Monetary Cost Optimizations for Hosting Workflow-as-a-Service in IaaS Clouds

Amelie Chi Zhou, Bingsheng He, and Cheng Liu

Abstract—Recently, we have witnessed workflows from science and other data-intensive applications emerging on Infrastructure-as-a-Service (IaaS) clouds, and many workflow service providers offering workflow-as-a-service (WaaS). The major concern of WaaS providers is to minimize the monetary cost of executing workflows in the IaaS clouds. The selection of virtual machines (instances) types significantly affects the monetary cost and performance of running a workflow. Moreover, IaaS cloud environment is *dynamic*, with high performance dynamics caused by the interference from concurrent executions and price dynamics like spot prices offered by Amazon EC2. Therefore, we argue that WaaS providers should have the notion of offering probabilistic performance guarantees for individual workflows to explicitly expose the performance and cost dynamics of IaaS clouds to users. We develop a scheduling system called *Dyna* to minimize the expected monetary cost given the user-specified probabilistic deadline guarantees. Dyna includes an $A^\star$-based instance configuration method for performance dynamics, and a hybrid instance configuration refinement for using spot instances. Experimental results with three scientific workflow applications on Amazon EC2 and a cloud simulator demonstrate (1) the ability of Dyna on satisfying the probabilistic deadline guarantees required by the users; (2) the effectiveness on reducing monetary cost in comparison with the existing approaches.

Index Terms—Cloud computing, cloud dynamics, spot prices, monetary cost optimizations, scientific workflows

1 Introduction

Cloud computing has become a popular computing infrastructure for many scientific applications. Recently, we have witnessed many workflows from various scientific and data-intensive applications deployed and hosted on the Infrastructure-as-a-Service (IaaS) clouds such as Amazon EC2 and other cloud providers. In those applications, workflows are submitted and executed in the cloud and each workflow is usually associated with a deadline as performance guarantee [1], [2], [3]. This has formed a new software-as-a-service model for hosting workflows in the cloud, and we refer it as Workflow-as-a-Service (WaaS). WaaS providers charge users based on the execution of their workflows and QoS requirements. On the other hand, WaaS providers rent cloud resources from IaaS clouds, which induces the monetary cost. Monetary cost is an important optimization factor for WaaS providers, since it directly affects the profit of WaaS providers. In this paper, we investigate whether and how WaaS providers can reduce the monetary cost of hosting WaaS while offering performance guarantees for individual workflows.

Monetary cost optimizations have been classic research topics in grid and cloud computing environments. Over the era of grid computing, cost-aware optimization techniques have been extensively studied. Researchers have addressed various problems: minimizing cost given the performance requirements [4], maximizing the performance for given budgets [5] and scheduling optimizations with both cost and performance constraints [6]. When it comes to cloud computing, the pay-as-you-go pricing, virtualization and elasticity features of cloud computing open up various challenges and opportunities [1], [7]. Recently, there have been many studies on monetary cost optimizations with resource allocations and task scheduling according to the features of cloud computing (e.g., [1], [2], [7], [8], [9], [10], [11]). Although the above studies have demonstrated their effectiveness in reducing the monetary cost, all of them assume static task execution time and consider only fixed pricing scheme (only *on-demand* instances in Amazon's terminology). Particularly, they have the following limitations.

First, cloud is by design a shared infrastructure, and the interference causes significant variations in the performance even with the same instance type. Previous studies [12], [13] have demonstrated significant variances on I/O and network performance. The assumption of static task execution time in the previous studies (e.g., [1], [2], [7], [8], [9], [10]) does not hold in the cloud. Under the static execution time assumption, the deadline notion is a "deterministic deadline". Due to performance dynamics, a more rigorous notion of deadline requirement is needed to cope with the dynamic task execution time.

Second, cloud, which has evolved into an economic market [14], has dynamic pricing. Amazon EC2 offers spot instances, whose prices are determined by market demand and supply. Spot instances have been an effective means to reduce monetary cost [15], [16], because the spot price is usually much lower than the price of on-demand instances of the same type. However, a spot instance may be terminated at any time when the bidding price is lower than the spot price (i.e., out-of-bid events). The usage of spot instances may cause excessive long

• *The authors are with the School of Computer Engineering, Nanyang Technological University, Singapore 637598.*
E-mail: {czhou1, LIUC0012}@e.ntu.edu.sg, bshe@ntu.edu.sg.

Manuscript received 14 Aug. 2014; revised 14 Jan. 2015; accepted 12 Feb. 2015. Date of publication 0 . 0000; date of current version 0 . 0000.
Recommended for acceptance by K. Keahey, I. Raicu, K. Chard, B. Nicolae.
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.2015.2404807

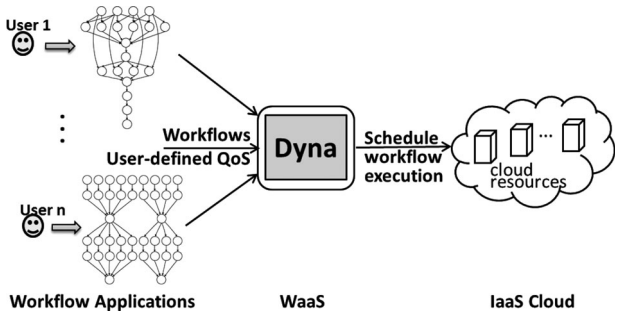


Fig. 1. Application scenario of this study.

latency due to failures. Most of the previous studies do not consider deadline constraints of individual workflows when using spot instances.

In order to address performance and price dynamics, we define the notion of probabilistic performance guarantees to explicitly expose the performance dynamics to users. Each workflow is associated with a probabilistic deadline requirement of p_r percent. WaaS provider guarantees that the p_rth percentile of the workflow's execution time distribution in the dynamic cloud environment is no longer than the predefined deadline. The WaaS provider may charge differently according to the deadline and the probability in the performance guarantee, and the users can select the suitable performance guarantee according to their requirements. This is just like many IaaS cloud providers offer different probabilistic availability guarantees. Under this notion, we propose a probabilistic scheduling system called *Dyna* to minimize the cost of the WaaS provider while satisfying the probabilistic performance guarantees of individual workflows predefined by the user. The system embraces a series of optimization techniques for monetary cost optimizations, which are specifically designed for cloud dynamics. We develop probabilistic models to capture the performance dynamics in I/O and network of instances in IaaS clouds. We further propose a hybrid instance configuration approach to adopt both spot and on-demand instances and to capture the price dynamics in IaaS clouds. The spot instances are adopted to potentially reduce monetary cost and on-demand instances are used as the last defense to meet deadline constraints.

We calibrate the cloud dynamics from a real cloud provider (Amazon EC2) for the probabilistic models on I/O and network performance as well as spot prices. We perform experiments using three workflow applications on Amazon EC2 and on a cloud simulator. Our experimental results demonstrate the following two major results. First, with the calibrations from Amazon EC2, Dyna can accurately capture the cloud dynamics and guarantee the probabilistic performance requirements predefined by the users. Second, the hybrid instance configuration approach significantly reduces the monetary cost by 15-73 percent over other state-of-the-art algorithms [1] which only adopt on-demand instances.

The rest of the paper is organized as follows. We formulate our problem and review the related work in Section 2. We present our detailed system design in Section 3, followed by the experimental results in Section 4. Finally, we conclude this paper in Section 5.

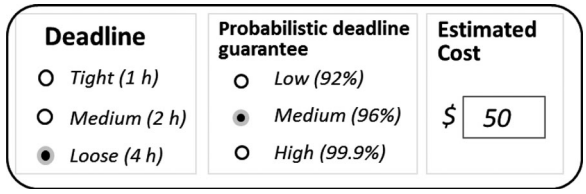


Fig. 2. Illustrative user interface.

2 Background and Related Work

2.1 Application Scenario

Fig. 1 illustrates our application scenario. In this study, we consider a typical scenario of offering software-as-a-service model for workflows on IaaS clouds [1]. We call this model Workflow-as-a-Service. We consider three parities in this scenario, namely the workflow application owner, WaaS provider and IaaS cloud provider. In this hosting, different application owners submit a number of workflows with different parameters to WaaS and the WaaS provider rent resources from the cloud provider to serve the applications.

The application owners submit workflows with specified deadlines for QoS purposes. WaaS providers charge users according to the execution of workflows and their QoS requirements. In this proposal, we argue that the WaaS provider should offer a probabilistic performance guarantee for users. Particularly, we can offer some fuzzy-style interfaces for users to specify their probabilistic deadline requirements, such as "Low", "Medium" and "High", as illustrated in Fig. 2. Inside Dyna, we translate these requirements into probabilities of deadline. For example, the user may select the loose deadline of 4 hours with the probability of 96 percent. Ideally, the WaaS provider tends to charge higher prices to users when they specify tighter deadline and/or higher probabilistic deadline guarantee. The design of the billing scheme for WaaS is beyond the scope of this paper, and we will explore it as future work.

Different workflow scheduling and resource provisioning algorithms can result in significant differences in the monetary cost of WaaS providers running the service on IaaS clouds. Considering the cloud dynamics, our goal is to provide a probabilistic scheduling system for WaaS providers, aiming at minimizing the expected monetary cost while satisfying users' probabilistic deadline requirements.

2.2 Terminology

Instance. An instance is a virtual machine offered by the cloud provider. Different types of instances can have different amount of resources such as CPUs and RAM and different capabilities such as CPU speed, I/O speed and network bandwidth. We model the dynamic I/O and network performances as probabilistic distributions. The details are presented in Section 3.

We adopt the instance definition of Amazon EC2, where an instance can be on-demand or spot. Amazon adopts the hourly billing model, where any partial hour of instance usage is rounded to 1 hour. Both on-demand and spot instances can be terminated when users no longer need them. If an instance is terminated by the user, the user has to pay for any partial hour (rounded up to one hour). For a

TABLE 1
Statistics on Spot Prices ($/hour, August 2013, US East Region) and On-Demand Prices of Amazon EC2

Instance type	Average	$stdev$	Min	Max	OnDemand
m1.small	0.048	0.438	0.007	10	0.06
m1.medium	0.246	1.31	0.0001	10	0.12
m1.large	0.069	0.770	0.026	40	0.24
m1.xlarge	0.413	2.22	0.052	20	0.48

spot instance, if it is terminated due to an out-of-bid event, users do not need to pay for any partial hour of usage.

