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

Amelie Chi Zhou, Bingsheng He and Cheng Liu
Nanyang Technological University

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 cloud. While there have been previous studies on this concern, most of them assume static task execution time and static pricing scheme, and have the QoS notion of satisfying a deterministic deadline. However, cloud environment is dynamic, with 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 on IaaS clouds. We develop a probabilistic scheduling framework called Dyna to minimize the monetary cost while offering probabilistic deadline guarantees. The framework includes an A^*-based instance configuration method for performance dynamics, and a hybrid instance configuration refinement for utilizing spot instances. Experimental results with three real-world scientific workflow applications on Amazon EC2 demonstrate (1) the accuracy of our framework on satisfying the probabilistic deadline guarantees required by the users; (2) the effectiveness of our framework 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 various applications. One attractive feature of cloud computing is the pay-as-you-go charging scheme, where users only need to pay for the actual consumption of storage and computation hours. This feature unlocks the opportunities of large-scale computation without physically owning a cloud. 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 [1], [2]. In those applications, workflows are submitted and executed in the cloud and each workflow is usually associated with a deadline for QoS purposes [3], [4]. 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 and performance are usually the two most important optimization factors for hosting WaaS for the providers in the cloud. 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 [5], maximizing the performance for given budgets [6] and scheduling optimizations with both cost and performance constraints [7]. 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 [3], [8]. For example, most cloud providers offer instance hour billing model. Partial-hour consumption is always rounded up to one hour. Although some other billing models have been proposed (e.g., Google’s IaaS service charges by minutes of use), hourly billing is still the most commonly adopted model. 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., [3], [4], [8], [9], [10], [11], [12]). 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. The resources in the cloud, such as the computation, storage and network resources, are shared by many concurrent jobs/tasks. Previous studies [13], [14] have demonstrated significant variances on I/O and network performance. The assumption of static task execution time in the previous studies (e.g., [3], [4], [8], [9], [10], [11], [12]) does not hold in the cloud. Under the static execution time assumption, existing cost optimizations and algorithms try to satisfy the conventional QoS notion of a soft deadline with 100% guarantee. We denote the conventional deadline notion as the “deterministic deadline” in the remainder of this paper. However, due to performance dynamics, the actual execution time of a job after optimizations can be varying values with different probabilities. Requiring all the varying execution time to meet the deterministic deadline
is costly and meaningless. Thus, the deterministic deadline notion is not desirable to offer performance guarantees in dynamic cloud environments and the existing studies need to be revisited and adapted to performance dynamics.

Second, cloud, which has evolved into an economic market [13], has dynamic pricing. Amazon EC2 offers spot instances, whose prices are determined by market demand and supply. Previous studies [3, 5, 16, 17, 4, 18] consider fixed pricing schemes only and their results need revisits in the existence of spot instances. On the other hand, spot instances can be used to reduce monetary cost [19, 20, 21, 22, 23, 24], 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 latency due to failures. Most of the previous studies do not consider deadline constraints of individual workflows.

Those two kinds of cloud dynamics make challenging the problem of minimizing the cost of WaaS providers while satisfying QoS requirements of individual workflows. They add two new dimensions to the problem, and dramatically increase the solution space (see Section 3). Moreover, the deterministic deadline notion will lead the optimizations to worst-case performance/price prediction. Even worse, worst-case predictions can be unknown or unpredictable in some cases (e.g., there are some exceptionally high spot prices in the price history of Amazon EC2).

In order to address performance and price dynamics, we define the notion of probabilistic performance guarantees to represent QoS. Each workflow is associated with a probabilistic deadline guarantee of $p\%$. Deterministic deadline guarantee can be viewed as a special case of probabilistic deadline guarantee of 100%. WaaS provider guarantees that the workflow’s execution time is at the $p$-th percentile of the distribution of the workflow execution time in the dynamic cloud environment. This is just like many IaaS cloud providers offer a resource availability guarantee of 99.95% [25]. Under this notion, we propose a probabilistic framework called Dyna to minimize the cost of the WaaS provider while satisfying the probabilistic performance guarantees of individual workflows pre-defined by the user. The framework embraces a series of static and dynamic optimizations for monetary cost optimizations, which are specifically designed for cloud dynamics. We develop probabilistic models to capture the dynamics in I/O and network performance, and spot prices [26, 27]. We further propose a hybrid execution with both spot and on-demand instances, where 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 real-world workflow applications on real cloud environment. Our experimental results demonstrate the following two major results.

1) With the calibrations from Amazon EC2, Dyna can accurately capture the cloud dynamics and obtain optimization results always with the same or slightly better probability than the probabilistic performance guarantees required by the users.

2) The hybrid instance configuration approach significantly reduces the monetary cost by 15–73% over other state-of-the-art algorithms which only adopt on-demand instances.

To the best of our knowledge, our study is the first probabilistic scheduling framework for WaaS providers in the cloud.

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 framework design in Section 3 followed by the experimental results in Section 4. Finally, we conclude this paper in Section 5.

2 BACKGROUND AND RELATED WORK

In this section, we present the application scenario and describe the terminology, followed by the related work.

2.1 Application Scenario

Figure 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 [3, 11]. We call this model Workflow-as-a-Service (WaaS). In this hosting, different application owners submit a number of workflow classes with different parameters. A workflow class represents the template of the same workflow structure with different input data parameters. WaaS providers allow users to instantiate a workflow execution by specifying the input data to a workflow class, with specified deadlines for QoS purposes. The WaaS providers charge users according to the workflow classes (which reflect the complexity of the workflow) and QoS requirements. We can see many applications with different workflow structures and purposes in the cloud [11, 2]. Users (e.g., scientists and officials) can submit their simulation tasks for predictions, or perform sensitivity analysis. Users can also perform data analysis on scientific data with data mining or machine learning techniques.

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 framework to WaaS providers, aiming at minimizing the monetary cost while satisfying users’ QoS requirements.

2.2 Terminology

Instance. An instance is a virtual machine offered by the cloud provider. Instance in the same type have the same amount of resources such as CPUs and RAM and the same capabilities such as CPU speed, I/O speed and network bandwidth.

The instance acquisition has a non-ignorable acquisition time. For simplicity, we assume the acquisition time is a constant, lag.

