A Declarative Optimization Engine for Resource Provisioning of Scientific Workflows in IaaS Clouds

Amelie Chi Zhou[1], Bingsheng He[1], Xuntao Cheng[2], and Chiew Tong Lau[1]

[1]School of Computer Engineering, Nanyang Technological University, Singapore

[2]LILY, Interdisciplinary Graduate School, Nanyang Technological University, Singapore

ABSTRACT

Resource provisioning for scientific workflows in Infrastructure-as-a-service (IaaS) clouds is an important and complicated problem for budget and performance optimizations of workflows. Scientists are facing the complexities resulting from severe cloud performance dynamics and various user requirements on performance and cost. To address those complexity issues, we propose a declarative optimization engine named *Deco* for resource provisioning of scientific workflows in IaaS clouds. Deco allows users to specify their workflow optimization goals and constraints of specific problems with an extended declarative language. We propose a novel probabilistic optimization approach for evaluating the declarative optimization goals and constraints in dynamic clouds. To accelerate the solution finding, Deco leverages the available power of GPUs to find the solution in a fast and timely manner. We evaluate Deco with several common provisioning problems. We integrate Deco into a popular workflow management system (Pegasus) and show that Deco can achieve more effective performance/cost optimizations than the state-of-the-art approaches.

Categories and Subject Descriptors

C.4 [**Computer Systems Organization**]: Performance of Systems

Keywords

Cloud; Resource Provisioning; Scientific Workflow

1. INTRODUCTION

Workflow models have been widely used by scientists to manage and analyze large-scale scientific applications in many research fields. For example, Montage workflow [28] is an example in astronomical study for generating sky mosaics. Workflow management systems (WMSes) are often used to support the execution of scientific workflows, for example, deciding which task runs on which resource. Such resource orchestration is a major component of WMSes and is important to the performance of workflows. Most of the WMSes, such as Pegasus [13] and Kepler [23], are originally designed for grid and local cluster environments. Their resource orchestration components (or schedulers) are mainly designed for performance optimizations.

Due to the pay-as-you-go benefits, many scientific workflow applications are recently deployed and executed on the infrastructure-as-a-service (IaaS) clouds [4, 40]. "Science clouds" become an emerging and promising platform for next-generation scientific computing [39, 29, 19]. New resource orchestration methods have been proposed to decide which task runs on which *instance* (or virtual machine, VM) in order to optimize the execution time, monetary cost or both. Since an IaaS cloud provider offers different types of instances and pricing models, existing WMSes have to be redesigned for the cloud environment. Particularly, they did not well address the following features in the cloud: various user requirements and cloud dynamics.

User requirements: In the cloud environment, scientists can have different or evolving requirements on the budget/performance goals and constraints. A user may want to minimize the execution time of a workflow on a cloud C1 with a pre-defined budget. In another scenario, she may consider running the workflow on multiple clouds besides C1. At this point, the optimal solution depends on the offerings of the multiple clouds and the network performance across clouds. For different problems, custom-designed approaches with problem-specific heuristics are proposed to find a suitable solution [25, 26]. For example, Mao et al. [25] use a series of heuristics for minimizing the monetary cost while satisfying the performance requirement of individual workflows (denoted as Autoscaling). Later, they have proposed a very different set of heuristics to optimize the performance with the budget constraint [26]. New algorithms for workflow optimizations are still emerging as clouds and applications evolve.

Different resource provisioning schemes result in significant monetary cost and performance variations. Figure 1 shows the normalized average cost of running Montage workflow with deadline constraint using different instance configurations on Amazon EC2. More details about experimental setup can be found in Section 6. We consider seven scenarios: the workflow is executed on a single instance type only (m1.small, m1.medium, m1.large and m1.xlarge), on randomly chosen instance types, and using the instance configurations decided by Autoscaling [25] and by this paper (denoted as Deco). Although the configurations m1.small and m1.medium obtain low average cost, they can not satisfy the performance constraint of the workflow. Among the configurations satisfying the deadline constraint, Deco obtains the lowest monetary cost. The cost obtained by Deco is only 40% of the cost obtained by the most expensive configuration (i.e., m1.xlarge).