Table 1 shows some statistics of the price history of four types of spot instances on Amazon in the US East region during August 2013. We also show the price of the on-demand instances for those four types. We have the following observations: a) The spot instances are usually cheaper than on-demand instances. There are some "outlier" points where the maximum spot price is much higher than the on-demand price. b) Different types have different variations on the spot price. These observations are consistent with the previous studies [17], [18].

Task. Tasks can have different characteristics, e.g., compute-intensive and I/O-intensive tasks, according to the dominating part of the total execution time. The execution time (or response time) of a task is usually estimated using estimation methods such as task profiling [19]. In this study, we use a simple performance estimation model on predicting the task execution time. Since scientific workflows are often regular and predictable [1], [4], this simple approach is sufficiently accurate in practice. Specifically, given the input data size, the CPU execution time and output data size of a task, the overall execution time of the task on a cloud instance can be estimated with the sum of the CPU, I/O and network time of running the task on this instance. Note, the CPU performance is usually rather stable [12]. Since the I/O and network performance of the cloud are dynamic (modeled as probabilistic distributions in this paper), the estimated task execution time is also a probabilistic distribution.

Job. A job is expressed as a workflow of tasks with precedence constraints. A job has a soft deadline. In this study, we consider the deadline of a job as a probabilistic requirement. Suppose a workflow is specified with a probabilistic deadline requirement of p_r percent. Rather than offering 100 percent deadline guarantee, WaaS provider guarantees that the p_rth percentile of the workflow's execution time distribution in the dynamic cloud environment is no longer than a predefined deadline constraint. Our definition of probabilistic deadline is consistent with previous studies [20] on defining the QoS in a probabilistic manner.

Instance configuration. The *hybrid instance configuration* of a task is defined as a n-dimension vector: $\langle(type_1, price_1, isSpot_1), (type_2, price_2, isSpot_2), \ldots, (type_n, price_n, isSpot_n)\rangle$, where $isSpot_i$ indicates whether the instance is spot (True) or on-demand (False). If the instance i is a spot instance, $price_i$ is the specified bidding price, and the on-demand price otherwise. In our hybrid instance configuration, only the last dimension of the configuration is on-demand instance and all previous dimensions are spot instances. We set the last dimension to on-demand instance to ensure the deadline of the task.

A hybrid instance configuration indicates a sequence of instance types that the task is potentially to be executed on. In the hybrid execution of spot and on-demand instances, a task is initially assigned to a spot instance of the type indicated by the first dimension of its configuration (if any). If the task fails on this spot instance, it will be re-assigned to an instance of the next type indicated by its configuration until it successfully finishes. Since the last dimension is an on-demand instance type, the task can always finish the execution, even when the task fails on all previous spot instances.

2.3 Related Work

There are a lot of works related to our study, and we focus on the most relevant ones on cost optimizations and cloud performance dynamics.

Cost-aware optimizations. Workflow scheduling with deadline and budget constraints (e.g., [2], [4], [5], [21], [22], [23], [24], [25], [26]) has been widely studied. Yu et al. [4] proposed deadline assignment for the tasks within a job and used genetic algorithms to find optimal scheduling plans. Multi-objective methods such as evolutionary algorithms [27], [28] have been adopted to study the tradeoff between monetary cost and performance optimizations for workflow executions. Those studies only consider a single workflow with on-demand instances only. Malawski et al. [2] proposed dynamic scheduling strategies for workflow ensembles. The previous studies [1], [29], [30], [31] proposed auto-scaling techniques based on static execution time of individual tasks. In comparison with the previous works, the unique feature of Dyna is that it targets at offering probabilistic performance guarantees as QoS, instead of deterministic deadlines. Dyna schedules the workflow by explicitly capturing the performance dynamics (particularly for I/O and network performance) in the cloud. Calheiros and Buyya and Calheiros [21] proposed an algorithm with task replications to increase the likelihood of meeting deadlines.

Due to their ability on reducing monetary cost, Amazon EC2 spot instances have recently received a lot of interests. Related work can be roughly divided into two categories: modeling spot prices [17], [18] and leveraging spot instances [15], [16], [32].

For modeling spot prices, Yehuda et al. [18] conducted reverse engineering on the spot price and figured out a model consistent with existing price traces. Javadi et al. [17], [33] developed statistical models for different spot instance types. Those models can be adopted to our hybrid execution.

For leveraging spot instances, Yi et al. [15] introduced some checkpointing mechanisms for reducing cost of spot instances. Further studies [16] used spot instances with different bidding strategies and incorporating with fault tolerance techniques such as checkpointing, task duplication and migration. Those studies are with spot instance only, without offering any guarantee on meeting the workflow deadline like Dyna. Similar to Dyna, Chu and Simmhan [34] proposed a hybrid method to use both on-demand and spot instances for minimizing total cost while satisfying deadline constraint. However, they did not consider the cloud performance dynamics.

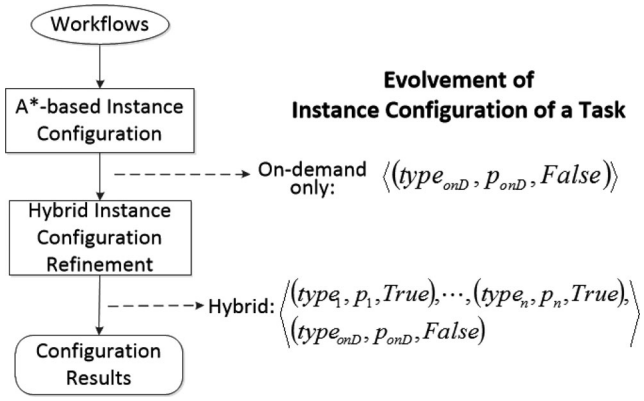


Fig. 3. Overview of the Dyna system.

Cloud performance dynamics. There have been some proposals to reduce the performance interference and unpredictability in the cloud, such as network performance [35] and I/O performance [36], [37]. This paper offers a probabilistic notion to capture the performance and cost dynamics, and further develop a probabilistic scheduling system to minimize the monetary cost with the consideration of those dynamics.

3 System Design and Implementation

We first present an overview of the Dyna system and then discuss the design details about the optimization techniques adopted in Dyna.

3.1 System Overview

We propose Dyna, a workflow scheduling system in order to minimize the monetary cost of executing the workflows in IaaS clouds. Compared with existing scheduling algorithms or systems [1], Dyna is specifically designed to capture the cloud performance and price dynamics. The main components of Dyna are illustrated in Fig. 3.

When a user has specified the probabilistic deadline requirement for a workflow, WaaS providers schedule the workflow by choosing the cost-effective instance types for each task in the workflow. The overall functionality of the Dyna optimizations is to determine the suitable instance configuration for each task of a workflow so that the monetary cost is minimized while the probabilistic performance requirement is satisfied. We formulate the optimization process as a search problem, and develop a two-step approach to find the solution efficiently. The instance configurations of the two steps are illustrated in Fig. 3. We first adopt an $A^\star$-based instance configuration approach to select the on-demand instance type for each task of the workflow, in order to minimize the monetary cost while satisfying the probabilistic deadline guarantee. Second, starting from the on-demand instance configuration, we adopt the hybrid instance configuration refinement to consider using hybrid of both on-demand and spot instances for executing tasks in order to further reduce cost. After the two optimization steps, the tasks of the workflow are scheduled to execute on the cloud according to their hybrid instance configuration. At runtime, we maintain a pool of spot instances and on-demand instances, organized in lists according to different instance types. Instance acquisition/release operations are performed in an auto-scaling manner. For the instances that do not have any task and are approaching multiples of full instance hours, we release them and remove them from the pool. We schedule tasks to instances in the earliest-deadline-first manner. When a task with the deadline residual of zero requests an instance and the task is not consolidated to an existing instance in the pool, we acquire a new instance from the cloud provider, and add it into the pool. In our experiment, for example, Amazon EC2 poses the capacity limitation of 200 instances. If this cap is met, we cannot acquire new instances until some instances are released.

The reason that we divide the search process into two steps is to reduce the solution space. For example, consider searching the instance configuration for a single task, where there are n on-demand types and m spot instance types. If we consider spot and on-demand instances together, the number of configurations to be searched is $\binom{n}{1} \times \binom{m}{1}$ while in our divide-and-conquer approach, the complexity is reduced to $\binom{n}{1} + \binom{m}{1}$. In each search step, we design efficient techniques to further improve the optimization effectiveness and efficiency. In the first step, we only consider on-demand instances and utilize the pruning capability of $A^\star$ search to improve the optimization efficiency. In the second step, we consider the hybrid of spot and on-demand instances as the refinement of the instance configuration obtained from the first step. We give the following example to illustrate the feasibility of the two-step optimization.

Example 1. Consider the instance configuration for a single task. In the $A^\star$-based instance configuration step, the on-demand instance configuration found for the task is $\langle(0, 0.1, False)\rangle$. In the refinement step, the on-demand instance configuration is refined to $\langle(0, 0.01, True), (0, 0.1, False)\rangle$. Assume the expected execution time of the task on type 0 instance is 1 hour and the spot price is lower than \$0.01 (equals to \$0.006) for 50 percent of the time. The expected monetary cost of executing the task under the on-demand instance configuration is \$0.1 and under the hybrid instance configuration is only \$0.053 ($\$0.006 \times 50\% + \$0.1 \times 50\%$). Thus, it is feasible to reduce the expected monetary cost by instance configuration refinement in the second step.

In the remainder of this section, we outline the design of the optimization components, and discuss on the implementation details.