An instance can be on-demand or spot. We adopt the instance definition of Amazon EC2. Amazon adopts the hourly
Task. A task can be of different classes, 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, machine benchmarking and statistical analysis. In this study, we use a simple performance estimation model on predicting the task execution time. Since workflows are often regular and predictable [3], [5], this simple approach is already sufficiently accurate in practice, as shown in our experiments on real cloud environments. Specifically, we estimate the execution time of a task on different types of instances using the following profile: \(<\#\text{instr}, d_{seqIO}, d_{radIO}, \text{netInput}, \text{netOutput}>,\) where \#\text{instr} represents the total number of instructions to be executed for the task, \(d_{seqIO}\) and \(d_{radIO}\) are the amount of I/O data for sequential and random accesses respectively to local disk, and \(\text{netInput}\) and \(\text{netOutput}\) are the amount of input and output data that need to be read in and sent out respectively. We estimate the task execution time as the summation of its CPU time, I/O time and networking time.

In the estimation, the CPU time is determined by the \#\text{instr} of the task as well as the CPU frequency of the instance that the task is executed on. Similarly, the I/O time and networking time is determined by the I/O and networking data size divided by the I/O and network bandwidth, respectively. We model the I/O and network performance in the cloud as probabilistic distributions and use them to estimate the dynamic execution time of tasks in the cloud.

Instance configuration. The hybrid instance configuration of a task is defined as an \(n\)-dimension vector: \(<(type_1, price_1, isSpot_1), (type_2, price_2, isSpot_2), ..., (type_n, price_n, isSpot_n)>,\) meaning the task is potentially to be executed by \(n\) instances belonging to \(type_i\) \((1 \leq i \leq n)\) in sequential. \(isSpot_i\) indicates whether the instance is spot or on-demand. 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. This is because the on-demand instance guarantees 100% of success execution.

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. To the best of our knowledge, this work is the first scheduling framework that captures both performance and price dynamics in the cloud.

Cost-aware optimizations. The pay-as-you-go nature of cloud computing attracts many research efforts in dynamic resource provisioning. Dynamic virtual machine provisioning has been determined by control theory [28], [29], machine learning [30] and models [31]. On the other hand, workflow scheduling with deadline and budget constraints (e.g., [5], [16], [17], [4], [18], [32], [33]) has been widely studied. Yu et al. [5] proposed deadline assignment for the tasks within a job and used genetic algorithms to find optimal scheduling plans. Klöppel et al. [4] studied the tradeoff between monetary cost and performance, and modeled the tradeoff as sky line operations in databases. Those studies only consider a single workflow with on-demand instances only. Malawski et al. [32] proposed dynamic scheduling strategies for workflow
ensembles. The previous studies \[3, 34, 35\] 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. Buyya et al. \[33\] 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 \[20, 21, 22, 23, 24, 36\] and leveraging spot instances \[19, 20, 21, 22, 23, 24, 36\].

For modeling spot prices, Yehuda et al. \[20\] conducted reverse engineering on the spot price and figured out a model consistent with existing price traces. Javadi et al. \[27\], \[37\] developed statistical models for different spot instance types. Those models can be adopted to our hybrid execution.

For leveraging spot instances, Chohan et al. \[21\] proposed a method to utilize the spot instances to speed up the MapReduce tasks, and suggested that fault tolerant mechanisms are essential to run MapReduce jobs on spot instances. Yi et al. \[19\] introduced some checkpointing mechanisms for reducing cost of spot instances. Ostermann et al. \[35\] utilize spot instances for large workflow executions when the Grid resources are not sufficient. Further studies \[23, 24\] 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.

**Cloud performance dynamics.** A few studies have evaluated the performance of cloud services from different aspects \[38, 13, 14\]. Our calibration results on Amazon EC2 is consistent with Schad et al.’s work \[13\]. Iosup et al. \[14\] did a cross-platform comparison on four commercial cloud providers (Amazon EC2, GoGrid, ElasticHosts and Mosso) for scientific computing workloads. There have been more in-depth performance studies on specific components (network performance \[49, 50\] and I/O interference \[41, 42\]).

There have been some proposals to reduce the performance interference and unpredictability in the cloud, such as network performance \[43\] and I/O performance \[44, 45, 46\]. However, by the design of cloud computing, cloud is shared by many concurrent executions. Therefore, the performance dynamics caused by the resource interference is unavoidable. This paper offers a probabilistic notion to capture the performance and cost dynamics, and further develop a probabilistic scheduling framework to minimize the monetary cost with the consideration of those dynamics.

### 3 Framework Design and Implementation

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

![Fig. 2. Static and dynamic optimizations in the Dyna framework](image)

#### 3.1 Framework Overview

When we design the framework for minimizing the monetary cost of WaaS provider, we consider the following design principles.

- **Effectiveness.** The proposed framework should optimize the monetary cost with the special consideration on the performance and price dynamics in the cloud environment. Moreover, while satisfying probabilistic deadline guarantees for individual workflows, the framework should be able to find the solution that is comparable or close to the optimal solution.

- **Generality.** The reason that we develop a general framework has two folds. First, there have been some classic job/task scheduling optimizations for monetary cost and performance \[5, 6, 7\]. A framework allows integrating existing optimizations in a holistic manner. Second, different cloud providers may have different behaviors and offerings in performance and price dynamics. A general framework allows more flexibility in implementing specific algorithms for different cloud providers, without affecting other key optimization components.

- **Low runtime overhead.** The cost optimization is an online process and should be lightweight. We should find a good balance between the quality of monetary cost optimization and the runtime overhead of the optimization process itself. Due to the huge space, a thorough exploration of the optimization space is impractical.

With the three principles, we propose Dyna, a probabilistic scheduling framework for workflows. The main components of Dyna are illustrated in Figure 2. The framework consists of static and dynamic optimizations.

