General Framework for Task Scheduling and Resource Provisioning in Cloud Computing Systems

Xiaomin Zhu*, Yabing Zha*, Ling Liu†, and Peng Jiao*
*College of Information Systems and Management, National University of Defense Technology
Changsha, China 410073
Email: xmzhu@nudt.edu.cn, zhayabing@139.com, crocus201@163.com
†College of Computing, Georgia Institute of Technology
266 Ferst Drive, Atlanta, GA 30332-0765, USA
E-mail: lingliu@cc.gatech.edu

***Abstract*—Clouds have become an important platform to deliver services for various applications. Task scheduling and resource provisioning are key components to improve system performance under provisioned resources and satisfy users' demands for quality of service (QoS). To address the diversity of cloud services and applications, much of recent research and development efforts have been engaged in designing and implementing scheduling strategies and algorithms for specific tasks, such as dependent or independent tasks, fault-tolerant tasks with real-time deadlines or energy-efficient tasks. However, these task scheduling and resource provisioning schemes, though optimized with specific objectives, suffer from several inherent problems in cloud execution environments. In this paper, we propose a general framework for task scheduling and resource provisioning in cloud computing systems with dynamic customizability. By utilizing software engineering framework as the design guideline, we incorporate multiple scheduling objectives and multiple types of tasks to be processed under varied resource constraints to enable cloud applications to dynamically select and assemble scheduling strategies and algorithms according to different runtime QoS requirements. We illustrate the flexibility and customizability of our framework through two example scheduling algorithms: *EASU* and *RAS*. We validate the effectiveness of our proposed framework through experimental evaluation of the effectiveness of our proposed algorithms using both simulation and in real cloud platforms.**

I. Introduction

Cloud computing has become an enabling paradigm for on-demand provisioning of computing resources. It achieves scalability, cost-efficiency, and high resource utilization by meeting dynamic and diverse workload requirements of applications through server consolidation by virtual machines (VMs), and application consolidation by multi-tenancy and pay-as-you-go utility computing model [1].

Cloud providers are motivated to improve resource utilization for high throughput and high profit. However, mechanisms for achieving high resource utilization may result in unacceptable response time for some users' requests, inevitably hurting the quality of service in terms of request latency. Thus, cloud providers should make great effort to use possibly minimal resources to accommodate as many requests as possible, and at the same time, guarantee the quality of service for all users. Hence, task scheduling and resource provisioning become critical for clouds to achieve high efficiency in both resource utilization and quality of service.

As the number of applications being deployed in the clouds increases, the diversity of these applications grows rapidly, ranging from business, government to various science and engineering fields. These applications not only vary in task types but also have distinct performance objectives. For example, an application of genome mapping in bioinformatics includes many events inferred from genetic sequence, which can be formulated into multiple tasks with logical sequences [2]. We call this type of tasks dependent tasks. In contrast, the tasks such as Web requests are typically independent tasks. In addition, many weather forecasting and medical simulations applications have real-time deadlines, which, once broken, make the result useless. Furthermore, most of the long running tasks such as Web crawling systems typically require to be fault tolerant to avoid high cost of roll back operations in the presence of failure, although this type of tasks is less sensitive to latency (the completion time) compared with those with real-time deadlines. By guaranteeing fault-tolerance, the tasks are allocated in a manner such that the impact of a failure on system performance is minimized. Similarity, one can also select energy conservation as the objective of performance optimization, where the resources are used in a manner to ensure that the total energy required to execute a given workload is minimized.

To address the diversity of cloud services and applications, much of recent research and development efforts have been engaged in designing and implementing scheduling strategies and algorithms for specific tasks, such as dependent or independent tasks, fault-tolerant tasks with real-time deadlines or energy efficient tasks. However, these task scheduling and resource provisioning schemes, though optimized with specific objectives, suffer from several inherent problems. First, the optimization goals, once set at the design time, will be statically built into the task scheduling and resource provisioning algorithm and implementation as monolithic system component, thus lacking flexibility and adaptability in the presence of changing workload characterization, changing resource provisioning and changing cloud execution environment. Second, many task scheduling and resource provisioning strategies and algorithms, though designed with varied different optimiza-

tion objectives, often share some common functional components and employ similar software engineering framework for implementation. However, adding new scheduling capability needs to be done for each scheduling algorithm one at a time, which is not only tedious but also expensive and error prone. In this paper, we propose a general framework for task scheduling and resource provisioning in cloud computing systems with dynamic customizability. By utilizing software engineering framework as the design guideline, we incorporate multiple scheduling objectives and multiple types of tasks to be processed under varied resource constraints to enable cloud applications to dynamically select and assemble scheduling strategies and algorithms according to different runtime QoS requirements. We illustrate the flexibility and customizability of our framework through two example scheduling algorithms: *EASU* and *RAS*. We validate the effectiveness of our proposed framework through experimental evaluation of the effectiveness of our proposed algorithms using both simulation and in real cloud platforms.

To the best of our knowledge, this work is the first one towards the design and development of a general framework for task scheduling and resource provisioning in cloud computing environments.

This paper makes the following contributions. First, we propose a general framework for task scheduling and resource provisioning in the clouds to enable scheduling algorithms to flexibly select objectives. Second, we construct a novel scheduling structure to support flexible and adaptive resource provisioning and task scheduling framework. Third but not the least, we provide two example task scheduling schemes, each with different objective and different task type, to showcase the efficiency and effectiveness of our framework in terms of software reuse and maintenance.

The rest of this paper is organized as follows. Section II reviews related work. Section III presents the design overview of our framework. System models for two examples are given in Section IV. Section V introduces some relevant algorithms. Section VI discusses the framework implementation in detail, and presents experimental results and performance analysis. Section VII concludes this paper.

II. Related Work

Many task scheduling and resource provisioning methods have been proposed in the context of cloud computing. In this section, we review a selection of related work in the context of different scheduling objectives and different task types.

In the context of energy-efficient scheduling, Cardosa et al. investigated the problem of energy conservation in clouds, and proposed an incremental time balancing algorithm that explicitly made trade-off between energy consumption and job run-time [3]. Deng et al. proposed an online control algorithm SmartDPSS for power supply system in a cloud data center based on the two-timescale Lyapunov optimization techniques to deliver reliable energy with arbitrary demand over time [4]. Singh et al. studied a cluster-based heuristic FastCEED, to optimize the energy consumption using mixed integer programming and duplication for communication intensive applications [5]. Taking reliability as objective, many investigations employ different techniques. Wang et al. extended the conventional PB model to incorporate the features of clouds and comprehensively analyzed the constraints while scheduling and considered the resource elastic provisioning [6]. Plankensteiner and Prodan studied the fault-tolerant problem in clouds and proposed a heuristic that combines the task replication and task resubmission to increase the percentage of workflows that finish within soft deadlines [7]. Zhou et al. studied the problem on enhancing the reliability of cloud service using checkpoint technique where the identical parts of all virtual machines that provide the same service are checkpointed once as the service checkpoint image to reduce the resource consumption [8]. Some work concentrates on multiple objectives. For example, Chen et al., considered both the energy-conservation and system uncertainty as scheduling objectives and proposed a scheduling algorithm PRS using proactive and reactive scheduling methods to control uncertainty. Duan et al. focused on the execution time and economic cost as two objectives and formulated the scheduling problem as a sequential cooperative game [9].