3.2 $A^\star$-Based On-Demand Instance Configuration

In this optimization step, we determine an on-demand instance type for each task in the workflow so that the monetary cost is minimized while the probabilistic performance guarantee specified by the user is satisfied. We formulate the process into an $A^\star$-based search problem. The reason that we choose $A^\star$ search is to take advantage of its pruning capability to reduce the large search space while targeting at a high quality solution. The challenging issues of developing the $A^\star$-based on-demand instance configuration include 1) how to define the state, state transitions and the search heuristics in $A^\star$ search; and 2) how to perform the

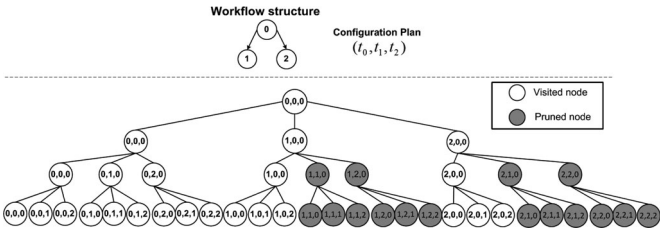


Fig. 4. An example of the configuration plan search tree in our $A^\star$ algorithm.

state evaluation so that the performance dynamics are captured to satisfy the probabilistic performance guarantee.

3.2.1 $A^\star$ Search Process

The process of $A^\star$ search can be modeled as a search tree. In the formulated $A^\star$ search, we first need to clarify the definitions of the *state* and the *state transitions* in the search tree. A state is a configuration plan to the workflow, represented as a multi-dimensional vector. Each dimension of the vector represents the instance configuration of an on-demand instance type for each task in the workflow. This configuration is extended to hybrid instance configuration in the hybrid instance configuration refinement (described in Section 3.3). For example, as shown in Fig. 4, a search state for a workflow with three tasks is represented as (t_0, t_1, t_2), meaning that task i $(0 \leq i \leq 2)$ is configured with on-demand instance type t_i. Starting from the initial state (root node of the search tree), the search tree is traversed by transitting from a state to its child states level by level. At level l, the state transition is to replace the lth dimension in the state with all equally or more expensive instance types. In the example of Fig. 4, suppose there are three on-demand instance types (type 0, 1 and 2 with increasing on-demand prices). From the initial state (represented as $(0, 0, 0)$) where all tasks are assigned to the cheapest instance type (instance type 0), we move to its child states by iterating the three available instance types for the first task (i.e., instance type 0, 1 and 2 and child states $(0, 0, 0)$, $(1, 0, 0)$ and $(2, 0, 0)$).

$A^\star$ search adopts several heuristics to enable its pruning capability. Particularly, $A^\star$ evaluates a state s by combining two distance metrics $g(s)$ and $h(s)$, which are the actual distance from the initial state to the state s and the estimated distance from the state s to the goal state, respectively. $g(s)$ and $h(s)$ are also referred as g score and h score for s, respectively. We estimate the total search cost for s to be $f(s) = g(s) + h(s)$. In the $A^\star$-based instance configuration, we define both $g(s)$ and $h(s)$ to be the monetary cost of configuration plan s. This is because if the monetary cost of a state s is higher than the best found result, its successors are unlikely to be the goal state since they have more expensive configurations than s. For example, assume state $(1, 1, 0)$ on the search tree in Fig. 4 has a high search cost, the grey states on the search tree are pruned since they have higher monetary cost than state $(1, 1, 0)$. During the $A^\star$ search, we maintain two lists, namely the OpenList and ClosedList. The OpenList contains states that are potential solutions to the problem and are to be searched later. States already been searched or with high search cost are added to the ClosedList and do not need to be considered again during the $A^\star$ search.

Algorithm 1 shows the optimization process of the $A^\star$-based instance configuration algorithm. Iteratively, we fetch states from the OpenList and add their neighboring states into the OpenList. Note, we only consider the feasible states that satisfy the probabilistic deadline guarantee (Line 7-9). $estimate\_performance$ is used to estimate the feasibility of states. We maintain the lowest search cost found during the search process as the *upper bound* to prune the unuseful states on the search tree (Line 12-13). Function $estimate\_cost$ returns the estimation for the h and g scores of states. When expanding the OpenList, we only add the neighboring states with lower search cost than the upper bound (Line 17-23).

Algorithm 1. $A^\star$-Based Instance Configuration Search from Initial State S to Goal State D

Require: Max_iter: Maximum number of iterations;
$deadline, p_r$: Required probabilistic deadline guarantee
1: ClosedList = empty;
2: OpenList = S;
3: $upperBound$ = 0;
4: $g[S] = h[S] = estimate\_cost(S)$;
5: $f[S] = g[S] + h[S]$;
6: **while not** (OpenList is empty **or** reach Max_iter) **do**
7: $current$ = the state in OpenList having the lowest f value;
8: $percentile = estimate\_performance(current, p_r)$;
9: **if** $percentile <= deadline$ **then**
10: $g[current] = h[current] = estimate\_cost(current)$;
11: $f[current] = g[current] + h[current]$;
12: **if** $f[current] < upperBound$ **then**
13: $upperBound = f[current]$;
14: D = $current$;
15: Remove $current$ from OpenList;
16: Add $current$ to ClosedList;
17: **for each** $neighbor$ in neighboring states of $current$ **do**
18: $g[neighbor] = h[neighbor] = estimate\_cost(neighbor)$;
19: $f[neighbor] = g[neighbor] + h[neighbor]$;
20: **if** $f[neighbor] >= upperBound$ **or** $neighbor$ is in ClosedList **then**
21: continue;
22: **if** $neighbor$ is not in OpenList **then**
23: Add $neighbor$ to OpenList;
24: Return D;

3.2.2 State Evaluation

The core operations of evaluating a state are to estimate the expected monetary cost (function $estimate\_cost$) and to evaluate the feasibility of a state (function $estimate\_performance$) whether it satisfies the probabilistic performance guarantee. Due to cloud performance dynamics, we develop probabilistic methods for the evaluation.

We model the execution time of tasks as probabilistic distributions. We develop probabilistic distribution models to describe the performance dynamics of I/O and network. Previous studies [12], [14] show that I/O and network are the major sources of performance dynamics in the cloud due to resource sharing while the CPU performance is rather stable for a given instance type. We define the probability of the I/O and network bandwidth equaling to a certain value x on instance type $type$ to

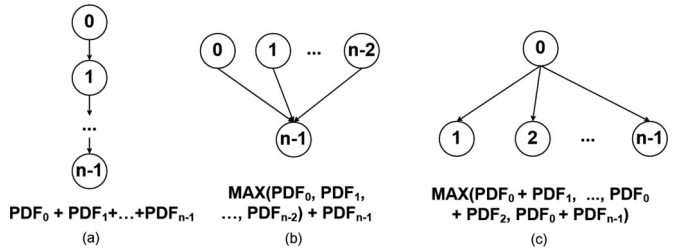


Fig. 5. Basic workflow structures and their probabilistic distributions of the execution time, denoting the execution time distribution of Task 0, 1,. . ., $n-1$ to be PDF_0, PDF_1,. . ., PDF_{n-1}, respectively.

be: $P_{seqBand,type}(seqBand = x)$, $P_{rndBand,type}(rndBand = x)$, $P_{inBand,type}(inBand = x)$ and $P_{outBand,type}(outBand = x)$ as the probabilistic distributions for the sequential I/O, random I/O, downloading and uploading network performance from/to the persistent storage, respectively. In our calibrations on Amazon EC2, $P_{rndBand,type}(rndBand = x)$ conforms to normal distributions and the other three conform to Gamma distributions (Section 4). Given the I/O and network performance distributions and the corresponding I/O and networking data size, we manage to model the execution time of a task on different instance types with probabilistic distribution functions (PDFs). For example, if the size of the input data on the disk is s_{in}, the probability of the time on reading the input data equalling to $\frac{s_{in}}{x}$ is $P_{seqBand,type}(seqBand = x)$, by assuming reading the input data is sequential accesses.

Having modeled the execution time of tasks as probabilistic distributions, we first introduce the implementation of function $estimate\_cost$. The monetary cost of a state s is estimated to be the sum of the expected monetary cost of each task running on the type of instance specified in s. Consider a task with on-demand instance type $type$ and on-demand price p. We estimate the expected monetary cost of the task to be p multiplied by the expected execution time of the task on the $type$-type on-demand instance. Here, we have ignored the rounding monetary cost in the estimation. This is because in the WaaS environment, this rounding monetary cost is usually amortized among many tasks. Enforcing the instance hour billing model could severely limit the optimization space, leading to a suboptimal solution (a configuration plan with suboptimal monetary cost).

Another core evaluation function is $estimate\_performance$. Given a state s and the execution time distribution of each task under the evaluated state s, we first calculate the execution time distribution of the entire workflow. Since the execution time of a task is now a distribution, rather than a static value, the execution time on the critical path is also dynamic. To have a complete evaluation, we apply a divide-and-conquer approach to get the execution time distribution of the entire workflow. Particularly, we decompose the workflow structure into the three kinds of basic structures, as shown in Fig. 5. Each basic structure has n tasks ($n \geq 2$). The decomposition is straightforward by identifying the basic structures in a recursive manner from the starting task(s) of the workflow.

The execution time distribution of each basic structure is calculated with the execution time distributions of individual tasks. For example, the execution time distribution of the structure in Fig. 5b is calculated as $MAX(PDF_0, PDF_1, \ldots, PDF_{n-2}) + PDF_{n-1}$, where PDF_i $(0 \leq i \leq n-1)$ is the probabilistic distribution of the execution time of task i. The "+" operation of two probabilistic distributions calculates the convolution of the two distributions and the MAX operation finds the distribution of the maximum of two random variables modeled by the two distributions. After obtaining the execution time distribution of the workflow, we check its percentile at the required probabilistic deadline guarantee. According to our notion of probabilistic deadline, only if the returned percentile is no longer than the deadline, the evaluated state is feasible.

3.3 Hybrid Instance Configuration Refinement

We consider the adoption of spot instances as a refinement to the configuration plan obtained from the previous step (the $A^\star$-based instance configuration algorithm) to further reduce monetary cost. The major problem of adopting spot instances is that, running a task on spot instances may suffer from the out-of-bid events and fail to meet the deadline requirements. We propose a simple yet effective hybrid instance configuration to tackle this issue. The basic idea is, if the deadline allows, we can try to run a task on a spot instance first. If the task can finish on the spot instance, the monetary cost tends to be lower than the monetary cost of running the task on an on-demand instance. It is possible that we can try more than one spot instances, if the previous spot instance fails (as long as it can reduce the monetary cost and satisfy the probabilistic performance guarantee). If all spot instances in the hybrid instance configuration fail, the task is executed on an on-demand instance to ensure the deadline. When a task finishes the execution on a spot instance, it is checkpointed, and the checkpoint is stored on the persistent storage of the cloud (such as Amazon S3). This is to avoid trigger the re-execution of its precedent tasks. Dyna performs checkpointing only when the task ends, which is simple and has much less overhead than the general checkpointing algorithms [15].