When a workflow is submitted to the WaaS provider with the pre-defined probabilistic deadline guarantee, we first determine the most cost-efficient configuration plan for each task as static optimizations. There are two major static optimizations for generating the configuration plan for a workflow: firstly an \(A^*\)-based instance configuration approach for selecting the on-demand instance type, and secondly the hybrid instance configuration refinement for considering spot instances. Note that for the same workflow class (with the same structure and task profiles), we only need to do the static optimization once, and we can retrieve the configuration plan directly from the configuration plan cache. The function of configuration plan cache is to store the configuration plans for the workflow classes that have been executed. At runtime, the tasks of the workflow are scheduled according to their instance configuration and several dynamic optimization techniques including
consolidation and instance reuse are applied to further reduce cost.

In the remainder of this section, we outline the design of static and dynamic optimizations, and discuss on the implementation details.

3.2 Static Optimizations

The overall functionality of static optimizations is to determine the suitable instance configuration for each task in the workflow so that the monetary cost is minimized while the probabilistic performance guarantees are satisfied. Ideally, one can consider on-demand and spot instances together. However, this will induce too large solution space. Therefore, we consider a divide-and-conquer approach on the static optimizations. The static optimizations include two kinds of optimizations: $A^*$-based instance configuration and hybrid instance configuration refinement. The rationale is to first determine the on-demand instance type for each task, and then to perform refinement by introducing hybrid execution configurations to workflow execution so that the monetary cost is further reduced. Particularly, we formulate the process of determining the on-demand instance type into an $A^*$-based approach. Based on the output from the $A^*$-based configuration, we determine the hybrid instance configuration as a refinement for each task.

3.2.1 $A^*$-Based Instance Configuration

In this optimization, we determine an on-demand instance type for each task in the workflow. We formulate the process into an $A^*$-based search problem. The reason that we choose $A^*$ search is to take advantage of its pruning capability to reduce the large search space while targeting at a high quality solution.

The $A^*$-based instance configuration is extended from the classical $A^*$ search process. In the formulated $A^*$ search, we define the state to be a configuration plan to the workflow. For ease of presentation, we represent a configuration plan to be a multi-dimensional vector of the instance configuration for each task in the workflow. The $i$-th dimension of the vector corresponds to the assigned instance configuration of the task of ID $i$. We have two issues to clarify. First, we ensure that each task has a unique ID. The order of assigning ID to the task does not affect the correctness of our algorithm. In this study, we simply use a topological order to assign the task ID. Second, at this static optimization, the instance configuration consists of a single on-demand instance, which will be extended to hybrid instance configuration in the next subsection.

The goal of our $A^*$-based algorithm is to search for an optimal state on the search tree, which has the lowest monetary cost and can satisfy the probabilistic deadline guarantee. Algorithm 1 shows the optimization process of the $A^*$-based instance configuration algorithm. The algorithm maintains all the states in the search tree with two list structures: ClosedList and OpenList. The OpenList is to store the states that are being considered to find the goal state and ClosedList is to store the states that do not need to consider again during the $A^*$ search. Functions estimate_h_score and estimate_g_score return the $h$ and $g$ scores of states, respectively, and estimate_cost and

![Workflow structure and Configuration Plan](image)

**Algorithm 1 $A^*$-Based Instance Configuration Search**

**Require**: Max_iter: Maximum number of iterations; deadline, $p_i$: Required probabilistic deadline guarantee

1. ClosedList := empty;
2. OpenList := $S$;
3. upperBound := 0;
4. $g[S] := 0$;
5. $f[S] := g[S] + \text{estimate}_h\text{-score}(S)$;
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, $p_i$);
   9. if percentile <= deadline then
      10. current_cost := estimate_cost(current);
      11. if current_cost < upperBound then
         12. upperBound = current_cost;
         13. $D = current$;
      14. Remove current from OpenList;
      15. Add current to ClosedList;
   16. for each neighbor in neighboring states of current do
      17. $g[neighbor] := \text{cal}_g\text{-score}(neighbor, S)$;
      18. $f[neighbor] := h[neighbor] + \text{estimate}_h\text{-score}(neighbor)$;
      19. if $f[neighbor] >= upperBound$ or neighbor is in ClosedList then
         20. continue;
      21. if neighbor is not in OpenList then
         22. Add neighbor to OpenList;
   23. Return $D$;

Originally, $A^*$ search algorithm is a heuristic searching method that searches for the shortest path from a given initial state to the specified goal state. During the search process, $A^*$ algorithm attempts to calculate the smallest cost so far and to prune the unnecessary states at each state. Particularly, $A^*$ 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 $f(s) = g(s) + h(s)$. The $A^*$ search algorithm is equivalent to the search and pruning process on a search tree. Figure 3 shows an example of the configuration plan search tree in our problem setting. Each node represents a state in the $A^*$ search algorithm (a state is a configuration plan for the workflow in our paper). A child state only differs with its parent state, by replacing a dimension with a more expensive instance type.

![Configuration Plan Search Tree](image)
The cost of the task to be completed within the specified time constraints could severely limit the optimization space. For example, if the execution time distribution of the structure in Figure 4(b) is calculated as $\text{MAX}(PDF_0, PDF_1, ..., PDF_{n-2}) + PDF_{n-1}$, 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 $\text{MAX}$ operation finds the maximum of the two distributions. After obtaining the execution time distribution of the workflow, we check its percentile at the required probabilistic deadline guarantee. Only if the returned percentile is smaller than deadline, the evaluated configuration plan is feasible.

We set the initial state of the $A^*$ search to configuration (0,0,0,...,0), where each task is configured with the cheapest instance type (instance type 0). The goal state is the optimal configuration plan. We define $g(s)$ to be the monetary cost difference between $s$ and the initial state. We estimate $h(s)$ to be the monetary cost of configuration plan $s$. We maintain the lowest cost found during the searching process as the upper bound to prune the useless states on the search tree. Note, we only consider the feasible states that satisfy the probabilistic performance guarantee. The leaves of the tree include all possible configuration plans of a workflow. Starting from the initial configuration, the search tree is constructed by expanding the state with its child states. If the monetary cost of a state $s$ is higher than the upper bound, its successors are unlikely to be the goal state since they have more expensive configurations than $s$. For example, suppose configuration (1,1,0) on the search tree in Figure 3 has a higher monetary cost than the upper bound. The grey nodes on the search tree can be pruned during the $A^*$ search.

