information Technology (IACSIT), Singapore, International Society of Research in Science and Technology (ISRST), and International Association of Engineers.