A hybrid instance configuration of a task is represented as a vector of both spot and on-demand instance types, as described in Section 2.2. The last dimension in the vector is the on-demand instance type obtained from the $A^\star$-based instance configuration step. The initial hybrid configuration contains only the on-demand instance type. Starting from the initial configuration, we repeatedly add spot instances at the beginning of the hybrid instance configuration to find better configurations. Ideally, we can add n spot instances (n is a predefined parameter). A larger n gives higher probability of benefiting from the spot instances while a smaller n gives higher probability of meeting deadline requirement and reduces the optimization overhead. In our experiments, we find that $n = 2$ is sufficient for obtaining good optimization results. A larger n greatly increases the optimization overhead with only very small improvement on the optimization results.

It is a challenging task to develop an efficient and effective approach for hybrid instance configuration refinement. First, coupled with the performance dynamics, it is a non-trivial task to compare whether one hybrid instance configuration is better than the other in terms of cost and performance. Second, since the cloud provider usually offers multiple instance types and a wide range of spot prices, we

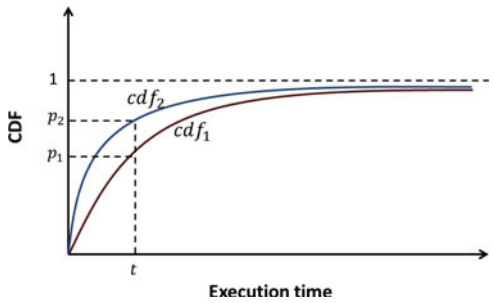


Fig. 6. The definition of configuration $C_2 \geq C_1$. cdf_1 and cdf_2 are the cumulative execution time distribution functions under configuration plan C_1 and C_2, respectively.

are facing a large space for finding the suitable spot instance type and spot price.

To address those two challenging issues, we develop efficient and effective heuristics to solve the problem. We describe the details in the remainder of this section. Refining a hybrid instance configuration C_{orig} of a task to a hybrid instance configuration $C_{refined}$, we need to determine whether $C_{refined}$ is better than C_{orig} in terms of monetary cost and execution time distributions. Particularly, we have the following two considerations. We accept the refined configuration $C_{refined}$ if both of the two considerations are satisfied.

1) *Probabilistic deadline guarantee consideration.* $C_{refined}$ should not violate the probabilistic deadline guarantee of the entire workflow;
2) *Monetary cost reduction.* The estimated monetary cost of $C_{refined}$ should be less than that of C_{orig}.

Probabilistic deadline guarantee consideration. A naive way is to first calculate the probabilistic distribution of the entire workflow's execution time under the refined configuration $C_{refined}$ and then to decide whether the probabilistic deadline requirement is met. However, this calculation introduces large overhead. We implement this process in the Oracle algorithm presented in Section 4. In Dyna, we propose a light-weight localized heuristic to reduce the overhead. As the on-demand configurations (i.e., the initial hybrid instance configuration) of each task found in the $A^\star$-based instance configuration step have already ensured the probabilistic deadline requirement, we only need to make sure that the refined hybrid instance configuration $C_{refined}$ of each task satisfies $C_{refined} \geq C_{orig}$, where $\geq$ is defined in Definition 1. Fig. 6 illustrates this definition. The integrals are represented as cumulative distribution functions (CDFs). With this heuristic, when evaluating the probabilistic deadline guarantee consideration for a refined configuration, we only need to calculate the probabilistic distribution of the execution time of a task rather than the entire workflow and thus greatly reduce the optimization overhead.

Definition 1. *Given two hybrid instance configurations C_1 and C_2 of task T, we have $C_2 \geq C_1$ if for $\forall t$, we have $\int_0^t P_{T,C_2}(time = x)\,\mathrm{d}x \geq \int_0^t P_{T,C_1}(time = x)\,\mathrm{d}x$, where P_{T,C_1} and P_{T,C_2} are the PDFs of task T under configuration C_1 and C_2, respectively.*

In order to compare two hybrid instance configurations according to Definition 1, we first discuss how to estimate the execution time distribution of a task given a hybrid instance configuration. Assume a hybrid instance configuration of task T is $C_{refined} = \langle (type_1, P_b, True), (type_2, P_o, False)\rangle$. Assume the probabilistic distributions of the execution time of task T on the spot instance of $type_1$ is $P_{T,type_1}$ and on the on-demand instance of $type_2$ is $P_{T,type_2}$. The overall execution time of task T under $C_{refined}$ can be divided into two cases. If the task successfully finishes on the spot instance (with probability p_{suc}), the overall execution time equals to the execution time of task T on the spot instance t_s with the following probability

$$P_{T,C_{refined}}(time = t_s) = P_{T,type_1}(time = t_s) \times p_{suc}. \quad (1)$$

Otherwise, the overall execution time equals to the time that task T has run on the spot instance before it fails, t_f, plus the execution time of task T on the on-demand instance t_o, with the following probability

$$\begin{aligned} P_{T,C_{refined}}(time = t_f + t_o) = {} & P_{T,type_1}(time = t_f) \\ & \times P_{T,type_2}(time = t_o) \\ & \times (1 - p_{suc}). \end{aligned} \quad (2)$$

Now we discuss how to calculate p_{suc}. Since a spot instance may fail at any time, we define a probabilistic function $ffp(t, p)$ to calculate the probability of a spot instance fails at time t for the first time when the bidding price is set to p. Existing studies have demonstrated that the spot prices can be predicted using statistics models [17] or reverse engineering [18]. We use the recent spot price history as a prediction of the real spot price for $ffp(t, p)$ to calculate the failing probability. We obtain that function with a Monte-Carlo based approach. Starting from a random point in the price history, if the price history becomes larger than p at time t for the first time, we add one to the counter $count$. We repeat this process for NUM times (NUM is sufficiently large) and return $\frac{count}{NUM}$ as the failing probability. Using the ffp function, we can define p_{suc} as follows

$$p_{suc} = 1 - \int_0^{t_s} ffp(x, P_b)\,\mathrm{d}x. \quad (3)$$

After obtaining the execution time distribution of a task under the refined hybrid instance configuration $C_{refined}$, we compare it with the configuration C_{orig} according to Definition 1. If $C_{refined} \geq C_{orig}$ is satisfied, the probabilistic deadline guarantee consideration is satisfied.

Monetary cost reduction. We estimate the monetary cost of a hybrid instance configuration of a task as the sum of the cost spent on the spot instance and the cost on the on-demand instance. Using Equation (1)-(3), we calculate the expected monetary cost of configuration $C_{refined}$ in Equation (4). Note that, we use the bidding price P_b to approximate the spot price in calculating the cost on spot instances. This calculation gives an upper bound of the actual expected monetary cost of the refined configuration and thus assures the correctness when considering the monetary cost reduction. If the estimated monetary cost of the refined configuration is lower than the monetary cost of the original configuration, the monetary cost reduction consideration is satisfied

$$cost(C_{refined}) = p_{suc} \times P_b \times t_s + (1 - p_{suc}) \times (P_b \times t_f + P_o \times t_o). \quad (4)$$

Repeatedly, we add spot instances to generate better hybrid instance configurations for each task in the workflow. Specifically, for each added spot instance, we decide its type and associated bidding price that satisfy the probabilistic deadline guarantee and monetary cost reduction considerations. Due to price dynamics of spot instances, making the decision is non-trivial. One straightforward way is that, we consider the cost of all spot instance types and its associated bidding price. The refined hybrid instance configuration is chosen as the one that has the smallest expected monetary cost and satisfies the probabilistic performance guarantee. However, this method needs to search for a very large solution space. To reduce the search space, we design a heuristic as described in Algorithm 2. We notice that, the added spot instance type should be at least as expensive as (i.e., the capability should be at least as good as) the on-demand instance type found in the $A^\star$ search step in order to ensure the probabilistic deadline guarantee. Thus, instead of searching all spot instance types, we only need to evaluate the types that are equally or more expensive than the given on-demand instance type. For each evaluated spot instance type, we search the bidding price using the binary search algorithm described in Algorithm 3.

Algorithm 2. Hybrid Instance Configuration Refinement for a Task T.

Require: $type_o$: the on-demand instance type found in the $A^\star$ instance configuration for task T
n: the dimension of the hybrid instance configuration
1: T.configList[n] = $type_o$;
2: T.prices[n] = on-demand price of $type_o$;
3: **for** $dim = 1$ **to** $n - 1$ **do**
4: T.configList[dim] = -1;
5: T.prices[dim] = 0;
6: **for** $dim = 1$ **to** $n - 1$ **do**
7: **for** $type_s = type_o$ **to** the most expensive instance type **do**
8: P_{max} = the on-demand price of instance type $type_s$;
9: P_b = binary_search(P_{min}, P_{max}, $type_s$);
10: **if** $P_b == -1$ **then**
11: continue;
12: **else**
13: T.configList[dim] = $type_s$;
14: T.prices[dim] = P_b;

The binary search algorithm in Algorithm 3 is illustrated as follows. If the probabilistic deadline guarantee consideration is not satisfied, it means the searched spot price is too low and we continue the search in the higher half of the search space (Line 10-12). If the monetary cost reduction consideration is not met, it means the searched spot price is too high and we continue the search in the lower half of the search space (Line 6-8). If both considerations are satisfied for a certain bidding price, this price is used as the bidding price in the hybrid instance configuration. We search for the bidding price in the range of $[P_{low}, P_{high}]$. In our implementation, P_{low} is 0.001 and P_{high} equals to the on-demand price of the evaluated spot instance type. Note, the spot instance with bidding price higher than P_{high} does not contribute to monetary cost reduction.

Algorithm 3. Binary_Search(P_{low}, P_{high}, $type_s$) for a Task T.