Cloud dynamics: As a shared infrastructure, the performance dynamics of the cloud have made the problem much complicated. Most resource provisioning approaches for scientific workflows in IaaS clouds [24, 7, 14] assume that the execution time of each task in the workflow is static on a given VM type. However, this

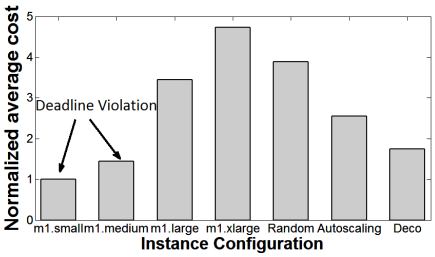


Figure 1: Average cost of running Montage workflows under different instance configurations on Amazon EC2.

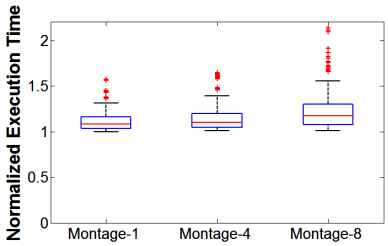


Figure 2: Execution time variances of running Montage workflows on Amazon EC2.

assumption does not hold in the cloud, due to the performance dynamics in the cloud mainly caused by I/O and network interferences [33]. Figure 2 shows the quantiles of the normalized execution time of the Montage workflows in different scales running on Amazon EC2 for 100 times each. The instance configurations of each workflow are optimized by Deco. The execution time of the three workflows varies significantly. The variances are mainly from the interferences from disk and network I/O. In fact, scientific workflows may process input data of a large size. For example, an Epigenomics workflow typically processes input data of dozens of GB, and Montage and Ligo on hundreds of GB [18]. Due to the significant performance variance of scientific workflows in IaaS clouds, the deterministic notions of performance/cost constraints are not suitable, and a more rigorous notion is required.

In this paper, we address the above two problems in WMSes. Specifically, we propose a declarative optimization engine called *Deco*, which can automatically generate resource provisioning plan for scientific workflows in the cloud, considering the cloud performance dynamics. Many WMSes, such as Pegasus and Kepler, allow users to define their own resource scheduling policies. In this study, Deco works as an alternative to the user-defined *callouts* inside the WMS. With Deco, users only need to declaratively specify their performance/monetary cost goals and constraints and leave the rest to the optimization engine. Particularly, Deco uses a declarative language *WLog* which extends ProLog and supports a novel probabilistic notion of performance/monetary cost constraints to capture cloud dynamics. For example, a user can specify a probabilistic deadline requirement of $p\%$ so that the p-th percentile of the workflow execution time distribution in the target IaaS cloud is no longer than the pre-defined deadline. Similar probabilistic definitions have been used in computer networks and real-time systems [1]. Given a WLog program, Deco formulates the problem of finding a good solution as a search problem or even an $A^\star$ search problem for better efficiency (if users offer some application specific heuristics to prune optimization space).

Considering the features of our problem as well as the hardware features of modern accelerators, Deco leverages the power of the GPU to find the solution in a fast and timely manner. We implement Deco into a popular WMS (Pegasus [13]), although the design of this engine can be generally applied to other WMS.

We have applied Deco to three representative workflow optimization problems, which allow us to showcase the key features of Deco: 1) the workflow scheduling problem [25], 2) the workflow ensemble problem [24] and 3) the follow-the-cost problem across multiple clouds [36]. Our experiments show that, 1) Deco can significantly reduce the complexity of developing the three resource provisioning algorithms in the cloud; 2) Deco can achieve more effective performance/cost optimizations than state-of-the-art approaches; 3) The GPU-based acceleration improves the response time of getting the solution. As a result, the optimization overhead of Deco takes 4.3-63.17 ms per task for a workflow with 20-1000 tasks. That means, our optimization engine is practical for timely resource provisioning of workflows.