From the task type point of view, there also exist a lot of investigations for task scheduling and resource provisioning. For instance, Rodriguez and Buyya suggested a resource provisioning and scheduling strategy for scientific workflows (dependent tasks) on IaaS cloud, in which the particle swarm optimization technique was employed [10]. Mao and Humphrey designed, implemented and evaluated two auto-scaling solutions to minimizing job turnaround time within budget constraints for dependent tasks in clouds [11]. In contrast, Xiao et al. concentrated on independent tasks and studied the dynamic request redirection and resource provisioning for cloud-based video services [12]. Additionally, some researchers pay attention to real-time task scheduling. For example, Abrishami et al. studied a two-phase scheduling algorithm named PCP for the cloud environment with the goal of minimizing workflow execution cost and ensuring predefined deadlines [13]. Hosseinimotlagh et al. proposed a cooperative two-tier task scheduling approach to benefit both cloud providers and their users in a real-time nature [14]. Non-real-time task scheduling are also studied in clouds. For instance, Bittencourt studied the scheduling issue in hybrid cloud to minimize the makespan of workflows without real-time requirement [15].

Recent application consolidation efforts develop container based technology for resource management, such as CloudStack [1], OpenStack [2], YARN [16], Apollo [17], etc. YARN is one of the most representative pieces of work. It manages CPU and memory resources as containers and allows application task manager to dynamically request containers. However, this container-based optimization is limited to the task level, CPU and memory resource consolidation. These solutions lack

[1]https://cloudstack.apache.org/
[2]https://www.openstack.org/

of support for task scheduling and resource provisioning by incorporating different performance optimization objectives, such as fault-tolerance, energy efficiency, and different types of tasks, such as dependent tasks, tasks with real-time deadlines.

III. Design Overview

To design and provide scheduling management framework for engineering implementation in IaaS Clouds, in this section, we introduce three important aspects in cloud computing resource management, i.e., scheduling management objective (SMO), tasks type, and resource characterization.

A. Scheduling Management Objectives

In the context of cloud computing systems, the scheduling resource management is to allocate tasks to a set of computing resources and at the same time, provision suitable computing resources to run these tasks. The following objectives are becoming the focus for academic and industrial researchers.

1) Service Level Agreement: Service Level Agreement (SLA in short) is a service contract between cloud providers (data centers) and users where a service is formally defined [18]. Specifically, the service includes response time, processing accuracy, cost and so on.

2) Energy Conservation: Cloud data centers consume tremendous amount of energy which accounts for a significant portion of worldwide energy usage [19]. High energy consumption not only results in high expenses but also affects environment and system reliability. Thereby, much attention has been drawn towards energy use minimization, whereby task scheduling and resource provisioning are made by minimizing the amount of energy needed to execute a given workload.

3) Reliability: Reliability is a basis to guarantee providing high quality services for users in clouds. It is reported that there will be one failure of servers per day for a system composed of 10 thousand super reliable servers [20]. Therefore, delivering fault-tolerant capability in clouds becomes a critical issue to enhance the system's reliability, especially for those applications with high reliability requirements.

4) Uncertainty: Uncertainty control is an important issue in clouds. Investigating how to measure and control uncertainty is capable of efficiently improving the scheduling precision. For example, the performances of virtual machines are varied during running. If uncertainties are not effectively handled, the scheduling decision maybe not work or even yield negative impact for task running.

B. Task Types

There are a variety of tasks that are able to run on cloud computing context. Especially, for a public cloud, many kinds of tasks may be processed simultaneously. Generally, tasks can be classified into four categories.

1) Independent and Dependent Tasks: Independent tasks refer to the tasks among which there are no data and control dependencies. In contrast, for dependent tasks, there is a partial order between tasks or control dependencies. The tasks must be executed in a certain order. Commonly, dependent tasks can be modelled by a Directed Acyclic Graph (DAG).

2) Real-time and Non-real-time Tasks: Real-time tasks are the tasks having deadlines, which means the tasks should be finished within the a given timing constraint. On the contrary, non-real-time tasks do not have deadlines, but it also pursues to have quick response time.

3) Periodic and Aperiodic Tasks: For periodic tasks, the interval time between two adjacent tasks' arrival time is a constant (i.e., periodic). So, once knowing the arrival time of the first task, the arrival time of following tasks can be calculated. However, the aperiodic tasks are the tasks whose arrival times are not known a priori.

4) Priority and Non-priority Tasks: Priority tasks are the tasks having priorities. The priorities can be given by 1) the users when submitting their tasks; 2) the negotiation from both users and resource providers; or 3) the calculation of systems based on some task features such as the tightness of tasks deadlines or the payment from users. As far as non-priority tasks are concerned, they have no specified priorities.

For a task, it can belong to some or all the combinations from the aforementioned four types. For example, a task can be a real-time, independent, aperiodic task with priority.

C. Resource Characteristics

Clouds have two important features that should be addressed in task scheduling and resource provisioning.

1) Virtualization: Virtualization is commonly used in cloud environments to provide flexible and scalable system services [21]. By employing virtualization technology, a single physical host can run multiple virtual machines (VMs) simultaneously. Consequently, VMs become basic computational instances rather than physical hosts, thus tasks are allocated to VMs instead of directly to physical hosts.

2) Dynamic Resource Provisioning: The distinct feature from other computing environments is that the resource provisioning of clouds is in a "pay-as-you-go" manner [22], which means resource provided by clouds is elastic according to the users' demand. Specifically, a cloud can be scaled up to satisfy the increased resource requests and scaled down to improve the system's resource utilization when the demand is reduced.

Fig. 1 illustrates the design overview in our scheduling framework. Different assembles of scheduling management objectives, tasks types and resource characteristics will employ specific algorithms, which from an algorithm library. For example, in Fig.1, the scheduling management objectives are SLA and reliability; task type is independent, real-time, aperiodic and having priority, combining the cloud features - virtualization and dynamic provisioning, Algorithm 1 should be specifically designed and implemented. It is worth noting that diverse task scheduling and resource provisioning algorithms can be designed and added to the algorithm library to deal with different tasks with given objectives.

IV. System Model

To support any kinds of task scheduling and resource provisioning, we design a general scheduling architecture that enables to handel different scheduling management objectives

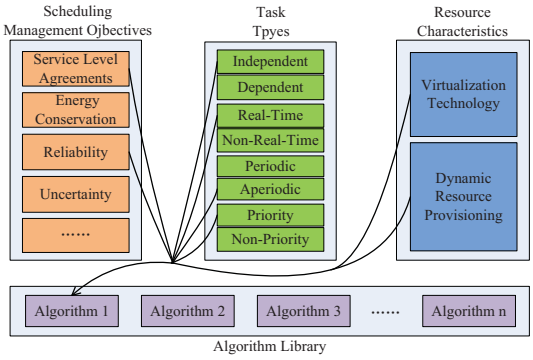

Fig. 1. Design Overview of Scheduling Framework.

and task types. Based on our proposed scheduling architecture, we present two scheduling models for some combinations in terms of SMO, task type and resource feature.