Require: P_{low}: the lowest bidding price searched
P_{high}: the highest bidding price searched
$type_s$: the evaluated spot instance type
C_{orig}: the hybrid configuration before adding the spot instance of type $type_s$
$C_{refined}$: the refined hybrid configuration with the spot instance of type $type_s$ added
1: **if** $P_{low} > P_{high}$ **then**
2: Return -1;
3: $P_{mid} = (P_{low} + P_{high})/2$;
4: $originalcost = estimate\_cost(C_{orig})$;
5: $C_{refined} = \langle (type_s, P_{mid}, True), C_{orig} \rangle$;
6: $refinedcost = estimate\_cost(C_{refined})$;
7: **if** $refinedcost > originalcost$ **then**
8: Return binary_search($P_{low}, P_{mid}, type_s$);
9: **else**
10: $satisfied = estimate\_performance(C_{refined})$;
11: **if not** $satisfied$ **then**
12: Return binary_search($P_{mid}, P_{high}, type_s$);
13: Return P_{mid};

4 Evaluation

In this section, we present the evaluation results of the proposed approach on Amazon EC2 and a cloud simulator.

4.1 Experimental Setup

We have two sets of experiments: firstly calibrating the cloud dynamics from Amazon EC2 as the input of our optimization system; secondly running scientific workflows on Amazon EC2 and a cloud simulator with the compared algorithms for evaluation.

Calibration. We measure the performance of CPU, I/O and network for four frequently used instance types, namely m1.small, m1.medium, m1.large and m1.xlarge. We find that CPU performance is rather stable, which is consistent with the previous studies [12]. Thus, we focus on the calibration for I/O and network performance. In particular, we repeat the performance measurement on each kind of instance for 10,000 times (once every minute in seven days). When an instance has been acquired for a full hour, it will be released and a new instance of the same type will be created to continue the measurement. The measurement results are used to model the probabilistic distributions of I/O and network performance.

We measure both sequential and random I/O performance for local disks. The sequential I/O reads performance is measured with *hdparm*. The random I/O performance is measured by generating random I/O reads of 512 bytes each. Reads and writes have similar performance results, and we do not distinguish them in this study.

We measure the uploading and downloading bandwidth between different types of instances and Amazon S3. The bandwidth is measured from uploading and downloading a file to/from S3. The file size is set to 8 MB. We also measured the network bandwidth between two instances using

Iperf [38]. We find that the network bandwidth between instances of different types is generally lower than that between instances of the same type and S3.

Workflows. There have been some studies on characterizing the performance behaviours of scientific workflows [19]. In this paper, we consider three common workflow structures, namely Ligo, Montage and Epigenomics. The three workflows have different structures and parallelism.

We create instances of Montage workflows using Montage source code. The input data is the 2MASS J-band images covering 8-degree by 8-degree areas retrieved from the Montage archive. The number of tasks in the workflow is 10,567. The input data size is 4 GB, where each of the 2,102 tasks on the first level of the workflow structure reads an input image of 2 MB. Initially, the input data is stored in Amazon S3 storage. Since Ligo and Epigenomics are not open-sourced, we construct synthetic Ligo and Epigenomics workflows using the workflow generator provided by Pegasus [39]. We use the DAX files with 1,000 and 997 tasks (Inspiral_1000.xml and Epigenomics_997.xml [39]) for Ligo and Epigenomics, respectively. The input data size of Ligo is 9.3 GB, where each of the 229 tasks on the first level of the workflow structure reads 40.5 MB of input data on average. The input data size of Epigenomics is 1.7 GB, where each of the seven tasks on the first level of the workflow structure reads 369 MB of DNA sequence data on average.

Implementations. In order to evaluate the effectiveness of the proposed techniques in Dyna, we have implemented the following algorithms.

- *Static.* This approach is the same as the previous study in [1] which only adopts on-demand instances. We adopt it as the state-of-the-art comparison. For a fair comparison, we set the workflow deadline according to the probabilistic QoS setting used in Dyna. For example, if the user requires 90 percent of probabilistic deadline guarantee, the deterministic deadline used for Static is set to the 90th percentile of the workflow's execution time distribution.
- *DynaNS.* This approach is the same as Dyna except that DynaNS does not use any spot instances. The comparison between Dyna and DynaNS is to assess the impact of spot instances.
- *SpotOnly.* This approach adopts only spot instances during execution. It first utilizes the $A^\star$-based instance configuration approach to decide the instance type for each task in the workflow. Then we set the bidding price of each task to be very high (in our studies, we set it to be \$1, 000) in order to guarantee the probabilistic deadline requirement.
- *Oracle.* We implement the Oracle method to assess the trade-off between the optimization overhead and the effectiveness of the optimizations in Dyna. Oracle is different from Dyna in that, Oracle does not adopt the localized heuristic as Definition 1 (Section 3.3) when evaluating the probabilistic deadline guarantee consideration. This is an offline approach, since the time overhead of getting the solution in Oracle is prohibitively high.
- *MOHEFT.* We select a state-of-the-art multi-objective approach [40] for comparison. According to the previous study [40], MOHEFT is able to search the instance configuration space and obtain a set of non-dominated solutions on the monetary cost and execution time.

We conduct our experiments on both real clouds and simulator. These two approaches are complementary, because some scientific workflows (such as Ligo and Epigenomics) are not publicly available. Specifically, when the workflows (including the input data and executables, etc.) are publically available, we run them on public clouds. Otherwise, we simulate the execution with synthetic workflows according to the workflow characteristics from existing studies [19].

On Amazon EC2, we adopt a popular workflow management system (Pegasus [41]) to manage the execution of workflows. We create an Amazon Machine Image (AMI) installed with Pegasus and its prerequisites such as DAGMan [42] and Condor [43]. We modify the Pegasus (release 4.3.2) scheduler to enable scheduling the tasks onto instances according to the hybrid instance configurations. A script written with Amazon EC2 API is developed for acquiring and releasing instances at runtime.

We develop a simulator based on CloudSim [44]. We mainly present our new extensions, and more details on cloud simulations can be found in the original paper [44]. The simulator includes three major components, namely Cloud, Instance and Workflow. The Cloud component maintains a pool of resources which supports acquisition and release of Instance components. It also maintains the I/O and network performance histograms measured from Amazon EC2 to simulate cloud dynamics. A spot price trace obtained from the Amazon EC2 history is also maintained to simulate the price dynamics. The Instance component simulates the on-demand and spot instances, with cloud dynamics from the calibration. We simulate the cloud dynamics in the granularity of seconds, which means the average I/O and network performance per second conform the distributions from calibration. The Workflow component manages the workflow structures and the scheduling of tasks onto the simulated instances.

Experimental settings. We acquire the four measured types of instances from the US East region using the created AMI. The hourly costs of the on-demand instance for the four instance types are shown in Table 1. Those four instances have also been used in the previous studies [15]. As for the instance acquisition time (lag), our experiments show that each on-demand instance acquisition costs 2 minutes and spot instance acquisition costs 7 minutes on average. This is consistent with the existing studies [45].

The deadline of workflows is an important factor for the candidate space of determining the instance configuration. There are two deadline settings with particular interests: D_{min} and D_{max}, the expected execution time of all the tasks in the critical path of the workflow all on the m1.xlarge and m1.small instances, respectively. By default, we set the deadline to be $\frac{D_{min}+D_{max}}{2}$.

We assume there are many workflows submitted by the users to the WaaS provider. In each experiment, we submit 100 jobs of the same workflow structure to the cloud. We assume the job arrival conforms to a Poisson distribution.

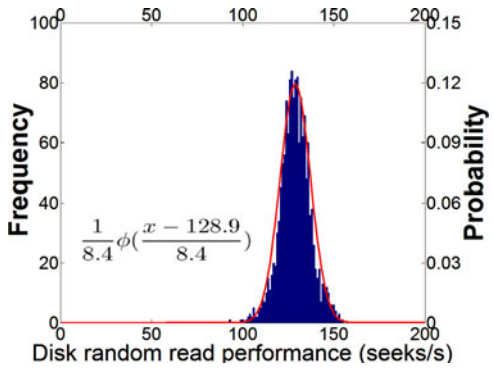


Fig. 7. The histogram and probabilistic distribution of random I/O performance on m1.medium instances.

The parameter λ in the Poisson distribution affects the chance for virtual machine reuse. By default, we set λ as 0.1.

As for metrics, we study the average monetary cost and elapsed time for a workflow. *All the metrics in the figures in Section 4.3 and 4.4 are normalized to those of Static.* Given the probabilistic deadline requirement, we run the compared algorithms multiple times on the cloud and record their monetary cost and execution time. We consider monetary cost as the main metric for comparing the optimization effectiveness of different scheduling algorithms when they all satisfy the QoS requirements. By default, we set the probabilistic deadline requirement as 96 percent. By default, we present the results obtained when all parameters are set to their default setting. In Section 4.4, we experimentally study the impact of different parameters with sensitivity studies.

4.2 Cloud Dynamics

In this section, we present the performance dynamics observed on Amazon EC2. The price dynamics have been presented in Table 1 of Section 2.

Figs. 7 and 8 show the measurements of random I/O performance and downloading network performance from Amazon S3 of m1.medium instances. We have observed similar results on other instance types. We make the following observations.

First, both I/O and network performances can be modeled with normal or Gamma distributions. We verify the distributions with null hypothesis, and find that (1) the sequential I/O performance, and uploading and downloading network bandwidth from/to S3 of the four instance types follow the Gamma distribution; (2) the random I/O

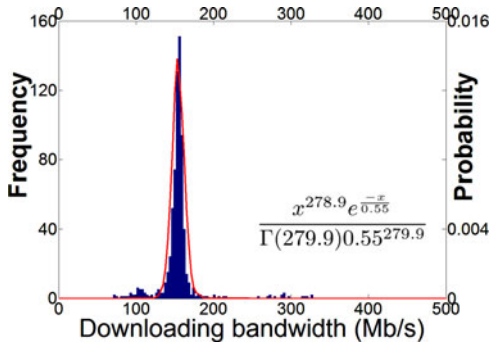


Fig. 8. The histogram and probability distribution of downloading bandwidth between m1.medium instances and S3 storage.