3.2.2 Hybrid Instance Configuration Refinement

We consider the adoption of spot instances as a refinement to the configuration plan obtained from the $A^*$-based instance configuration algorithm to further reduce monetary cost. The major issue with 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 hybrid instance configuration with the adoption of both on-demand and spot instances to tackle this issue. The basic idea is, if the deadline allows, we can run a task on a spot instance in advance (before we run the task on an on-demand instance). If the task can finish on the spot instance, the monetary cost tends to be lower. 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. Figure 5 compares on-demand, spot and hybrid configurations for a single task. If a task fails on a spot
instance, its failure does not trigger the re-execution of its
precedent tasks. The results of precedent tasks are already
checkpointed, and materialized to the persistent storage in
the cloud (such as Amazon S3). Dyna performs checkpointing
only when the task ends, which is simple and has much less
overhead than the general checkpointing algorithms [19].

A hybrid instance configuration 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 $A^*$-based
instance configuration step. Starting from the initial hybrid config-
uration to be only one on-demand type, we repeatedly add
spot instances in front of the hybrid configuration to find
the optimal hybrid configuration. Refining a hybrid instance
configuration $C_1$ of task $T$ to a hybrid instance configuration
$C_2$, we need to determine whether $C_2$ is better than $C_1$ in
terms of execution time distributions. Particularly, we define
$C_2 \geq C_1$ if for all $t$, we have $\int_0^t P_{T,C_2}(time = x) dx \geq
\int_0^t P_{T,C_1}(time = x) dx$, where $P_{T,C_1}$ and $P_{T,C_2}$ are the PDFs
of task $T$ under hybrid instance configuration $C_1$ and $C_2$,
respectively. Figure 6 illustrates this definition. The integrals
are represented as CDFs (Cumulative distribution functions).

Ideally, we can add $n$ spot instances ($n$ is a predefined pa-
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 small improvement
on the optimization results.

For each added spot instance in the hybrid instance config-
uration, we need to decide its type and associated bidding price.
Due to price dynamics of spot instances, making the decision
is non-trivial. We search for the spot instance types and the
associated bidding prices as described in Algorithm 2. To
reduce the search space, we design a heuristic. The added spot
instance type should be at least as expensive as the on-demand
instance type in the hybrid configuration in order to ensure
the probabilistic deadline guarantee. For each spot instance
type, we search the bidding price in the range of $[P_{\min}, P_{\max}]$
using the search algorithm described in Algorithm 3. In our
implementation, $P_{\min} = 0.001$ and $P_{\max}$ equals to the price
of the on-demand instance of the same type. The spot instance
with the bidding price higher than $P_{\max}$ does not contribute
to monetary cost reduction. If an bidding price is found for
a certain instance type, this instance type is assigned to the
hybrid instance configuration with the found bidding price. If
no bidding price is suitable for any spot instance type, the
added spot instance type is assigned with $-1$ indicating the
spot instance is not added.

**Algorithm 2** Hybrid instance configuration refinement for a
task $T$.

---

**Algorithm 3** Binary_search function for task $T$.

---

In Algorithm 3, the bidding price is searched using a binary
search algorithm. We have the following two considerations:
first, adding the spot instance should not violate the proba-
listic deadline guarantee; second, the estimated cost of the
refined hybrid configuration should be less than that before the
refinement. If both considerations are satisfied for a certain
bidding price, we return this price to the hybrid instance
configuration.
the probabilistic deadline guarantee, one way is to calculate the probabilistic distribution of the entire workflow execution time and then to decide whether the required percentile in the probabilistic distribution is smaller than deadline. However, this calculation requires large overhead. We implement this process in the Oracle algorithm presented in Section 4 while in Dyna, we propose a light-weight localized heuristic to reduce the overhead. As the on-demand configuration found in the A* based instance configuration step has already ensured the probabilistic deadline requirement, we only need to make sure that the hybrid configuration of each task \( C_{hybrid} \) satisfies \( C_{hybrid} \geq C_{ondemand} \) where \( C_{ondemand} \) is the on-demand only configuration. If this requirement is not satisfied, it means the current bidding price is too low and we continue the search of the bidding price in the higher half of the current search range.

Since a spot instance may fail at any time, we define a probabilistic function \( ffp(t, p) \) to calculate the 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 [27] or reverse engineering [26]. 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.

In order to decide whether a refined hybrid instance configuration is better, we first discuss how to estimate the overall execution time distribution of a task given a hybrid instance configuration. Assume a hybrid instance configuration of task \( T \) is \( C_{hybrid} = <(type_1, P_0, isSpot), (type_2, P_o, isOnDemand)> \). Assume the probabilistic distributions 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 \( C_{hybrid} \) can be divided into two cases. If the task successfully finishes on the spot instance, the overall execution time equals to the execution time of task \( T \) on the spot instance \( t_s \) with the following probability.

\[
P_{T,C_{hybrid}}(time = t_s) = P_{T,type_1}(time = t_s) \times (1 - \int_0^{t_s} ffp(x, P_0) \, dx)
\]

Otherwise, the overall execution time equals to the time 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.

\[
P_{T,C_{hybrid}}(time = t_f + t_o) = P_{T,type_1}(time = t_f) \times P_{T,type_2}(time = t_o) \times ffp(t_f, P_o)(t_f \leq t_s)
\]

After obtaining the execution time distribution of a task under the hybrid instance configuration \( C_{hybrid} \), we compare it with the on-demand configuration \( C_{ondemand} \) using the definition shown in Figure 5. If \( C_{hybrid} \geq C_{ondemand} \) is satisfied, we accept the refined hybrid instance configuration for the task. We use this heuristic to localize the computation of the execution time distribution of the entire workflow to each task and greatly reduce the optimization overhead.

**Monetary cost consideration.** If the expected monetary cost of the refined configuration is higher than that before the refinement, it means the bidding price is set too high. Thus we continue the search of the bidding price in the lower half of the search range. We estimate the cost of a hybrid configuration of a task as described in Algorithm 4. Similar to the analysis for performance estimation, we obtain the execution time distribution of the hybrid configuration and calculate the expected monetary cost under the searched bidding price. We compare the expected monetary cost of the hybrid configurations before and after refinement, and add the spot instance only if the refinement can reduce the expected monetary cost.