A. Scheduling Architecture

Fig. 2 illustrates the scheduling architecture for task scheduling and resource provisioning. It consists of *Task Analyzer*, *SMO Analyzer*, *Objective Pool*, *Task Scheduler*, *Resource Monitor*, *Resource Allocator*, *Algorithm Library*, and *Resource Pool*. When a new task arrives, the steps of task scheduling and resource provisioning are as follows:

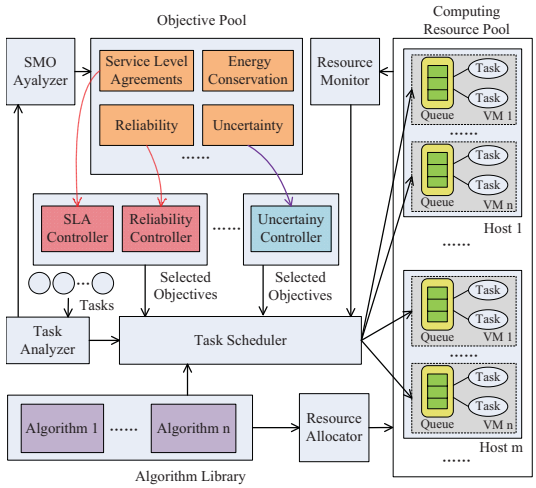

Fig. 2. Scheduling Architecture.

Step 1. The *Task Analyzer* firstly gets tasks' attributes such as arrival time, estimated execution time, deadlines and task types. Then it sends these information to *SMO Analyzer* and *Task Scheduler*.

Step 2. The *SMO Analyzer* decides which objectives should be selected. The SMO decision can derive from three aspects, i.e., task nature, resource state, and system providers. For instance, if some tasks have high reliability requirements (e.g., patient monitoring), the reliability should be added into the selected objectives; otherwise for some Web surfing request tasks, the reliability may not be included due to having extra cost. Another example, in emergency such as earthquake data processing, energy conservation can be put into the second position whereas quick response becomes dominated, so the energy reduction may not be incorporated into SMOs.

Step 3. Based on the analysis from *SMO Analyzer*, the selected objectives can be produced. Obviously, as shown in Fig. 1, there are multiple combinations in terms of objectives. Then these objectives will be sent to *Task Scheduler*.

Step 4. The *Task Scheduler* calls specific algorithm from algorithm library based on the task nature, selected objectives and current resource information from *Resource Monitor*, and then allocates tasks to a VM for execution. At the same time, the *Resource Monitor* constantly collects the allocated task state information and reports them to *Task Scheduler*. If some tasks cannot be finished as expected, correction mechanisms will be used to handel them.

Step 5. The *Resource Allocator* works in two cases. 1) When a task cannot be finished within expected time using current active hosts, the *Resource Allocator* will create a VM to accommodate this task. Creating VMs can be realized by starting a host and then put a VM on it or consolidating VMs and then add a new VM. 2) If the system is in light load, *Resource Allocator* will make resource consolidation and then shuts down some hosts or leaves them in sleep mode.

B. Scheduling Model

To support our flexible scheduling framework, in this subsection, we present two examples by combining different scheduling management objectives and task types. 1) Example 1 considers energy conservation and uncertainty as objectives for real-time, independent, aperiodic tasks without priorities, whereas 2) Example 2 takes reliability as its objective for real-time, dependent, aperiodic tasks without priorities.

1) General Scheduling Model: In our framework, some general scheduling models can be obtained to make them reuse in any case. For example, the virtualized cloud computing resource can be modeled as a general one as follows:

We consider a virtualized cloud that consists of a set $H = \{h_1, h_2, \cdots\}$ of physical computing hosts. The active host set is modeled by H_a with n elements, $H_a \subseteq H$. For a given host h_k, its processing capability p_k is characterized by its CPU performance in Million Instructions Per Second (MIPS). For each host $h_k \in H$, it contains a set $V_k = \{v_{1k}, v_{2k}, \cdots v_{|V_k|k}\}$ of virtual machines (VMs) and each VM $v_{jk} \in V_k$ has the processing capability p_{jk} that is subject to $\sum_{j=1}^{|V_k|} p_{jk} \leq p_k$. The ready time of v_{jk} is denoted by r_{jk}.

In addition, some task attributes can be modeled as a general one no matter what type of this task is. For example, each task has arrival time regardless of its task type. Thereby, we have the following main general task model.

We consider a set $T = \{t_1, t_2, \cdots\}$ of tasks. For a given task t_i, it can be denoted as $t_i = \{a_i, s_i, d_i, p_i\}$, where a_i, s_i, d_i and p_i are the arrival time, task size, deadline, and priority of task t_i, respectively. If t_i is a non-real-time task, d_i is set

to be $+\infty$ and if there is no priority for t_i, p_i is set to be 0. Let s_{ijk} be the start time of task t_i on VM v_{jk}. Similarly, f_{ijk} represents the finish time of task t_i on v_{jk}. We let e_{ijk} be the execution time of task t_i on VM v_{jk}. Besides, x_{ijk} is used to reflect task mapping on VMs in a virtualized cloud, where x_{ijk} is "1" if task t_i is allocated to VM v_{jk} at host h_k and is "0", otherwise. Furthermore, We use z_{ijk} to denote whether task t_i has been successfully finished.

2) Specific Scheduling Model: To realize given scheduling objectives for a certain kind of tasks, some specific scheduling models are required. For instance, Example 1 is for independent tasks, whereas Example 2 is for dependent tasks, so the task model regarding task dependency is surely different. It is enough for Example 1 to use the general task model as shown above. However, for Example 2, extra task models should be added to represent tasks' features. Fig. 3 shows the constitution of general scheduling model and special scheduling model.

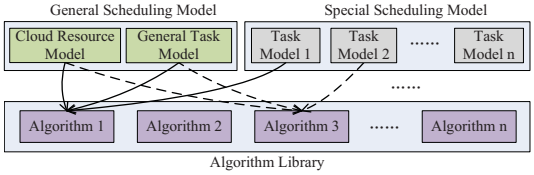

Fig. 3. Constitution of Scheduling Models.

For the special task model in Example 2, a set of dependent tasks can be modeled as $G = \{T, E\}$, where $T = \{t_1, t_2 \cdots\}$ is the same to that in general task model. A set of directed edges E represents dependencies among tasks. $e_{ij} = (t_i, t_j)$ indicates that task t_j depends on the results generated from task t_i for its execution. Task t_i is an immediate predecessor of t_j and t_j is an immediate successor of t_i. For a given task t_i, we use $P(t_i)$ and $S(t_i)$ to denote its immediate predecessor set and immediate successor set, respectively. $P(t_i) = \emptyset$ if t_i has no predecessors, and $S(t_i) = \emptyset$ if t_i has no successors.