TABLE 2
Parameters of I/O Performance Distributions

Instance type	Sequential I/O (Gamma)	Random I/O (Normal)
m1.small	$k = 129.3, \theta = 0.79$	$\mu = 150.3, \sigma = 50.0$
m1.medium	$k = 127.1, \theta = 0.80$	$\mu = 128.9, \sigma = 8.4$
m1.large	$k = 376.6, \theta = 0.28$	$\mu = 172.9, \sigma = 34.8$
m1.xlarge	$k = 408.1, \theta = 0.26$	$\mu = 1{,}034.0, \sigma = 146.4$

performance distributions on the four instance types follow the normal distribution. The parameters of those distributions are presented in Tables 2 and 3. Those results are mainly based on the measurement on real clouds. It is the result when different network and disk I/O pattern interplayed with the shared virtualization environments. However, we do not know the underlying reason that the random disk I/O performance follows the normal distribution and other patterns follow the Gamma distribution.

Second, the I/O and network performance of the same instance type varies significantly, especially for m1.small and m1.medium instances. This can be observed from the θ parameter of Gamma distributions or the σ parameter of normal distributions in Tables 2 and 3. Additionally, random I/O performance varies more significantly than sequential I/O performance on the same instance type. The coefficient of variance of sequential and random I/O performance on m1.small are 9 and 33 percent, respectively. That indicates the necessity of capturing the performance dynamics into our performance guarantee.

Third, the performance between different instance types also differ greatly from each other. This can be observed from the $k \cdot \theta$ parameter (the expected value) of Gamma distributions or the μ parameter of normal distributions in Tables 2 and 3. Due to the significant differences among different instance types, we need to carefully select the suitable instance types so that the monetary cost is minimized.

Finally, we observe that the performance distributions of the on-demand instance types are the same as or very close to those of the spot instance types.

4.3 Overall Comparison

In this subsection, we present the overall comparison results of Dyna and the other compared algorithms on Amazon EC2 and the cloud simulator under the default settings. Sensitivity studies are presented in Section 4.4. Note that we have used the calibrations from Section 4.2 as input to Dyna for performance and cost estimations.

Fig. 9 shows the average monetary cost per job results of Static, DynaNS, SpotOnly, Dyna and Oracle methods on the Montage, Ligo and Epigenomics workloads. The standard

TABLE 3
Gamma Distribution Parameters on Bandwidth between an Instance and S3

Instance type	Uploading bandwidth	Downloading bandwidth
m1.small	$k = 107.3, \theta = 0.55$	$k = 51.8, \theta = 1.8$
m1.medium	$k = 421.1, \theta = 0.27$	$k = 279.9, \theta = 0.55$
m1.large	$k = 571.4, \theta = 0.22$	$k = 6{,}187.7, \theta = 0.44$
m1.xlarge	$k = 420.3, \theta = 0.29$	$k = 15{,}313.4, \theta = 0.23$

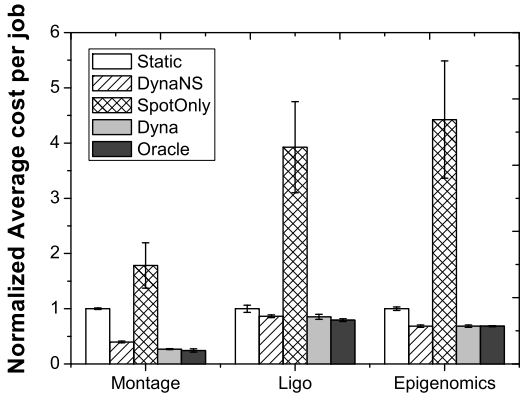


Fig. 9. The normalized average monetary cost optimization results of compared algorithms on Montage, Ligo and Epigenomics workflows.

errors of the monetary cost results of Static, DynaNS, SpotOnly, Dyna and Oracle are 0.01-0.06, 0.02-0.04, 0.40-0.76, 0.01-0.03 and 0.01-0.03, respectively, on the tested workloads. The absolute values of the average monetary cost of Static are $285, $476 and $214 for Montage, Ligo and Epigenomics, respectively. Overall, Dyna obtains the smallest monetary cost among the online approaches in all three workloads, saving monetary cost over Static, DynaNS and SpotOnly by 15-73, percent 1-33 percent and 78-85 percent, respectively. We make the following observations.

First, DynaNS obtains smaller monetary cost than Static, because the proposed $A^{\star}$ configuration search technique is capable of finding cheaper instance configurations and is suitable for different structures of workflows. This also shows that performing deadline assignment before instance configuration in the Static algorithm reduces the optimization effectiveness. For example, with the deadline assignment approach, the instance configuration of a task has to make sure that its execution time is no longer than its assigned sub-deadline. However, this task can actually make use of the left-over time from its previous tasks and be assigned to a cheaper instance type.

Second, Dyna obtains smaller monetary cost than DynaNS, meaning that the hybrid configuration with both spot and on-demand instances is effective on reducing monetary cost, in comparison with the on-demand only approach. For lower probabilistic deadline guarantees, the monetary cost saved by Dyna over DynaNS gets higher

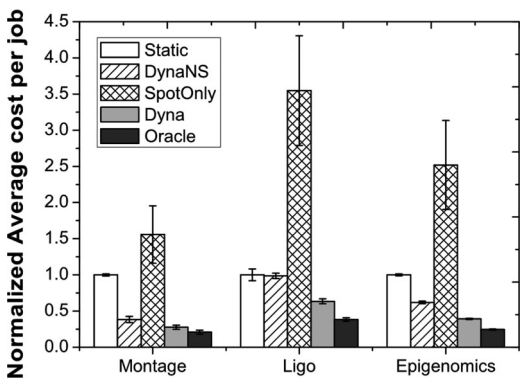
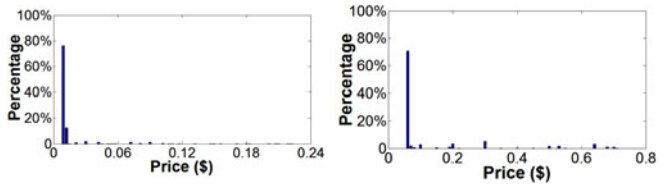


Fig. 10. The normalized average monetary cost results of compared algorithms on Montage, Ligo and Epigenomics workflows when the probabilistic deadline guarantee is 90 percent.



(a) Histogram of the spot price of m1.small instance type

(b) Histogram of the spot price of m1.xlarge instance type

Fig. 11. Histogram of the spot price history in August 2013, US East Region of Amazon EC2.

because the opportunity for leveraging spot instances gets higher. Fig. 10 shows the monetary cost results of the compared algorithms when the probabilistic deadline guarantee is set to 90 percent. In this setting, the monetary cost reduction of Dyna over DynaNS is even higher than the default setting, by 28-37 percent.

Third, SpotOnly obtains the highest monetary cost among all the compared algorithms. This is due to the dynamic characteristic of spot price. Fig. 11 shows the histogram of the spot price during the month of the experiments. Although the spot price is lower than the on-demand price of the same type in most of the time, it can be very high compared to on-demand price at some time. As shown in Table 1, the highest spot price for a m1.small instance in August 2013 is $10 which is more than 160 times higher than the on-demand price. Nevertheless, this observation depends on the fluctuation of spot price. The results on comparing SpotOnly and Dyna can be different if we run the experiments at other times. We study the sensitivity of Dyna and SpotOnly to spot price with another spot price history in Section 4.4.

Fig. 12 shows the average execution time of a workflow of Static, DynaNS, SpotOnly, Dyna and Oracle methods on the Montage, Ligo and Epigenomics workloads. The standard errors of the execution time results of the compared algorithms are between 0.01-0.06 on the tested workloads. Static has the smallest average execution time, which are around 3.4, 6.3 and 2.5 hours for Montage, Ligo and Epigenomics, respectively. This is because Static configures each task in workflows with better and more expensive instance types. The careful selection of bidding price for each task in the workflow in Dyna and high bidding prices in SpotOnly diminish the out-of-bid events during execution. All of

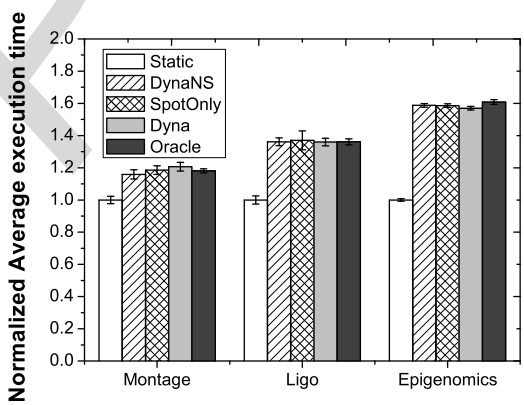
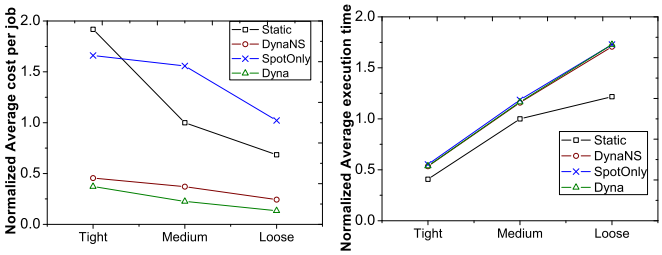


Fig. 12. The normalized average execution time optimization results of compared algorithms on Montage, Ligo and Epigenomics workflows.

TABLE 4
Optimization Overhead of the Compared Algorithms on Montage, Ligo and Epigenomics Workflows (Seconds)

	Static	DynaNS	SpotOnly	Dyna	Oracle
Montage	1	153	153	163	2,997
Ligo	1	236	236	244	10,452
Epigenomics	1	166	166	175	2,722



(a) Average monetary cost per job (b) Average execution time

Fig. 13. The normalized average monetary cost and average execution time results of sensitivity studies on deadline.

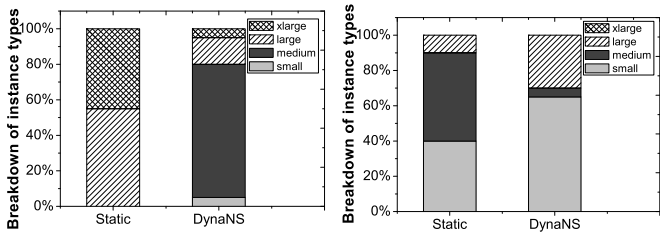
DynaNS, SpotOnly, Dyna and Oracle are able to guarantee the probabilistic deadline requirement.