We summarize the main contributions of this paper as follows:

- *Declarative optimization engine.* Deco supports a novel declarative language for scientific workflow optimization problems in IaaS clouds, which combines special constructs of workflows and dynamics of cloud environment. Our engine supports both probabilistic and deterministic optimizations, while the existing approaches only support deterministic optimizations.
- *WMS integration.* We have developed a memory- and performance-optimized GPU-based implementation for Deco and integrated it into a popular WMS.
- *Use cases.* We have applied Deco to three representative use cases and the evaluation results have shown the effectiveness of Deco on these workflow optimization problems. Beyond these three use cases, we envision Deco has the potential to solve a wide class of workflow optimization problems.

The rest of this paper is organized as follows. Section 2 presents the overview of our system. We present three motivating examples in Section 3. Sections 4 and 5 present the major components of our system, including the declarative language and the GPU-accelerated solver. We evaluate the proposed optimization engine in Section 6 and review the related work in Section 7. We finally conclude this paper in Section 8.

2. SYSTEM OVERVIEW

WMSes [13, 23, 35] are often used by scientists to execute and manage scientific workflows. Those workflow management systems often have dependent software tools such as Condor and DAGMan [11], and require specific skills to implement the specific optimization algorithms in the cloud. All those software packages are interplayed with the resource provisioning problem in the cloud. It is desirable to abstract these complexities from users and shorten the development cycle. In this paper, we develop a declarative resource provisioning engine named Deco and integrate it into a popular WMS named Pegasus for executing scientific workflows in IaaS clouds. Figure 3 presents a system overview of Deco and its integration in the Pegasus WMS.

When using Pegasus to manage and execute workflows in the cloud, users submit workflows to Pegasus with DAX files, which describe workflows in the XML format. An example DAX file in Figure 4 describes a pipeline workflow. A job[1] XML element describes a task in the workflow, including its executable file (e.g., "process1" for task "ID01"), input and output files (e.g., "f.a" and "f.b1" respectively for task "ID01"). ⟨child⟩ and ⟨parent⟩ elements show the dependency between tasks. For example, in the pipeline workflow, task "ID02" takes the output of task "ID01" (i.e., file

[1]The term "job" is slightly misleading in the DAX file. This paper uses "task" to describe the minimum execution unit in a workflow, instead of "job" .

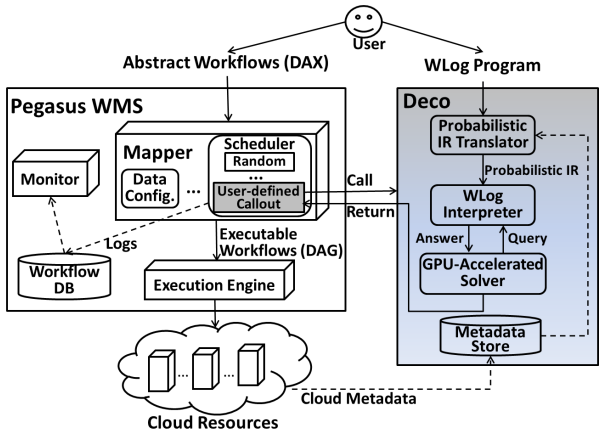


Figure 3: System Overview of Deco with integration in Pegasus

```
<!-- definition of all jobs -->
<job namespace="pipeline" name="process1" id="ID01">
<uses name="f.b1" link="output" />
<uses name="f.a" link="input" />
</job>
<job namespace="pipeline" name="process2" id="ID02">
<uses name="f.b2" link="output"/>
 <uses name="f.b1" link="input" />
</job>
<!-- list of control-flow dependencies -->
 <child ref="ID02"> <parent ref="ID01" /> </child>
```

Figure 4: An example DAX file.

"f.b1") as its input and thus is the child task of task "ID01". The mapper component of Pegasus takes the DAX file as input to generate executable workflows. An executable workflow contains information such as where to find the executable file of a task and which site the task should execute on. The execution engine of Pegasus distributes executable workflows to the cloud resources for execution.