**Algorithm 4** Estimate monetary cost, \( C \), for a task with hybrid instance configuration of 2 instances

**Require:** Hybrid instance configuration of the task \( <type_1, P_0, isSpot> \) and \( <type_2, P_o, isOnDemand> \); Execution time samples of the task on \( type_1 \) instance: \( stime_i \) (\( 1 \leq i \leq N \)); Execution time samples of the task on \( type_2 \) instance: \( atime_i \) (\( 1 \leq i \leq N \));

1. \( C = 0 \);
2. for \( i = 1 \) to \( N \); \( i = i + 1 \) do
3. \( p = 0 \);
4. for \( t = 0 \) to \( stime_i \); \( t = t + \) step do
5. \( p = p + ffp(t, P_0) \);
6. \( C = C + P_0 \times stime_i + P_o \times atime_i \);
7. \( C = \frac{C}{\text{NUM}} \);
8. Return \( C \);

### 3.3 Dynamic Optimizations

When a task’s all preceding tasks in the workflow have finished execution, the task becomes ready and can be scheduled to execute on an instance according to the instance configuration determined from the static optimization. In the dynamic optimization stage, we adopt the consolidation and scaling technique and implement online instance reuse to further optimize execution cost. These techniques have been widely used in the previous studies [3], [12]. Still, we need to extend them with the consideration of hybrid instance configuration. In the following, we briefly introduce those techniques, with the special attention to the difference brought by supporting hybrid instance configurations.

**Consolidation and Scaling.** Consolidating different instances could reduce the new instance acquisition time as well

![Fig. 7. An example of hybrid execution](image-url)
as the instance-hour billing overhead. For each instance in the pool, we record the instance’s remaining partial hours.

Due to the difference between spot and on-demand instances, virtual machine consolidation among them needs special care. If the remaining time of the spot (resp. on-demand) instance is long enough for a ready task’s execution, and the task is going to acquire a spot (resp. on-demand) instance of the same type, the task will be assigned to the instance. The consolidation between spot and on-demand instances is possible only for the case whereby spot instance request is consolidated to an on-demand instance, due to the success guarantee.

Online Instance Reuse. At runtime, we maintain a pool of running instances, organized in lists according to different instance types. The spot and on-demand instances with the same instance type are organized in separated lists. During runtime, an instance request is processed at the instance starting time by first looking into the instance pool for an idle instance of the requested type. If such an instance is found, it will be selected for workflow execution. Otherwise, a new instance of the requested type is acquired from the IaaS cloud. Thus, the instances started during workflows execution can be properly reused and their utilizations are improved. Additionally, if we can reuse the instances, the instance acquisition time is eliminated.

Figure 7 shows an example of the pool of current spot and on-demand instances as well as two tasks named T1 and T2 to be executed. Initially, task T1 is assigned to a spot instance with type $\text{type}_1$. But the execution of task T1 fails due to out-of-bid event and restarts on a $\text{type}_2$ spot instance. If there is no $\text{type}_2$ spot instance available in the instance pool, the task will wait for the setup of a new $\text{type}_2$ spot instance which will be bid with the price indicated in the task’s hybrid instance configuration. At the meantime, as T2 has failed on both $\text{type}_1$ and $\text{type}_2$ spot instances, it will execute on a $\text{type}_2$ on-demand instance. Similarly, if there is no $\text{type}_2$ on-demand instance in the pool, a new $\text{type}_2$ on-demand instance is requested first and added into the instance pool.

4 Evaluation

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

4.1 Experimental Setup

We have two sets of experiments on real cloud environments: firstly calibrating the cloud dynamics from Amazon EC2 as the input of our optimization framework and secondly running real-world scientific workflows on Amazon EC2 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 [13]. 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 1,0000 times. 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 $\text{hdparm}$. 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 8MB. We also measured the network bandwidth between two instances using Iperf [47]. We find that the network bandwidth between instances of different types is generally lower than that between instances of the same type and S3.

Workload. There have been some works on characterizing the performance behaviours of scientific workflows [48], [49]. In this paper, we consider three common workflow structures, namely Ligo, Montage and Epigenomics. Ligo (Laser Interferometer Gravitational Wave Observatory) is an application used to detect gravitational-wave. Montage is an astronomical application widely used as a Grid and parallel computing benchmark. Epigenomics is a data processing pipeline of various genome sequencing operations. As shown in Figure 8, the three workflows have different structures and parallelism. They also have different requirements on computation resources. For example, Montage is I/O-intensive workload, Ligo is memory-intensive workload and Epigenomics is CPU-intensive. Thus, with the three workflows, we are able to examine the effectiveness of our proposed algorithm on different workloads. These workflows have been characterized in details by Juve et al [48] and readers can refer to their work for more detailed information.

Comparisons. In order to evaluate the effectiveness of the proposed techniques in Dyna, we have implemented the following algorithms on Amazon EC2.

- Static. This approach is the same as the previous study in [3] 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 QoS setting used in Dyna. For example, if user requires 90% of probabilistic deadline, the deterministic deadline used in the experiment is set to the 90-th percentile of its
execution time distribution.

- **Dyna-NS.** This approach is the same as Dyna except that Dyna-NS does not use any spot instances. The comparison between Dyna and Dyna-NS is to assess the impact of spot instances.

- **SpotOnly.** This approach adopts only spot instances during execution. It first utilizes the A* based 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 required deadline meeting rate.

- **Oracle.** We implement the Oracle method to assess the tradeoff between the optimization overhead and the effectiveness of the optimizations in Dyna. Oracle is different from Dyna in the following key designs. In the binary search method, which is used to search bidding prices for spot instances in the hybrid configurations of tasks, Oracle does not adopt the localized heuristic for percentile calculation as Dyna. On the contrary, for each searched bidding price of a task, it calculates the overall execution time distribution of the workflow to decide if the searched bidding price for the task satisfies the probabilistic deadline requirement. This is an offline approach, since the time overhead of getting the solution in Oracle is prohibitively high.

We have adopted DAGMan [50] to manage task dependencies and added Condor [51] to the Amazon Machine Image (AMI) to manage task execution and instance acquisition.