Finally, we analyze the optimization overhead of the compared algorithms. The optimization overhead results are shown in Table 4. Note that, for workflows with the same structure and profile, our system only need to do the optimization once. Although Oracle obtains smaller monetary cost than Dyna, the optimization overhead of Oracle is 16-44 times as high as that of Dyna. This shows that Dyna is able to find optimization results close to the optimal results in much shorter time. Due to the long execution time of the Oracle optimization, in the rest of the experiments, we do not evaluate Oracle but only compare Dyna with Static, DynaNS and SpotOnly.

4.4 Sensitivity Studies

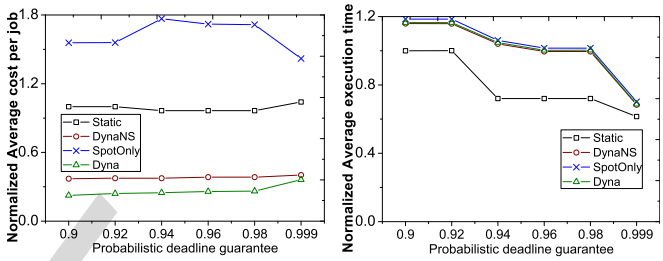
We have conducted sensitivity studies on different workflows. Since we observed similar results across workflows, we focus on Montage workflows in the following. In each study, we vary one parameter at a time and keep other parameters in their default settings.

Deadline. Deadline is an important factor for determining the instance configurations. We evaluate the compared algorithms under deadline requirement varying from $1.5 \times D_{min}$ (denoted as "Tight"), $0.5 \times (D_{min} + D_{max})$ (denoted as "Medium") to $0.75 \times D_{max}$ (denoted as "Loose"). All results are normalized to those of Static when the deadline is Medium. Fig. 13 shows the average monetary cost per job and average execution time results. Dyna obtains the smallest average monetary cost among the compared algorithms under all tested deadline settings. As the deadline gets loose, the monetary cost is decreased since more cheaper instances (on-demand instances) are used for execution. We break down the number of different types of on-demand instances when the deadlines are Tight and Loose as shown in Fig. 14. When the deadline is Loose, more cheap instances are utilized. Under the same deadline, e.g., Tight, DynaNS utilizes more cheap instances than Static, which again shows our $A^\star$ approach is better than the existing heuristics [1] in



(a) Breakdown when deadline is Tight (b) Breakdown when deadline is Loose

Fig. 14. Breakdown of the instance types adopted by compared algorithms when the deadlines are Tight and Loose.



(a) Average monetary cost per job (b) Average execution time

Fig. 15. The normalized average monetary cost and average execution time results of sensitivity studies on the probabilistic deadline guarantees.

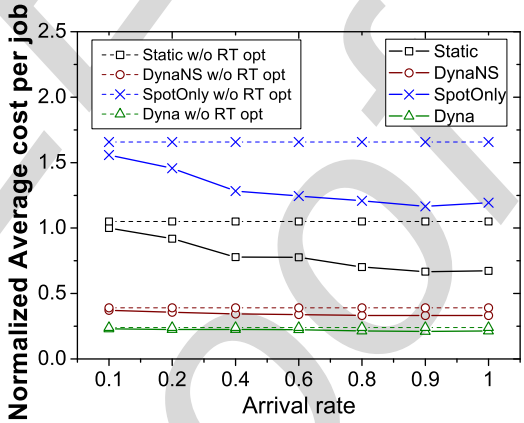


Fig. 16. The normalized average monetary cost results of sensitivity studies on the arrival rate of workflows.

the previous study. This trend does not apply to SpotOnly because the spot price of the m1.medium instance can be lower than the m1.small instance at some time. We have validated this phenomena with studying the spot price trace.

Probabilistic deadline guarantee. We evaluate the effectiveness of Dyna on satisfying probabilistic deadline requirements when the requirement varies from 90 to 99.9 percent. Fig. 15 shows the average monetary cost per job and average execution time results of the compared algorithms. Dyna achieves the smallest monetary cost for different probabilistic deadline guarantee settings. With a lower probabilistic deadline requirement, the monetary cost saved by Dyna is higher. DynaNS, SpotOnly and Dyna can guarantee the probabilistic deadline requirement under all settings.

Arrival rate. We evaluate the effectiveness of Dyna when the arrival rate λ of workflows varies from 0.1, 0.2, 0.4, 0.6, 0.8, 0.9 to 1.0. All results are normalized to those when arrival rate is 0.1. Fig. 16 shows the optimized average monetary cost per job. Dyna obtains the smallest average

TABLE 5
Statistics on Spot Prices ($/hour, December 2011, Asia Pacific Region) and On-Demand Prices of Amazon EC2

Instance type	Average	*stdev*	Min	Max	OnDemand
m1.small	0.041	0.003	0.038	0.05	0.06
m1.medium	0.0676	0.003	0.064	0.08	0.12
m1.large	0.160	0.005	0.152	0.172	0.24
m1.xlarge	0.320	0.009	0.304	0.336	0.48

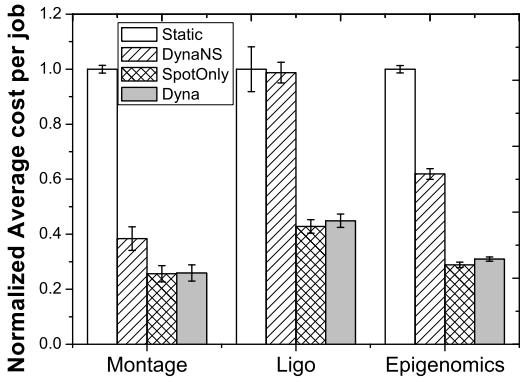


Fig. 17. The simulation result of the normalized average monetary cost obtained by the compared algorithms, using the spot price history of the Asia Pacific Region of Amazon EC2 in December, 2011.

monetary cost under all job arrival rates. As the job arrival rate increases, the average cost per job is decreasing. This is because the runtime optimizations that we adopt from the previous study [1], including consolidation and instance reuse, can enable resource sharing between workflow jobs. When the arrival rate increases, there are more jobs arriving in the WaaS at the same time. The resource utilization of the WaaS is increased and the partial instance time is better utilized. The dashed lines in Fig. 16 indicates the average monetary cost of the compared algorithms without the runtime optimizations. For the arrival rates that we have evaluated, the instance configuration is still essential for the monetary cost reduction in Dyna, in comparison with runtime consolidations and instance reuse.

Spot price. To study the sensitivity of Dyna and SpotOnly to the spot price variance, we use simulations to study the compared algorithms on different spot price histories. Particularly, we study the compared algorithms with the spot price history of the Asia Pacific Region in December 2011. As shown in Table 5, the spot price during this period is very low and stable, in comparison with the period that we performed the experiments in August 2013. Thus the spot instances are less likely to fail during the execution (the failing probability ffp is rather low). Fig. 17 shows the obtained monetary cost result. SpotOnly and Dyna obtain similar monetary cost results, which are much lower than Static and DynaNS. This demonstrates that Dyna is able to obtain good monetary cost optimization results for different spot price distributions.

4.5 Comparison with Multi-Objective Method

Finally, we present the comparison results of Dyna with MOHEFT [40] on the Epigenomics workflows with simulations. For each solution obtained from MOHEFT, we use its expected execution time as the deadline constraint for Dyna, and then we compare the monetary cost between Dyna and MOHEFT. All other parameters are set as default. All results are normalized to the lowest monetary cost obtained by MOHEFT.

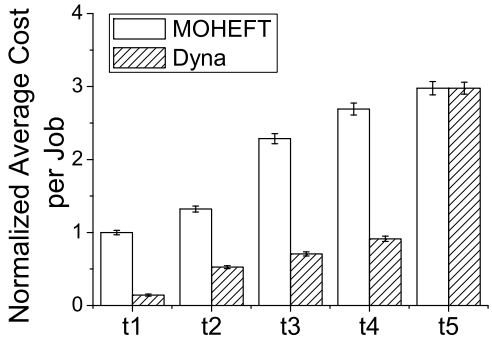


Fig. 18. Simulation results of Dyna and MOHEFT with Epigenomics workflow.

Fig. 18 shows the normalized average monetary cost results of the compared algorithms. In this experiment, MOHEFT outputs five different solutions. We denote the average execution time of different MOHEFT solutions to be t1, t2,..., t5. Dyna obtains smaller monetary cost than MOHEFT due to the utilization of spot instances, and can ensure the deadline constraint under all settings. Although MOHEFT optimizes both the monetary cost and execution time as a multi-objective optimization problem, none of its solutions dominates the solution of Dyna.

5 Conclusions

As the popularity of various scientific and data-intensive applications in the cloud, hosting WaaS in IaaS clouds becomes emerging. However, the IaaS cloud is a dynamic environment with performance and price dynamics, which make the assumption of static task execution time and the QoS definition of deterministic deadlines undesirable. In this paper, we propose the notion of probabilistic performance guarantees as QoS to explicitly expose the cloud dynamics to users. We develop a workflow scheduling system named Dyna to minimize the monetary cost for the WaaS provider while satisfying predefined probabilistic deadline guarantees for individual workflows. We develop an $A^\star$ search based instance configuration method to address the performance dynamics, and hybrid instance configuration of both spot and on-demand instances for price dynamics. We deploy Dyna on both Amazon EC2 and simulator and evaluate its effectiveness with three scientific workflow applications. Our experimental results demonstrate that Dyna achieves much lower monetary cost than the state-of-the-art approaches (by 73 percent) while guaranteeing users' probabilistic deadline requirements.

Acknowledgments

The authors would like to thank anonymous reviewers for their valuable comments. The authors acknowledge the support from the Singapore National Research Foundation under its Environmental & Water Technologies Strategic Research Programme and administered by the Environment & Water Industry Programme Office (EWI) of the PUB,

under project 1002-IRIS-09. This work is partly supported by a MoE AcRF Tier 1 grant (MOE 2014-T1-001-145) in Singapore. Amelie Chi Zhou is also with Nanyang Environment and Water Research Institute (NEWRI). Amelie Chi Zhou is the corresponding author.

REFERENCES

[1] 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.*, 2011, pp. 1–12

[2] 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.*, 2012, pp. 1–11.