Apart from task models, specific models for different objectives should be provided. Now, we present unique models in Example 1 and Example 2 as follows:

Example 1: Regarding the uncertainty, we consider in our study the CPU performance p_{jk} of VM vm_{jk} varies over the time and the task size s_i cannot be accurately measured. Hence, the start time, execution time, finish time of task t_i are uncertain too. $\widetilde{p}_{jk}$, $\widetilde{s}_i$, $\widetilde{st}_{ijk}$, $\widetilde{et}_{ijk}$, $\widetilde{ft}_{ijk}$, and $\widetilde{c}_{jk}$ are used to denote their uncertain parameters. We employ the interval number [23] to describe parameters as follows: $\widetilde{s}_i = [s_i^-, s_i^+]$ and $\widetilde{p}_{jk} = [p_{jk}^-, p_{jk}^+]$, where $s_i^-, s_i^+, p_{jk}^-, p_{jk}^+ > 0$. The uncertain execution time $\widetilde{et}_{ijk}$ can be calculated as [24]:

$$\begin{aligned}\widetilde{et}_{ijk} =& \widetilde{s}_i \oslash \widetilde{p}_{jk} = [s_i^-, s_i^+] \oslash [p_{jk}^-, p_{jk}^+] \\ =& [s_i^-, s_i^+] \otimes [s/p_{jk}^-, s/p_{jk}^+] \\ =& [s_i^-/p_{jk}^+, s_i^+/p_{jk}^-].\end{aligned} \tag{1}$$

Then, the uncertain finish time $\widetilde{ft}_{ijk}$ can be calculated as:

$$\widetilde{ft}_{ijk} = \widetilde{st}_{ijk} \oplus \widetilde{et}_{ijk}. \tag{2}$$

Since we consider real-time task scheduling, if a task t_i's real finish time is less or equal to its deadline, which is successful finish for this task. z_{ijk} can be determined as below:

$$z_{ijk} = \begin{cases} 1, & \text{if } ((ft_{ijk} \le d_i) \text{ and } (x_{ijk} = 1)), \\ 0, & \text{otherwise.} \end{cases} \tag{3}$$

So the count of successful finish tasks should be maximized:

$$\max \sum_{i=1}^{|T|} \sum_{j=1}^{|H_a|} \sum_{k=1}^{|V_k|} \frac{z_{ijk}}{|T|} | \sum_{j=1}^{|H_a|} \sum_{k=1}^{|V_k|} z_{ijk} \le 1, \forall t_i \in T. \tag{4}$$

Since we also take the energy conservation as an objective in this example, the total energy consumption should be minimized while scheduling. Thus, we have:

$$\min \sum_{j=1}^{|H_a|} \int_{st}^{ft} (k \cdot p_j \cdot y_j^t + (1-k) \cdot p_j \cdot u(t)) dt, \tag{5}$$

where $y_j^t \in \{1, 0\}$ represents whether host h_j is active or not at time instant t; k is the fraction of energy consumption rate consumed by idle hosts; p_j is the host overall power consumption; $u(t)$ is the CPU utilization of host h_j at time t.

Example 2: The reliability is considered as scheduling management objective in this example, we employ the primary/backup (PB in short) model [25] to guarantee system reliability, which is fit to tolerate one host failure at one time instant. With the PB model, each task t_i has a backup t_i^B. If a fault is occurred on the host on which t_i has been assigned before it finishes, t_i^B will be executed. Since the probability that two hosts fail simultaneously is small, we assume the backup copies can be successfully finished if their primary copies are on the failed host like that in [26]. All the variables or parameters with superscript "B" are relevant to t_i^B.

The backup t_i^B can have two execution modes, i.e., passive and active. We use $s(t_i^B)$ to denote its execution mode that can be determined by the following formula:

$$s(t_i^B) = \begin{cases} passive & \text{if } d_i - f_i \ge et_{ijk}^B, \\ active & \text{otherwise.} \end{cases} \tag{6}$$

where f_i is the expected finish time of t_i.

Thereby, the real execution time ret_{ijk}^B of t_i^B can be:

$$ret_{ijk}^B = \begin{cases} et_{ijk}^B & \text{if } z_{ijk} = 0, \\ (0, et_{ijk}^B] & \text{if } z_{ijk} = 1 \text{ and } s(t_i^B) = active, \\ 0 & \text{if } z_{ijk} = 1 \text{ and } s(t_i^B) = passive. \end{cases} \tag{7}$$

The total real execution time of backups should be minimized to save more time slots. Hence, we have the objective when making the reliability-aware scheduling.

$$\min \sum_{i=1}^{|T|} \sum_{j=1}^{|H_a|} \sum_{k=1}^{|V_k|} ret_{ijk}^B \cdot x_{ijk}^B. \tag{8}$$

Since the dependent tasks are considered in this example, we use e_{ij}^{XY} to denote the edge between t_i^X and t_j^Y where t_i^X and t_j^Y can be either a primary or a backup. For each edge

e_{ij}^{XY}, there is an associated data transfer time tt_{ij}^{XY} that is the amount of time needed by t_j^Y from $v(t_i^X)$. If two dependent tasks t_i^X and t_j^Y are on the same host, $tt_{ij}^{XY} = 0$. In addition, let dv_{ij} be the transfer data volume between task t_i and t_j. Let $ts\left(h(t_i^X), h(t_j^Y)\right)$ denote the transfer speed between $h(t_i^X)$ and $h(t_j^Y)$. Subsequently, if $h(t_i^X) \neq h(t_j^Y)$, we have:

$$tt_{ij}^{XY} = \frac{dv_{ij}}{ts\left(h(t_i^X), h(t_j^Y)\right)}. \tag{9}$$

The actual start time s_j^Y of task t_j^Y is the time at which the task is scheduled for execution. Task t_j^Y is able to be placed between est_j^Y and lft_j^Y if there exist slack time slots that can accommodate t_j^Y. One of the goals of our scheduling is to find suitable start time of tasks to process as many tasks as possible in case of guaranteeing fault-tolerance, so as to achieve high overall throughput.

Also, the system reliability can be quantitatively measured by reliability cost that can be defined as [26]:

$$rc = \sum_{i=1}^{|T|} \sum_{j=1}^{|H_a|} \sum_{k=1}^{|V_k|} \lambda_j \cdot x_{ijk} \cdot et_{ijk}. \tag{10}$$

So by using PB model, the total reliability to execute primaries and backups can be calculated as:

$$\begin{aligned} rc =& rc(T) + rc(T^B) \\ =& \sum_{k=1}^{|H_a|} \sum_{j=1}^{|V_k|} \sum_{i=1}^{|T|} \lambda_j \cdot (x_{ijk} \cdot ec_{ijk} + x_{ijk}^B \cdot z_{ijk}^B \cdot ret_{ijk}^B). \end{aligned} \tag{11}$$

Consequently, reliability cost can be considered to enhance the system reliability.