We acquire the instances from the US East region. The hourly costs of the on-demand instance for the four instance types used in workflow execution are shown in Table 1. Those four instances have also been used in the previous studies [19]. 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 by Mao et al. [52].

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: $D_{\text{min}}$ and $D_{\text{max}}$, 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 $D_{\text{max}} + D_{\text{min}} / 2$.

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

As for metrics, we study the average monetary cost and elapsed time for a workflow. All the metrics 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

Fig. 9. The histogram and probabilistic distribution of random I/O performance on Medium instances

Fig. 10. The histogram and probability distribution of downloading bandwidth between Medium instances and S3 storage

the required deadline hit rate as 96%. 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 subsection, we present the performance dynamics observed on Amazon EC2. The price dynamics have been presented in Table 1 of Section 2.

Figure 2 and Figure 10 show the measurements of random I/O performance and downloading network performance from Amazon S3 of 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 Gamma distribution; (2) the random I/O performance distributions on the four instance types follow normal distribution. The parameters of these distributions are presented in Tables 2 and 3.

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 $\theta$ parameter of Gamma distributions or the $\sigma$ 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%, respectively.
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      | $k = 3.6/0.6, \theta = 0.28$ | $\mu = 172.9, \sigma = 34.8$ |
| m1.xlarge     | $k = 408.4, \sigma = 0.26$ | $\mu = 1354.0, \sigma = 146.4$ |

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 $\mu$ parameter of normal distributions in Tables 2 and 3.

4.3 Overall Comparison

In this sub-section, we present the overall comparison results of Dyna and the other compared algorithms on Amazon EC2 under the default settings. Sensitivity studies are presented in Section 4.4. Table 4 shows the obtained deadline hit rates. Note that we have used the calibrations from Section 4.2 as input to the Dyna optimizations. The results demonstrate the capability of our designed techniques on accurately satisfying the required probabilistic deadline guarantees. Although Static can guarantee the probabilistic deadline requirement with a higher deadline hit rate, it causes a much higher extra monetary cost of the WaaS provider, as we will demonstrate later in this subsection.

Figure 11 shows the average monetary cost per job results of Static, DynaNS, SpotOnly, Dyna and Oracle methods on Montage, Ligo and Epigenomics workflows when the probabilistic deadline guarantee is 90%.

First, DynaNS obtains smaller monetary cost than Static, because the proposed $A^*$ configuration search technique is capable of finding cheaper instance configurations and is suitable for different structures of workflows. This also shows that

TABLE 5
Optimization overhead of the compared algorithms on Montage, Ligo and Epigenomics workflows (seconds).

|          | Static | DynaNS | SpotOnly | Dyna | Oracle |
|----------|--------|--------|----------|------|--------|
| Montage  | 1      | 153    | 153      | 163  | 2997   |
| Ligo     | 1      | 236s   | 236      | 244  | 10452  |
| Epigenomics | 1   | 166    | 166      | 175  | 2722   |

Fig. 11. The average monetary cost optimization results of compared algorithms on Montage, Ligo and Epigenomics workflows in Amazon EC2.

Fig. 12. The average execution time optimization results of compared algorithms on Montage, Ligo and Epigenomics workflows in Amazon EC2.

Fig. 13. The average monetary cost results of compared algorithms on Montage, Ligo and Epigenomics workflows when the probabilistic deadline guarantee is 90%.

Fig. 14. Histogram of the spot price history in August 2013, US East Region of Amazon EC2.
performing deadline assignment before instance configuration in the Static algorithm reduces the optimization effectiveness.

Second, Dyna obtains smaller monetary cost than DynaNS, meaning that the hybrid configuration with spot instances is effective on reducing monetary cost. As the probabilistic deadline guarantee set lower, the monetary cost saved by Dyna over DynaNS gets higher. Figure 13 shows the monetary cost results of the compared algorithms when the probabilistic deadline guarantee is set to 90%. In this setting, Dyna saves more monetary cost than DynaNS by 28–37%.

Third, SpotOnly obtains the highest monetary cost among all the compared algorithms. This is due to the dynamic characteristic of spot price. Figure 14 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. The relative monetary cost of SpotOnly over the other compared algorithms is especially higher on Ligo because the average execution time of Ligo is longer than the other workflows. 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 4.3.

Figure 12 shows the average execution time of a workflow of Static, DynaNS, SpotOnly, Dyna and Oracle methods on the Montage, Ligo and Epigenomics workloads in Amazon EC2. The standard errors of the execution time results of the compared algorithms are small, all in 0.004–0.01 on the tested workflows. Static has the smallest average execution time. This is because Static configures each task in workflows with better and more expensive instance types. The average execution times of SpotOnly, Dyna and Oracle are similar. This is because the three algorithms all use the proposed static optimization to configure each task in the workflow with a certain instance type. Also, the careful selection of bidding price for each task in the workflow in Dyna and high bidding prices in SpotOnly diminishes the out-of-bid events during execution. On the other hand, we can also see that the bidding price searched during static optimization is able to diminish the out-of-bid events.

Finally, we analyze the optimization overhead of the compared algorithms. The optimization overhead results are shown in Table 5. Note that, for workloads with the same structure and profile, our framework 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 large optimization overhead, in the rest of the experiments, we do not evaluate Oracle but only compare Dyna with Static, DynaNS and SpotOnly.

![Average monetary cost per job](image1)

**Fig. 15.** The average monetary cost and average execution time results of sensitivity studies on deadline.

![Breakdown when deadline is 1.5 \times D_{min} and 0.75 \times D_{max}](image2)

**Fig. 16.** Breakdown of the instance types adopted by compared algorithms when the deadlines are 1.5 \times D_{min} and 0.75 \times D_{max}.

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}, 0.5 \times (D_{min} + D_{max}) to 0.75 \times D_{max}, which are 138 minutes, 284 minutes and 360 minutes on average for the Montage workflow, respectively. All results are normalized to those of Static when deadline is 0.5 \times (D_{min} + D_{max}). Figure 15 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. 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. We further break down the number of different types of on-demand and spot instances when the deadlines are 1.5 \times D_{min} and 0.75 \times D_{max} as shown in Figure 16. The breakdown results of SpotOnly and Dyna are the same as DynaNS because they all use the \(A^\star\)-based instance configuration method. When the deadline is loose (0.75 \times D_{max}), more cheap instances are utilized. Also, when under the same deadline, e.g., 1.5 \times D_{min}, DynaNS, SpotOnly and Dyna utilize more cheap instances than Static, which again shows our \(A^\star\) approach is better than the existing heuristics [3].