[3] A. C. Zhou, B. He, and S. Ibrahim, "A taxonomy and survey on escience as a service in the cloud," *Arxiv Preprint Arxiv:1407.7360*, 2014.

[4] J. Yu, R. Buyya, and C. K. Tham, "Cost-based scheduling of scientific workflow application on utility grids," in *Proc. 1st Int. Conf. E-Science Grid Comput.*, 2005, pp. 8–147.

[5] R. Sakellariou, H. Zhao, E. Tsiakkouri, and M. D. Dikaiakos, "Scheduling workflows with budget constraints," in *Proc. CoreGRID*, 2007, pp. 189–202.

[6] R. Duan, R. Prodan, and T. Fahringer, "Performance and cost optimization for multiple large-scale grid workflow applications," in *Proc. ACM/IEEE Conf. Supercomput.*, 2007.

[7] S. Abrishami, M. Naghibzadeh, and D. H. J. Epema, "Deadline-constrained workflow scheduling algorithms for IaaS clouds," *Future Generation Comput. Syst.*, vol. 29, pp. 15–169, 2013.

[8] E.-K. Byun, Y.-S. Kee, J.-S. Kim, and S. Maeng, "Cost optimized provisioning of elastic resources for application workflows," *Future Gen. Comput. Syst.*, vol. 27, pp. 1011–1026, 2011.

[9] S. Maguluri, R. Srikant, and L. Ying, "Stochastic models of load balancing and scheduling in cloud computing clusters," in *Proc. IEEE INFOCOM*, 2012, pp. 702–710.

[10] F. Zhang, J. Cao, K. Hwang, and C. Wu, "Ordinal optimized scheduling of scientific workflows in elastic compute clouds," in *Proc. IEEE 3rd Int. Conf. Cloud Comput. Technol. Sci.*, 2011, pp. 9–17.

[11] A. C. Zhou and B. He, "Simplified resource provisioning for workflows in IaaS clouds," in *Proc. IEEE 6th Int. Conf. Cloud Comput. Technol. Sci.*, 2014, pp. 650–655.

[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, pp. 460–471, 2010.

[13] A. Iosup, S. Ostermann, N. Yigitbasi, R. Prodan, T. Fahringer, and D. Epema, "Performance analysis of cloud computing services for many-tasks scientific computing," *IEEE Trans. Parallel Distrib. Syst.*, vol. 22, no. 6, pp. 931–945, Jun. 2011.

[14] H. Wang, Q. Jing, R. Chen, B. He, Z. Qian, and L. Zhou, "Distributed systems meet economics: pricing in the cloud," in *Proc. HotCloud*, 2010, pp. 1–7.

[15] S. Yi, A. Andrzejak, and D. Kondo, "Monetary cost-aware checkpointing and migration on amazon cloud spot instances," *IEEE Trans. Services Comput.*, vol. 5, no. 4, pp. 512–524, 4th Quarter 2012.

[16] M. Mazzucco and M. Dumas, "Achieving performance and availability guarantees with spot instances," in *Proc. IEEE 13th Int. Conf. High Perform. Commun.*, 2011, pp. 296–303.

[17] B. Javadi, R. Thulasiram, and R. Buyya, "Statistical modeling of spot instance prices in public cloud environments," in *Proc. IEEE 4th Int. Utility Cloud Comput.*, 2011, pp. 219–228.

[18] O. Agmon Ben-Yehuda, M. Ben-Yehuda, A. Schuster, and D. Tsafrir, "Deconstructing amazon EC2 spot instance pricing," in *Proc. IEEE 3rd Int. Conf. Cloud Comput. Technol. Sci.*,, 2011, pp. 304–311.

[19] G. Juve, A. Chervenak, E. Deelman, S. Bharathi, G. Mehta, and K. Vahi, "Characterizing and profiling scientific workflows," *Future Gen. Comput. Syst.*, vol. 29, pp. 682–692, 2013.

[20] L. Abeni and G. Buttazzo, "QoS guarantee using probabilistic deadlines," in *Proc. Euromicro Conf. Real-Time Syst.*, 1999, pp. 242–249.

[21] R. N. Calheiros and R. Buyya, "Meeting deadlines of scientific workflows in public clouds with tasks replication," *IEEE Trans. Parallel Distrib. Syst.*, 2013, pp. 1786–1796.

[22] H. Kloh, B. Schulze, R. Pinto, and A. Mury, "A bi-criteria scheduling process with CoS support on grids and clouds," *Concurrency Computat. Pract. Exp.*, vol. 24, pp. 1443–1460, 2012.

[23] I. M. Sardiña, C. Boeres, and L. M. De A. Drummond, "An efficient weighted bi-objective scheduling algorithm for heterogeneous systems," in *Proc. Int. Conf. Parallel Process.*, 2009, pp. 102–111.

[24] C. Lin and S. Lu, "Scheduling scientific workflows elastically for cloud computing," in *Proc. IEEE Int. Conf. Cloud Comput.*, 2011, pp. 746–747.

[25] S. Di, C.-L. Wang and F. Cappello, "Adaptive algorithm for minimizing cloud task length with prediction errors," *IEEE Trans. Cloud Comput.*, vol. 2, no. 2, pp. 194–207, Apr.–Jun. 2014.

[26] M. 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] D. de Oliveira, V. Viana, E. Ogasawara, K. Ocana, and M. Mattoso, "Dimensioning the virtual cluster for parallel scientific workflows in clouds," in *Proc. 4th ACM Workshop Sci. Cloud Comput.*, 2013, pp. 5–12.

[28] D. Oliveira, K. A. Ocaña, F. Baião, and M. Mattoso, "A provenance-based adaptive scheduling heuristic for parallel scientific workflows in clouds," *J. Grid Comput.*, vol. 10, pp. 521–552, 2012.

[29] N. Roy, A. Dubey, and A. Gokhale, "Efficient autoscaling in the cloud using predictive models for workload forecasting," in *Proc. IEEE Int. Conf. Cloud Comput.*, 2011, pp. 500–507.

[30] J. Yang, J. Qiu, and Y. Li, "A profile-based approach to just-in-time scalability for cloud applications," in *Proc. IEEE Int. Conf. Cloud Comput.*, 2009, pp. 9–16.

[31] A. C. Zhou and B. He, "Transformation-based monetary cost optimizations for workflows in the cloud," *IEEE Trans. Cloud Comput.*, vol. 2, no. 1, pp. 85–98, Jan.–Mar. 2013.

[32] S. Ostermann and R. Prodan, "Impact of variable priced cloud resources on scientific workflow scheduling," in *Proc. 18th Int. Conf. Parallel Process.*, 2012, pp. 350–362.

[33] B. Javadi, R. K. Thulasiram, and R. Buyya, "Characterizing spot price dynamics in public cloud environments," *Future Gen. Comput. Syst.*, vol. 29, pp. 988–999, 2013.

[34] H.-Y. Chu and Y. Simmhan, "Cost-efficient and resilient job lifecycle management on hybrid clouds," in *Proc. IEEE 28th Int. Parallel Distrib. Process. Symp.*, 2014, pp. 327–336.

[35] H. Ballani, P. Costa, T. Karagiannis, and A. Rowstron, "Towards predictable datacenter networks," in *Proc. ACM SIGCOMM Conf.*, 2011, pp. 242–253.

[36] M. Hovestadt, O. Kao, A. Kliem, and D. Warneke, "Evaluating adaptive compression to mitigate the effects of shared I/O in clouds," in *Proc. IEEE Int. Symp. Parallel Distrib. Process. Workshops Phd Forum*, 2011, pp. 1042–1051.

[37] S. Ibrahim, H. Jin, L. Lu, B. He, and S. Wu, "Adaptive disk i/o scheduling for MapReduce in virtualized environment," in *Proc. Int. Conf. Parallel Process.*, 2011, pp. 335–344.

[38] Iperf [Online]. Available: http://iperf.sourceforge.net, Jul. 2014.

[39] Workflow Generator. (2014, Jul.) [Online]. Available: https://confluence.pegasus.isi.edu/display/pegasus/WorkflowGenerator

[40] J. J. Durillo, R. Prodan, and H. M. Fard, "MOHEFT: A multi-objective list-based method for workflow scheduling," in *Proc. IEEE 4th Int. Conf. Cloud Comput. Technol. Sci.*, 2012, pp. 185–192.

[41] E. Deelman, G. Singh, M.-H. Su, J. Blythe, Y. Gil, C. Kesselman, G. Mehta, K. Vahi, G. B. Berriman, J. Good, A. Laity, J. C. Jacob, and D. S. Katz, "Pegasus: A framework for mapping complex scientific workflows onto distributed systems," *Sci. Program.*, vol. 13, pp. 219–237, 2005.

[42] CondorTeam, DAGMan [Online]. Available: http://cs.wisc.edu/condor/dagman, Jul. 2014.

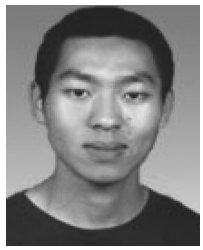
[43] M. Litzkow, M. Livny, and M. Mutka, "Condor—A hunter of idle workstations," in *Proc. 8th Int. Conf. Distrib. Comput. Syst.*, 1988, pp. 104–111.

[44] R. N. Calheiros, R. Ranjan, A. Beloglazov, C. A. F. De Rose, and R. Buyya, "Cloudsim: A toolkit for modeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms," *Softw. Pract. Exper.*, vol. 41, pp. 23–50, 2011.

[45] M. Mao and M. Humphrey, "A performance study on the VM startup time in the cloud," in *Proc. IEEE 5th Int. Conf. Cloud Comput.*, 2012, pp. 423–430.

Amelie Chi Zhou received the bachelor's and master's degrees from Beihang University. She is currently working toward the PhD degree at School of Computer Engineering of NTU, Singapore. Her research interests include cloud computing and database systems.

Bingsheng He received the bachelor's degree in computer science from SJTU, and the PhD degree in computer science from HKUST. He is an assistant professor in School of Computer Engineering of NTU, Singapore. His research interests include high-performance computing, cloud computing, and database systems.



Cheng Liu received the bachelor's and master's degrees from USTC. He is currently a research assistant in School of Computer Engineering of NTU, Singapore. His areas of expertise include structured peer-to-peer network and compiler.