V. Algorithm Design

In this section, we present two task scheduling algorithms *EASU* (Energy-Aware Scheduling under Uncertainty) and *RAS* (Reliability-Aware Scheduling) for Example 1 and Example 2, respectively. They can be added to the *Algorithm Library* in our framework.

A. EASU Algorithm

Before introducing *EASU*, we define two queues - waiting queue WQ and urgent queue UQ to handel uncertainty. If a task in WQ becomes urgent task, it will be put into UQ. In this paper, we define an urgent task is a task whose laxity L_i [24] is less than or equal to a given threshold L_i that is the time for turning on a host and create a VM on it. Algorithm 1 is the pseudocode of *EASU*.

In our *EASU*, we employ the interval number to measure the uncertainty. ft_{ijk}^- and ft_{ijk}^+ are the estimated minimal finish time and estimated maximal finish time, respectively. The minimal ft_{ijk}^-, i.e., ft_{MIN} and maximal f_{ijk}^+, i.e., ft_{MAX} are recorded by checking all the VMs (See Lines 4-9). If ft_{MAX} is smaller than or equal to d_i, representing t_i can be successfully finished, then it will be allocated to a VM with the minimal energy consumption (See Lines 10-11). If ft_{MAX} is larger than d_i and at the same time ft_{MIN} is smaller than or equal to d_i. *EASU* chooses the VM with minimal ft_{ijk}^+ to improve the schedulability (See Lines 12-13). If ft_{MIN} is larger than d_i, and $L_i < L_d$, the task t_i has to be rejected, whereas if $L_i > L_d$, which indicates that starting up a host and then creating a new VM to run t_i is feasible (see Lines

Algorithm 1: Pseudocode of *EASU*

```
1  WQ ← ∅; UQ ← ∅;
2  foreach new task t_i do
3      ft_MAX ← 0; ft_MIN ← +∞;
4      foreach VM v_j k do
5          Calculate its estimated minimal finish time ft^-_ijk and
           maximal finish time ft^+_ijk and energy ec_ijk;
6          if ft^-_ijk < ft_MIN then
7              ft_MIN ← ft^-_ijk;
8          if ft^+_ijk > ft_MAX then
9              ft_MAX ← ft^+_ijk;
10     if ft_MAX ≤ d_i then
11         Allocate t_i to the VM v_jk with minimal ec_ijk;
12     else if ft_MAX > d_i & ft_MIN ≤ d_i then
13         Allocate t_i to the VM v_jk with minimal ft^+_ijk;
14     else
15         if L_i < L_d then
16             Reject task t_i;
17         else
18             Call scaleUpResources() and Allocate t_i on a new
               VM;
```

Algorithm 2: Pseudocode of Primaries Scheduling in *RAS*

```
1  H_candidate ← top α% hosts in H_a;
2  eft ← +∞; v ← NULL;
3  while !all hosts in H_a have been scanned do
4      foreach h_k in H_candidate do
5          if h_k satisfies t_i's scheduling dependent constraints
           then
6              foreach v_kl in h_k.VmList do
7                  Calculate the earliest start time est_i;
8                  eft^P_i ← est^P_i + e^P_ikl;
9                  if eft^P_i < eft then
10                     eft ← eft^P_i;
11                     v ← v_kl;
12     if eft > d_i then
13         H_candidate ←next top α% hosts in H_a;
14     else
15         break;
16 if eft > d_i then
17     if scaleUpResources(t_i) then
18         return true;
19     else
20         Allocate t_i to v_kl;
21 else
22     Allocate t^P_i to v_kl;
```

15-18). The function scaleUpResources() will be introduced in resource provisioning algorithms.

B. RAS Algorithm

Example 2 is about the reliability-aware scheduling using PB model, thus the primaries and backups should be scheduled, respectively. Algorithm 2 is the pseudocode of primaries scheduling in *RAS*.

Algorithm 2 firstly selects the top $\alpha\%$ hosts with fewer primaries as candidate hosts (see Line 1). Then, the VM that offers the earliest finish time for the primary copy is selected (see Lines 4-11). If no VM on the candidate hosts can finish the primary before its deadline, the next top $\alpha\%$ hosts are chosen for the next round search (see Lines 12-15). By this method, the primaries can be evenly distributed. If no existing VMs can accommodate the primary, the function scaleUpResources() will be called (see Line 17).

Backup scheduling algorithm is similar to Algorithm 2, so it is omitted in this paper. It should be noted that a backup cannot be scheduled to the host on which its primary has been allocated. Besides, some dependent constraints must be incorporated in backup scheduling algorithm.

C. Resource Provisioning Algorithm

Resource provisioning algorithm include two functions, i.e., scaleUpResources() and scaleDownResources(). Now, we present them devised in our previous work.

Algorithm 3: Pseudocode of *scaleUpResources()*

```
1  Select a kind of VM v_j with minimal MIPS on condition that
   t_i can be finished before its deadline;
2  Sort the hosts in H_a in the decreasing order of the CPU
   utilization;
3  foreach host h_k in H_a do
4      if VM v_j can be added in host h_k then
5          Create VM v_jk; findTag ← TRUE; break;
6  if findTag == FALSE then
7      Search the host h_s with minimal CPU utilization;
8      Find the VM v_ps with minimal MIPS in h_s;
9      foreach host h_k except h_s in H_a do
10         if VM v_ps can be added in host h_k then
11             Migrate VM v_ps to host h_k; break;
12     if VM v_j can be added in host h_s then
13         Create VM v_js;
14         if t_i can be finished in v_js before its deadline then
15             findTag ← TRUE;
16 if findTag == TRUE then
17     Start a host h_n and put it in H_a;
18     Create VM v_jn on h_n;
19     if t_i can be finished in v_jn before its deadline then
20         findTag ←; TRUE;
```

Function *scaleUpResources()* selects a host with possibly minimal capability to accommodate v_j (See Lines 3-5). If no such host can be found, it migrates the VM (See Lines 7-11). After that, it checks if VM v_j can be added to the host where a VM has been migrated. If so, the v_j will be created and it checks whether the task can be finished on v_j before the task's deadline (See Lines 12-15). If no migration is feasible or the task cannot be successfully finished, then it starts up a host h_n and then creates v_{jn} on it. Then it checks if the task can be finished successfully on v_{jn} (See Lines 16-20).