**Probabilistic deadline Guarantee.** We evaluate the effectiveness of Dyna on satisfying probabilistic deadline re-
requirements when the requirement varies from 90% to 99.9%. Figure 17 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. Table 6 shows the obtained deadline hit rate by the compared algorithms with varying required deadline hit rate. DynaNS, SpotOnly and Dyna can accurately satisfy the probabilistic deadline guarantees while Static cannot.

Arrival rate. We evaluate the effectiveness of Dyna when the arrival rate $\lambda$ 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. Figure 18 shows the optimized average monetary cost per job. Dyna obtains the smallest average monetary cost under all job arrival rates. As the job arrival rate increases, the average cost per job is decreasing. This is because there are more jobs sharing the resources rented by the WaaS provider from the cloud and also sharing the hourly cost of instances charged by the IaaS cloud.

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 7, 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). We conjecture that more users are using the spot instances from Amazon EC2, which causes the larger fluctuations in August 2013 than December 2011. Figure 19 shows the obtained monetary cost result. SpotOnly and Dyna obtain similar monetary cost results, which are much lower than Static and DynaNS. This demonstrates Dyna is able to outperform SpotOnly on monetary cost optimization for different spot price distributions.

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 in dynamic cloud environments. We develop a probabilistic framework named Dyna for scheduling scientific workflows with the goal of minimizing the monetary cost while satisfying their probabilistic deadline guarantees. The framework embraces a series of static and dynamic optimizations. We further develop hybrid instance configuration of spot and on-demand instances for price dynamics. We deploy Dyna on Amazon EC2 and evaluate its effectiveness with real

Fig. 17. The average monetary cost and average execution time results of sensitivity studies on the probabilistic deadline guarantees.

Fig. 18. The average monetary cost and average execution time results of sensitivity studies on the arrival rate of workflows.

Fig. 19. The simulation result of the average monetary cost obtained by the compared algorithms, using the spot price history of the Asia Pacific Region of Amazon EC2 in December, 2011.

TABLE 6
Obtained deadline hit rates by the compared algorithms with varying required deadline hit rate.

| Required deadline hit rate | Static | DynaNS | SpotOnly | Dyna |
|---------------------------|--------|--------|----------|------|
| 90%                       | 98.7%  | 90.7%  | 90.6%    | 90.7%|
| 92%                       | 98.7%  | 92.4%  | 92.3%    | 92.5%|
| 94%                       | 99.9%  | 94.3%  | 94.4%    | 94.3%|
| 96%                       | 99.9%  | 96.6%  | 96.5%    | 96.6%|
| 98%                       | 99.9%  | 98.2%  | 98.2%    | 98.0%|
| 99.9%                     | 100%   | 100%   | 100%     | 100%|

TABLE 7
Statistics on spot prices ($/hour, December 2011, Asia Pacific Region) and on-demand prices of Amazon EC2.

| Instance type | Average | stddev | 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    |
scientific workflows. Our experimental results demonstrate that Dyna achieves much lower monetary cost than the state-of-the-art approaches (by 73%) while accurately meeting users’ probabilistic requirements.