In order to schedule the workflows in the cloud, users can alternatively choose from several traditional schedulers provided by Pegasus and our proposed Deco. For example, Pegasus provides a Random scheduler by default, which randomly selects the instance to execute for each task in the workflow. With Deco, we model the resource provisioning problem as a constrained optimization problem. Users can specify various optimization goals and constraints with *WLog* programs. WLog is a declarative language extended from ProLog, with special designs for scientific workflows and the dynamic clouds. Deco allows users to use *probabilistic* notions to specify their optimization requirements in the dynamic clouds. We model the dynamic cloud performance with probabilistic distributions, which is transparent to users. Deco automatically translates a WLog program submitted by users to probabilistic intermediate representation (IR) and interpret it using the WLog interpreter. We traverse the solution space to find a good solution for the optimization problem. For each searched solution, we evaluate it with the probabilistic IR, which requires a lot of computation [12]. To effectively and efficiently search for a good solution in a reasonable time, we implement a GPU-accelerated parallel solver to leverage the massive parallelism of the GPU. After the optimization process, Deco returns the found resource provisioning plan (indicating the selected execution site for each task in the workflow) to Pegasus for generating the executable workflow.

3. USE CASES

We present three use cases of resource provisioning problems for workflows in IaaS clouds, and demonstrate their common components to motivate our design of Deco. Those three use cases cover different aspects of typical workflow optimization problems. The first case addresses the resource provisioning for a single workflow, and the second case is for workflow ensembles (multiple workflows as a group). Both the first and the second cases can be performed in an offline setting. In the third case, we consider a dynamic workflow optimization problem, which makes workflow migration decisions across multiple clouds at runtime. Different from the former two cases, it assesses the efficiency of Deco. For each use case, we formally formulate the optimization goal and constraints of the problem. We present the declarative language specifications and detailed WLog programs of the use cases in later sections.

3.1 Workflow Scheduling Problem

In IaaS clouds, users rent instances to run workflows and pay for the instance hours accordingly. Since the cloud provides many types of instances with different capabilities and prices, the workflow scheduling problem is to select a good instance type for each task of the workflows, satisfying users' optimization requirements. While existing scheduling approaches use custom-designed heuristics for specific user requirements [25, 26], they are unlikely to work well upon changing optimization goals and constraints.

Different from the existing approaches, Deco takes the user requirements (i.e., optimization goals and constraints) as input and can adaptively work for a range of optimization problems. An example optimization goal is to minimize the average monetary cost of a workflow while satisfying its deadline requirement. Due to the cloud performance dynamics, we adopt the probabilistic deadline requirement notion for this problem. The probabilistic deadline requires that the p-th percentile of the workflow execution time distribution is no longer than D.

We assume the workflow has N tasks with IDs $0,1,\ldots,N-1$, and the cloud provider offers K types of instances with IDs $0,1,\ldots,K-1$. The optimization variable of this problem is vm_{ij}, which means assigning task i to instance type j. The value of vm_{ij} is 1 (i.e., task i is assigned to instance type j) or 0 (otherwise). Note, for each task i ($i = 0,1,\ldots,N-1$), only one of the variables vm_{ij} ($j = 0,1,\ldots,K-1$) can be 1. We denote the execution time distribution of task i on instance type j as $T_{ij}(t)$, which means $T_{ij}(t) = P(t_{ij} = t)$. That is, the probability that task i has execution time t on instance type j is $T_{ij}(t)$. The unit price of instance type j is U_j. We calculate the overall monetary cost of the workflow by summing up the average monetary cost of each task in the workflow, where the average monetary cost of a task is calculated using its mean execution time (M_{ij}) and the unit price of its assigned instance (see Equation (1)). We calculate the overall execution time of the workflow by summing up the execution time of all tasks on the critical path (denoted as CP) of the workflow (see Equation (3)). Formally, we express the problem as below:

$$\min \sum_i \sum_j (M_{ij} \times U_j \times vm_{ij}) \quad (1)$$

$$M_{ij} = \sum_{t=0}^{D} (T_{ij}(t) \times t) \quad (2)$$

subject to:

$$P(t_w \leq D) \geq p, \text{ where } t_w = \sum_{i \in CP} \sum_j (t_{ij} \times vm_{ij}) \quad (3)$$