Algorithm 4: Pseudocode of *scaleDownResources()*

```
1  SH ← ∅; DH ← ∅;
2  foreach VM v_jk in the system do
3      if v_jk's idle time it_jk > THRESH then
4          Remove VM v_jk from host h_k and delete it;
5  foreach host h_k in H_a do
6      if there is no VM on h_k then
7          Shut down host h_k and remove it from H_a;
8  Sort the hosts in H_a in an increasing order of the CPU
   utilization;
9  SH ← H_a; DH ← H_a and sort DH inversely;
10 foreach host h_k in SH do
11     shutDownTag ← TRUE; AH ← ∅;
12     foreach VM v_jk in h_k do
13         migTag ← FALSE;
14         foreach host h_p in DH except h_k do
15             if v_jk can be added in h_p then
16                 migTag ← TRUE; AH ← h_p; break;
17         if migTag == FALSE then
18             shutDownTag ← FALSE; break;
19     if shutDownTag ← TRUE then
20         Migrate VMs in h_k to destination hosts;
           SH ← SH − AH − h_k; DH ← DH − h_k;
21         Shut down host h_k and remove it from H_a;
```

In *scaleDownResources()*, if there exists any VM whose idle time is larger than a predefined threshold, then it deletes this VM (See Lines 2-4). If no VMs run on a host, it shuts down this host (See Lines 5-7). If all the VMs running on a host in SH can be added to one or some hosts in DH, it migrates these VMs to destination hosts, and then shuts down the host after migration. Otherwise, in the host if there is one or some VMs that cannot be migrated, then it gives up the migration of all the VMs on the host (See Lines 10-21).

VI. Framework Implementation

We conducted multiple experiments under our framework. In order to ensure the repeatability of the experiments, we choose CloudSim toolkit [27] as a simulation platform. Also, we deploy an experimental cloud environment on the basis of Apache CloudStack 4.2.0. We set up a KVM cluster with five Dell Optiplex 7010 MT, each of physical host has one CPU (i3 3.9GHz 4 cores), 3.7G memory and 500G disk storage, and the peak power of a host is 200W. In addition, the CPU and memory required for each VM are two CPU cores (i.e., 23.9 GHz) and 1.5G, respectively. In addition, the five machines run on a 1Gbps Ethernet network [24].

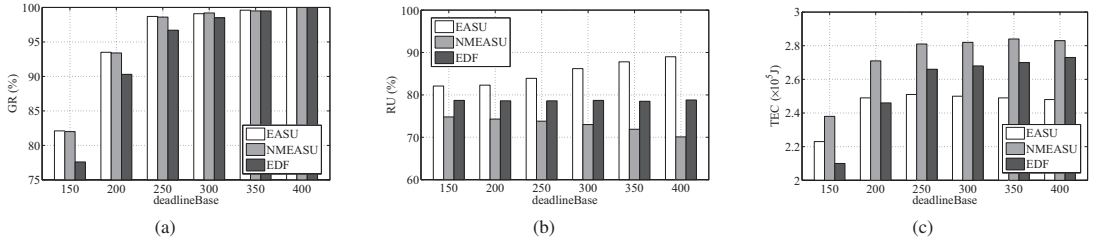

Fig. 4. Performance Impacts of Task Deadlines.

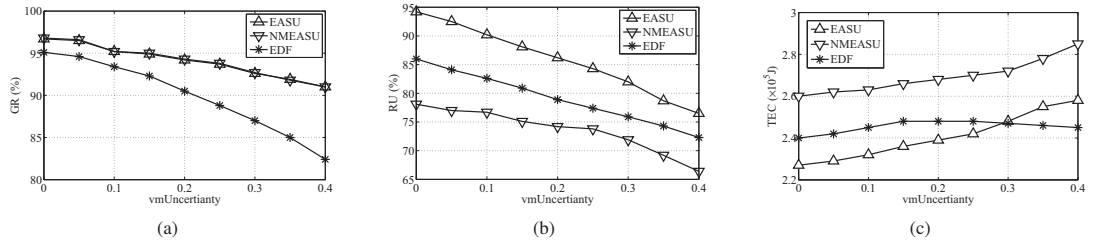

Fig. 5. Performance Impacts of VM Uncertainty.

A. Parameter Setting and Experimental Results of Example 1

To embody the uncertainty, we use parameter $vmUncertainty$ to represent the uncertainty upper bounds of VMs in the system, and the lower and upper bonds of a VM's performance are modeled as below:

$$p_{jk}^{-} = p_{jk}^{+} \times (1 - U[0, vmUncertainty]). \quad (12)$$

where $U[0, vmUncertainty]$ is an uniformly distributed random variable between 0 and $vmUncertainty$, and p_{jk}^{+} is the CPU performance capacity required for vm_{jk}.

To reflect tasks' real-time nature, we use parameter $deadlineBase$ to control a task's deadline as follows:

$$d_i = a_i + U[deadlineBase, a \times deadlineBase]. \quad (13)$$

where parameter $deadlineBase$ decides whether the deadline of a task is loose or tight. We set $deadlineBase = 400s$, $a = 4$ in this study.

The arrival rate is in Poisson distribution, and we use parameter $intervalTime$ to determine the time interval between two consecutive tasks. We set $intervalTime = 0.5s$.

We compare *EASU* with two baseline algorithms - *NMEASU* (no migration is considered in *scaleDownResources()*) and *EDF* (Earliest Deadline First) from the following three metrics:

1) Guarantee Ratio (GR): the ratio of tasks finished before their deadlines;

2) Resource Utilization (RU): the average host utilization that can be calculated as:

$$RU = (\sum_{i=1}^{|T|} \sum_{j=1}^{|H_a|} \sum_{k=1}^{|V_k|} et_{ijk} \cdot z_{ijk}) / (\sum_{j=1}^{|H_a|} p_j \cdot wt_j), \quad (14)$$

where wt_j is the active time for host h_j during an experiment.

3) Total Energy Consumption (TEC): the total energy consumed by hosts to execute task set T.

1) Performance Impact of Task Deadline: It can be observed from Fig. 4(a) that the GRs of the three algorithms increase with the increase of $deadlineBase$ because more tasks can be finished with their deadlines without using the resource scale-up mechanism. Another observation is that *EASU* and *NMEASU* have higher GRs than *EDF*. The reason is three-fold: 1) *EASU* and *NMEASU* give urgent tasks high priority so that more tasks can meet their deadline; 2) *EASU* and *NMEASU* can dynamically add resources to accommodate more tasks; 3) the uncertainties are considered in *EASU* and *NMEASU* leading to better scheduling quality.

From Fig. 4(b), we can observe that when $deadlineBase$ increases, the RUs of the three algorithms increase, which can be contributed to the fact that as the deadlines of tasks become looser, more tasks can be finished in the current active hosts without starting more hosts, thus the utilization of active hosts is higher. In addition, *EASU* has higher RU than *NMEASU* because VM migration can efficiently consolidate resources and thus resource can be effectively utilized. There is no scale-down function designed for *EDF* resulting in the lowest RU.

Fig. 4(c) shows that the TECs of *EASU*, *NMEASU*, and *EDF* become larger with the increase of $deadlineBase$. The reason is that as $deadlineBase$ increases, more tasks will be executed leading to more energy being consumed. *EASU* consumes less energy than *NMEASU* because *NMEASU* does not efficiently use resource. Without considering energy conservation, *EDF* consumes more energy than *EASU* when $deadlineBase$ is larger than 200s.