REFERENCES

[1] Amazon Case Studies, [http://aws.amazon.com/solutions/case-studies/]
[2] Windows Azure Case Studies, [http://www.microsoft.com/azure/casestudies.htm]
[3] M. Mao and M. Humphrey, “Auto-scaling to minimize cost and meet application deadlines in cloud workflows,” in SC ’11, 2011, pp. 49:1–49:12.
[4] H. Kløp, E. Sitari, M. M. Tsangaris, and Y. Ioannidis, “Schedule optimization for data processing flows on the cloud,” in SIGMOD ’11, 2011, pp. 289–300.
[5] J. Yu, R. Buyya, and C. K. Tham, “Cost-based scheduling of scientific workflow application on utility grids,” in e-Science ’05, 2005, pp. 140–147.
[6] R. Sakellariou, H. Zhao, E. Tsiaikouri, and M. D. Dikaiaokos, “Scheduling workflows with budget constraints,” in CoreGRID ’05, 2005.
[7] R. Duan, R. Prodan, and T. Fahringer, “Performance and cost optimization for multiple large-scale grid workflow applications,” in SC ’07, 2007, pp. 12:1–12:12.
[8] S. Abrishami, M. Naghibzadeh, and D. Epena, “Deadline-constrained workflow scheduling algorithms for iaas clouds,” in SC ’12, 2012.
[9] E.-K. Byun, Y.-S. Kee, J.-S. Kim, and S. Maeng, “Cost optimized provisioning of elastic resources for application workflows,” in Future Gener. Comput. Syst., 2011, pp. 1011–1026.
[10] S. T. Maguluri, S. R. Srikan, and L. Ying, “Stochastic models of load balancing and scheduling in cloud computing clusters,” in INFOCOM ’12, 2012.
[11] F. Zhang, J. Cao, K. Hwang, and C. Wu, “Ordinal optimized scheduling of scientific workflows in elastic compute clouds,” in CLOUDCOM ’11, 2011.
[12] M. Malawski, G. Juve, E. Deelman, and J. Nabrzyski, “Cost- and deadline-constrained provisioning for scientific workflow ensembles in iaas clouds,” in SC ’12, 2012, pp. 22:1–22:11.
[13] J. Schad, J. Dittrich, and J.-A. Quiñé-Ruiz, “Runtime measurements in the cloud: observing, analyzing, and reducing variability,” Proc. VLDB Endow., vol. 3, no. 1-2, pp. 460–471, Sep. 2010.
[14] A. Issup, S. Ostermann, N. Yigitbasi, R. Prodan, T. Fahringer, and D. Epena, “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.
[15] H. Wang, Q. Jing, R. Chen, B. He, Z. Qian, and L. Zhou, “Distributed systems meet economics: pricing in the cloud,” in HotCloud ’10, 2010, pp. 6–6.
[16] S. K. Garg, R. Buyya, and H. J. Siegel, “Time and cost trade-off management for scheduling parallel applications on utility grids,” Future Gener. Comput. Syst., vol. 26, Oct. 2010.
[17] R. Sakellariou, H. Zhao, E. Tsiaikouri, and M. D. Dikaiaokos, “Scheduling workflows with budget constraints,” in CloudCom ’10, 2010, pp. 351–359.
[18] S. Yi, A. Andrzejak, and D. Kondo, “Monetary cost-aware checkpointing and migration on amazon cloud spot instances,” in Proc. of the ACM SIGCOMM 2011 conference, ser. SIGCOMM ’11, 2011, pp. 242–253.
[19] M. Malawski, G. Juve, E. Deelman, and J. Nabrzyski, “Cost- and deadline-constrained provisioning for scientific workflow ensembles in iaas clouds,” in SC ’12, 2012.
[20] E. Sitaridi, M. M. Tsangaris, and Y. Ioannidis, “Schedule provisioning system for the cloud,” in CloudCom ’10, 2010, pp. 49:1–49:12.
[21] S. Yi, A. Andrzejak, and D. Kondo, “Deconstructing amazon ec2 spot instance pricing,” in CloudCom ’11, 2011, pp. 304–311.
[22] B. Javadi, R. Thulasiram, and R. Buyya, “Statistical modeling of spot instance prices in public cloud environments,” in UCC ’11, 2011.
[23] Q. Zhu and G. Agrawal, “Resource provisioning with budget constraints for adaptive applications in cloud environments,” in HPDC ’10, 2010, pp. 304–307.
[24] B. Bouterse and H. Perros, “Scheduling cloud capacity for time-varying customer demand,” in CloudNet ’12, 2012.
[25] S. Zhang, L. Cherkasova, and E. Smirni, “A regression-based analytic model for dynamic resource provisioning of multi-tier applications,” in ICAC ’07, 2007.
[26] C. Castillo, M. Spreitzer, M. Steinder, A. Tantawi, and A. Schuster, “Who is your neighbor: Net i/o performance interference in virtualized cloud environments,” in IPDPSW ’11, May 2011, pp. 1042–1051.
[27] X. Lin, Y. Mao, F. Li, and R. Ricci, “Towards fair sharing of block storage in a multi-tenant cloud,” in HotCloud ’12, Jun. 2012.
[28] R. C. Chiang and H. H. Huang, “Tracon: interference-aware scheduling for data-intensive applications in virtualized environments,” in SC ’11, 2011, pp. 47:1–47:12.
[29] Iperf,” [http://iperf.sourceforge.net]
[30] S. Buyya, R. Buyya, H. J. Siegel, “Time and cost trade-off management for scheduling parallel applications on utility grids,” Future Gener. Comput. Syst., vol. 26, October 2010.
[31] M. Malawski, G. Juve, E. Deelman, and J. Nabrzyski, “Cost- and deadline-constrained provisioning for scientific workflow ensembles in iaas clouds,” in SC ’12, 2012, pp. 22:1–22:11.
[32] S. Yi, A. Andrzejak, and D. Kondo, “Monetary cost-aware checkpointing and migration on amazon cloud spot instances,” in Proc. of the ACM SIGCOMM 2011 conference, ser. SIGCOMM ’11, 2011, pp. 242–253.
[33] M. Hovestadt, O. Kao, A. Kliem, and D. Warneke, “Evaluating adaptive compression to mitigate the effects of shared I/O in clouds,” in IPDPSW ’11, May 2011, pp. 1042–1051.
[34] X. Lin, Y. Mao, F. Li, and R. Ricci, “Towards fair sharing of block storage in a multi-tenant cloud,” in HotCloud ’12, Jun. 2012.
[35] H. Librat, P. Costa, T. Karagianis, and A. Rowstron, “Towards predictable datacenter networks,” in Proc. of the ACM SIGCOMM 2011 conference, ser. SIGCOMM ’11, 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 IPDPSW ’11, May 2011, pp. 1042–1051.
[37] X. Lin, Y. Mao, F. Li, and R. Ricci, “Towards fair sharing of block storage in a multi-tenant cloud,” in HotCloud ’12, Jun. 2012.
[38] R. C. Chiang and H. H. Huang, “Tracon: interference-aware scheduling for data-intensive applications in virtualized environments,” in SC ’11, 2011, pp. 47:1–47:12.
[39] [http://iperf.sourceforge.net]
[40] S. Buyya, R. Buyya, H. J. Siegel, “Time and cost trade-off management for scheduling parallel applications on utility grids,” Future Gener. Comput. Syst., vol. 26, October 2010.
[41] M. Malawski, G. Juve, E. Deelman, and J. Nabrzyski, “Cost- and deadline-constrained provisioning for scientific workflow ensembles in iaas clouds,” in SC ’12, 2012, pp. 22:1–22:11.
[42] S. Yi, A. Andrzejak, and D. Kondo, “Monetary cost-aware checkpointing and migration on amazon cloud spot instances,” in Proc. of the ACM SIGCOMM 2011 conference, ser. SIGCOMM ’11, 2011, pp. 242–253.
[43] M. Hovestadt, O. Kao, A. Kliem, and D. Warneke, “Evaluating adaptive compression to mitigate the effects of shared I/O in clouds,” in IPDPSW ’11, May 2011, pp. 1042–1051.
[44] X. Lin, Y. Mao, F. Li, and R. Ricci, “Towards fair sharing of block storage in a multi-tenant cloud,” in HotCloud ’12, Jun. 2012.
[45] H. Librat, P. Costa, T. Karagianis, and A. Rowstron, “Towards predictable datacenter networks,” in Proc. of the ACM SIGCOMM 2011 conference, ser. SIGCOMM ’11, 2011, pp. 242–253.
[46] M. Hovestadt, O. Kao, A. Kliem, and D. Warneke, “Evaluating adaptive compression to mitigate the effects of shared I/O in clouds,” in IPDPSW ’11, May 2011, pp. 1042–1051.
[47] X. Lin, Y. Mao, F. Li, and R. Ricci, “Towards fair sharing of block storage in a multi-tenant cloud,” in HotCloud ’12, Jun. 2012.