3.2 Workflow Ensemble

Many scientific workflow applications have a group of workflows with similar structure but different parameters (e.g., input data, number of tasks). These groups of workflows are called workflow ensembles [24], in which each workflow is associated

with a priority to indicate its importance in the group and has a pre-defined deadline constraint. There is also an overall budget constraint for executing the entire ensemble. Such a resource provisioning problem aims to execute as many high-priority workflows as possible using limited resources. Given the budget and deadline constraints, the optimization goal is to maximize the total score of completed workflows in an ensemble e (see Equation (4)). The score of a workflow is proportional to its priority. We denote the priority of a workfow w as $Priority(w)$ (the highest-priority workflow has $Priority(w) = 0$, the next highest workflow has $Priority(w) = 1$, and so on). We use $CMP(e)$ to represent the set of completed workflows in the assemble e. Each workflow w in the ensemble has a probabilistic deadline constraint, which requires that the p_w-th percentile of the workflow execution time distribution is no longer than D_w (see Equation (6)). The budget constraint is set for the entire ensemble, which requires that the overall monetary cost of all workflows in ensemble e is smaller than the pre-defined budget B (see Equation (5)). We can formulate this problem as below:

$$\max \sum_{w \in CMP(e)} 2^{-Priority(w)} \tag{4}$$

subject to:

$$\sum_{i \in w, w \in e} \sum_j (M_{ij} \times U_j \times vm_{ij}) \leq B \tag{5}$$

$$\forall w \in CMP(e), \quad P(t_w \leq D_w) \geq p_w \tag{6}$$

3.3 Follow-the-Cost

Our third use case is a dynamic workflow migration problem called follow-the-cost. We consider migrating multiple workflows among multiple cloud platforms to optimize the cost. We consider a task as the smallest migration unit. Different clouds may have different pricing schemes. Even within the same cloud provider, the pricing in multiple data centers may be different. For example, in Amazon EC2, prices of instances in the Singapore region are higher than those of the same type in the US East region. Migrating a partially executed workflow to another more cost-efficient cloud can reduce the monetary cost. However, the migrations induce extra cost on transferring intermediate data between clouds. We denote the monetary cost on the execution and migration of a task i as EC_i and MC_i, respectively. If a task is migrated from one cloud to another, the migration cost is the networking cost spent on transferring necessary data for executing the task. The goal of this problem is to minimize the overall monetary cost of all workflows while satisfying the deadline constraint of each individual workflow.

Due to cloud performance dynamics, the migration decisions have to be made at runtime for the trade-off between the monetary cost on execution and migration. Because of the light-weight characteristic of Deco, our engine is able to effectively handle this type of dynamic optimization problems. To assess runtime optimization, we use the traditional static deadline notion for this problem, which is that the expected execution time of each workflow w is no longer than a pre-defined deadline D_w. We denote the runtime execution time of a task i on instance type j as t_{ij}. $Unfinished(sw)$ denotes all unfinished tasks in the set of workflows sw. The optimization parameter of this problem is G_i^{mn}, meaning migrating a task i from a cloud m to another cloud n ($n \neq m$). We denote the transferred data size for task i as $data_i$, the network bandwidth between the original and migrated clouds as $Band_{mn}$ and the networking price between the two clouds as K_{mn}. Given the above variables, we formulate the problem as follows:

$$\min \sum_{i \in Unfinished(sw)} (EC_i + MC_i) \tag{7}$$

$$EC_i = \sum_j (t_{ij} \times U_j \times vm_{ij}) \tag{8}$$

$$MC_i = \sum_n (data_i \times K_{mn} \times G_i^{mn}) \tag{9}$$

subject to:

$$\forall w \in sw, \sum_{i \in CP(w)} (\sum_j t_{ij} \times vm_{ij} + \sum_n \frac{data_i}{Band_{mn}} \times G_i^{mn}) \leq D \tag{10}$$

Observations on Use Cases. We have the following observations. First, there are some common optimization requirements in the workflow optimization problems, such as the deadline and budget constraints. Second, the workflow structures and the data dependencies between tasks make the optimization problem much complex. Third, the variate cloud offerings and dynamic cloud performance add another dimension for the resource provisioning problems. Thus, it is feasible and also desirable to attempt common constructs for different resource provisioning problems of workflows in IaaS clouds