2) Performance Impact of Uncertainty: From Fig. 5(a), we can observe that when $vmUncertainty$ increases, the GRs of the three algorithms decrease, especially the trend of *EDF*

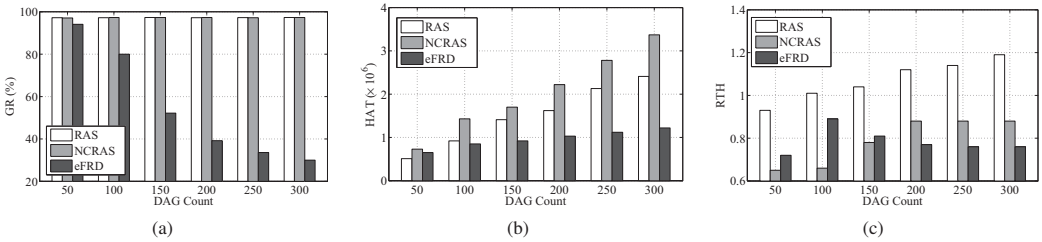

Fig. 6. Performance Impacts of DAG Count.

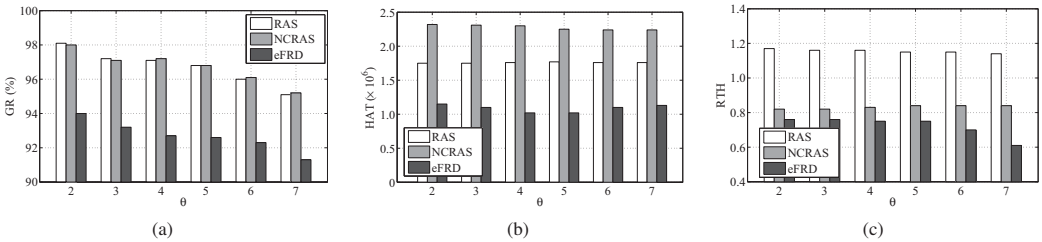

Fig. 7. Performance Impacts of Task Dependence.

is outstanding. The reason is that *EDF* does not employ any strategies to control the uncertainties while scheduling resulting in the worst scheduling quality.

Fig. 5(b) depicts that the RUs of the three algorithms descend significantly with the increase of $vmUncertainty$. The reason is that when $vmUncertainty$ increases, the performance degradation of VMs will become pronounced, resulting in consuming more resources of physical hosts. Specifically, *EASU* performs better than others because of employing the uncertainty-aware scheduling and VM migration strategy.

Fig. 5(c) shows that with the increase of $vmUncertainty$, the TECs of *EASU*, and *NMEASU* increase because they allocate more urgent tasks that cannot be completed before their deadlines to VMs as $vmUncertainty$ becomes larger, thus costing more resources. *EASU* has more energy consumption than *EDF* when $vmUncertainty$ is larger than 0.3, This can be explained that *EASU* strives to control the uncertainty and accept more tasks with the sacrifice of energy consumption.

B. Parameter Setting and Experimental Results of Example 2

Example 2 considers the dependent tasks. We assume each task set T (or a DAG job) has random precedence constraints that are generated by the steps in [26]. Given a number N of dependent tasks, we set the message count M among them is $M = \theta \times N$. Other parameters are the same to Example 1.

We compare *RAS* with two baseline algorithms - *NCRAS* (no consolidation is considered in *RAS*) and *eFRD* (As Early as Possible Strategy for Primary and Backup Scheduling) [26] from the following three metrics:

1) Guarantee Ratio (GR): the same to that in Example 1;

2) Host Active Time (HAT): the total active time of all hosts in cloud;

3) Ratio of Task time and Hosts time (RTH): the ratio of total tasks' execution time over total active time of hosts.

1) Performance Impact of DAG Count: It can be observed from Fig. 6(a) that *RAS* and *NCRAS* basically maintain stable GRs for different DAG counts, which can be attributed to the fact that *RAS* and *NCRAS* consider the infinite resources in the cloud, thus when DAG count increases, new hosts will be dynamically available for more DAGs. While eFRD has no ability to adjust resources, so with the increase of DAG count, the GR of eFRD decreases.

In Fig. 6(b), *RAS* has lower HAT than *NCRAS*, which indicates the policies in *RAS* can work well to improve the resource utilization. In addition, because there is no consolidation mechanism employed in *NCRAS*, some resources are in idle state, resulting in the highest resource consumption.

It can be seen from Fig. 6(c) that the RTHs of *RAS* and *NCRAS* increase. For eFRD, its RTH increases first and then decreases, which can be explained below. With eFRD, the system resource is assumed fixed. The resource is sufficient enough to accommodate most DAGs (in the range from 50 to 100). But the system becomes saturated when DAG count is further increased (DAG count $>$ 100), making the host running longer and therefore lowing the resource utilization. We can also find that due to incorporating the mechanism to consolidate resources, *RAS* hence achieves the highest RTH.

2) Performance Impact of Task Dependence: From Fig. 7(a), we can observe that with the increase of θ, the GRs of all the algorithms slightly decrease. This is because the increased task dependency degrades the system schedulability. It can also be found that similar to other experiments, *RAS* and *NCRAS* offer the highest GRs.

Fig. 7(b) shows that *NCRAS* has higher HATs than others. With the small task dependence, more tasks in a DAG can

be executed in parallel and can be finished earlier; thus some hosts can be freed and shut down earlier. However, such an advantage is not exploited by *NCRAS*. Hence it consumes more resources. When θ increases, the tasks that can be executed in parallel decrease, thus the difference of HATs between *NCRAS* and other algorithms becomes smaller.

It can be observed from Fig. 7(c) that *RAS* maintains the highest RTH. With the increase of θ, the RTHs of all the algorithms decrease because high task dependence increases execution constraints, making less tasks available to VMs even if some of them are idle. When the value of θ becomes larger, the tasks that can be executed in parallel decrease, hence less resource is wasted leading to lower RTH.

VII. Conclusions

This paper investigates a general framework for task scheduling and resource provisioning in virtualized clouds. In this framework, different scheduling management objectives can be flexibly assembled to process diverse tasks. Specifically, we first propose a novel scheduling architecture that significantly considers any kind of combination of objectives and task features, thus it can be applied to more general scheduling in clouds where a variety of distinct requests must be handled. Besides, a scheduling process using our framework is presented in detail. Under this framework, we present two examples with different objective combinations for different kinds of tasks. General models and specific models are presented based on the two examples to support our framework. Moreover, we propose two algorithms - *EASU* and *RAS* for Example 1 and Example 2, respectively, which can be dynamically added in the algorithm library of our framework. Furthermore, we conduct several experiments to validate our algorithms and implement the proposed framework.

VIII. Acknowledgments

This research was supported by the National Natural Science Foundation of China under grants No. 61572511 and No. 11428101, the Hunan Natural Science Foundation of China under grant No. 2015JJ3023, and the Southwest Electron & Telecom Technology Institute under Grant No. 2015014. Xiaomin Zhu is the corresponding author.

References

[1] J. Rao, Y. Wei, J. Gong, and C. Xu, "QoS guarantees and service differentiation for dynamic cloud applications," *IEEE Trans. Network and Service Management*, vol. 10, no. 1, pp. 43-55, 2013.

[2] G. Mehta, E. Deelman, J. A. Knowles, T. Chen, Y. Wang, J. Vöckler, S. Buyske, T. Matise, "Enabling data and compute intensive workflows in bioinformatics," *Euro-Par: Parallel Processing Workshops*, pp.23-32, 2011.

[3] M. Cardosa, A. Singh, H. Pucha, and A. Chandra, "Exploiting spatiotemporal tradeoffs for energy-aware mapreduce in the cloud," *IEEE Trans. Computers*, vol. 61, no. 12, pp. 1737-1751, 2012.

[4] W. Deng, F. Liu, H. Jin, and C. Wu, "Smartdpss: cost-minimizing multi-source power supply for datacenters with arbitrary demand," *Proc. Int'l Conf. Distributed Computing Systems (ICDCS '13)*, pp. 420-429, 2013.

[5] J. Singh, S. Betha, B. Mangipudi, and N. Auluck, "Contention aware energy efficient scheduling on heterogeneous multiprocessors," I*IEEE Trans. Parallel and Distributed Systems*, vol. 26, no. 5, pp. 1251-1264, 2015.

[6] J. Wang, W. Bao, X. Zhu, L. T. Yang, Y. Xiang, "FESTAL: fault-tolerant elastic scheduling algorithm for real-time tasks in virtualized clouds," *IEEE Trans. Computers*, vol. 64, no. 9, pp. 2545-2558, 2015.

[7] K. Plankensteiner and R. Prodan, "Meeting soft deadlines in scientific workflows using resubmission impact," *IEEE Trans. Parallel and Distributed Systems*, vol. 23, no. 5, pp. 890-901, 2012.

[8] A. Zhou, S. Wang, Z. Zheng, C. Hsu, M. R. Lyu, and F. Yang, "On cloud service reliability enhancement with optimal resource usage," *IEEE Trans. Cloud Computing*, DOI 10.1109/TCC.2014.2369421, 2015.

[9] R. Duan, R. Prodan, and X. Li, "Multi-objective game theoretic scheduling of bag-of-tasks workflows on hybrid clouds," *EEE Trans. Cloud Computing*, vol. 2, no. 1, pp. 29-42, 2014.

[10] M. A. Rodriguez and R. Buyya, "Deadline based resource provisioning and scheduling algorithm for scientific workflows on clouds," *IEEE Trans. Cloud Computing*, vol. 2, no. 2, pp. 222-235, 2014.

[11] M. Mao and M. Humphrey, "Scaling and scheduling to maximize application performance within budget constraints in cloud workflows," *Proc. 27th Int'l Symp. Parallel & Distributed Processing*, pp. 67-78, 2013.

[12] W. Xiao, W. Bao, X. Zhu, C. Wang, L. Chen, L. T. Yang, "Dynamic request redirection and resource provisioning for cloud-based video services under heterogeneous environment," *IEEE Trans. Parallel and Distributed Systems*, DOI: 10.1109/TPDS.2015.2470676, 2015.

[13] S. Abrishami, M. Naghibzadeh, and D. H. Epema, "Deadline-constrained workflow scheduling algorithms for infrastructure as a service clouds," *Future Generation Computer Systems*, vol. 29, no. 1, pp. 158-169, 2013.

[14] S. Hosseinimotlagh, F. Khunjush, and S. Hosseinimotlagh, "A cooperative two-tier energy-aware scheduling for real-time tasks in computing clouds," *Proc. 22nd Euromicro Int'l Conf. Parallel, Distributed, and Network-Based Processing (PDP '15)*, pp. 178-182, 2014.

[15] L. F. Bittencourt, E. R. M. Madeira, and N. L. S. da Fonseca, "Scheduling in hybrid clouds," *IEEE Communications Magazine*, pp. 42-47, 2012.

[16] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth, B. Saha, C. Curino, O. Malley, S. Radia, B. Reed, and E. Baldeschwieler, "Apache hadoop YARN: yet another resource negotiator," *In: Proc. 4th ACM Symp. Cloud Computing (SoCC '13)*, 2013.

[17] E. Boutin, J. Ekanayake, W. Lin, B. Shi, J. Zhou, Z. Qian, M. Wu, and L. Zhou, "Apollo: scalable and coordinated scheduling for cloud-scale computing," *Proc. 11th USENIX Symp. Operating Systems Design and Implementation (OSDI '14)*, pp. 285-300, 2014.

[18] O. Sukwong, A. Sangpetch, and H. Kim, "SageShift: managing SLAs for highly consolidated cloud," *Proc. 31st IEEE Int'l Conf. Computer Communicatons (Infocom '12)*, pp. 208-216, 2012.

[19] X. Zhu, C. He, K. Li, and X. Qin, "Adaptive energy-efficient scheduling for real-time tasks on DVS-enabled heterogeneous clusters," *J. Parallel and Distributed Computing*, vol. 72, no. 6, pp. 751-763, 2012.

[20] J. Dean, "Designs, lessons and advice from building large distributed systems," *Proc. 3rd ACM SIGOPS Int'l Workshop on Large Scale Distributed Systems and Middleware (LADIS 09)*, 2009.

[21] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, and I. Stoica, "A view of cloud computing," *Communications of the ACM*, vol. 53, no. 4, pp. 50-58, 2010.

[22] R. Buyya, C. S. Yeo, S. Venugopal, J. Broberg, I. Brandic, "Cloud computing and emerging IT platforms: vision, hype, and reality for delivering computing as the 5th utility," *Future Generation Computer Systems*, vol. 57, no. 3, pp. 599-616, 2009.

[23] A. Sengupta, T. K. Pal, "Fuzzy preference ordering of interval numbers in decision problems," 2009, Springer.

[24] H. Chen, X. Zhu, H. Guo, J. Zhu, X. Qin, J. Wu, "Towards energy-efficient scheduling for real-time tasks under uncertain cloud computing environment," emphJ. Systems and Software, vol. 99, pp. 20-35, 2015.

[25] A. Amin, R. A. Ammar, and S. S. Gokhale, "An Efficient Method to Schedule Tandem of Real-Time Tasks in Cluster Computing with Possible Processor Failures," *Proc. 8th IEEE Intl Symp. Computers and Communication (ISCC '03)*, pp. 1207-1212, Jun. 2003.

[26] X. Qin and H. Jiang, "A Novel Fault-Tolerant Scheduling Algorithm for Precedence Constrained Tasks in Real-Time Heterogeneous Systems," *Parallel Computing*, vol. 32, no. 5, pp. 331-356, 2006.

[27] R. N. Calheiros, R. Ranjan, A. Beloglazov, C. A. De Rose, and R. Buyya, "CloudSim: a Toolkit for Modeling and Simulation of Cloud Computing Environments and Evaluation of Resource Provisioning Algorithms," *Software: Practice and Experience*, vol. 41, no. 1, pp. 23-50, 2011.