4. PROBABILISTIC SPECIFICATION LANGUAGE

WLog is designed as a declarative language for resource provisioning of scientific workflows in IaaS clouds. The major goal is to ease the programming effort of users. WLog is extended from ProLog [22]. ProLog is a powerful declarative language that allows users to define their constraint logics in a very convenient manner. We can leverage efficient techniques in ProLog such as ordering and cut pruning [30] to efficiently solve optimization problems. Users can also declare their own predicates to improve the search efficiency. For example, in Deco, we allow users to offer hints to our solver so that Deco can leverage the efficient $A^\star$ search. We model the cloud dynamics with probabilistic distributions and offer probabilistic notions for users to specify their optimization requirements. Some extensions of ProLog, such as the probabilistic extension [12], can be very useful for specifying the workflow optimization problems in dynamic clouds.

In the remainder of this section, we present the details on WLog language, by using a concrete example on the workflow scheduling problem. Additional examples on the workflow ensemble and follow-the-cost problems are presented in the appendix of our technical report [47].

4.1 ProLog Conventions

Following the conventions in ProLog, a WLog program consists of a set of declarative rules. Each rule has the form of `h :- c1, c2, ..., cn.`, which means that "clauses `c1` and `c2` and ... and `cn` implies `h`". Here, commas connecting the clauses represent logical "AND". `h` is the *head* of the rule and `c1, c2, ..., cn` constitutes the *body* of the rule. A rule with empty body is called a *fact*. Rules can refer to one another recursively until a fact is reached. The order of the rules appearing in a program as well as the order of the predicates in a rule are not semantically meaningful, but affect the efficiency of interpreting a program. Conventionally, the names of variables begin with upper-case letters while constant names start with lower-case letters.

Prolog offers many built-in predicates, such as the ones for arithmetic operations (e.g., `is`, `max` and `sum`) and the ones for list-based operations (e.g., `setof`, `findall`). These built-in predicates help users to write neat and concise WLog programs. We highlight the Prolog built-in predicates in blue color to differentiate from other terms. For example, one may write a rule in Example 1 (see Section 4.3):

```
cost(Tid,Vid,C) :- price(Vid,Up), exetime(Tid,Vid,T),
configs(Tid,Vid,Con), C is T*Up*Con.
```

Table 1: Workflow and cloud specific built-in functions and keywords in WLog.

Function/Keyword	Remark
`goal`	Optimization goal defined by the user.
`cons`	Problem constraint defined by the user.
`var`	Problem variable to be optimized.
`import(daxfile)`	Import the workflow-related facts generated from a DAX file.
`import(cloud)`	Import the cloud-related facts from the cloud metadata.
`budget(p, b)`	A probabilistic budget requirement that the p-th percentile of workflow monetary cost distribution is no larger than b.
`deadline(p, d)`	A probabilistic deadline requirement that the p-th percentile of workflow execution time distribution is no longer than d.
`enabled(astar)`	The $A^\star$ heuristic is enabled for efficiently finding solutions.

where `price(Vid,Up)` stores the unit price of instance `Vid` into the unit price variable `Up`, `configs(Tid,Vid,Con)` indicates that task `Tid` runs on instance of type `Vid` and `exetime(Tid,Vid,T)` stores the execution time of task `Tid` on instance `Vid` into the execution time variable `T`. This rule calculates the monetary cost (`C`) of executing task `Tid` on instance `Vid`.

4.2 WLog Extensions

We extend ProLog in two major aspects to custom it for solving resource provisioning problems of scientific workflows. The first kind is to extend ProLog for workflow processing and IaaS clouds (i.e., cloud offerings and dynamic cloud performance). Given the dynamic cloud performance, WLog offers the probabilistic notion of specifying performance and budget goals. The second kind is to balance the design intentions on the expressive power and simplicity. For instance, WLog offers high-level workflow specific built-ins like `budget` and `deadline` to simplify the expressions.

WLog provides keywords for users to specify their optimization problems, as shown in Table 1. The `goal` keyword indicates the optimization goal. The `cons` keyword indicates the optimization constraints. The `var` keyword indicates the variables to be optimized. WLog also provides several workflow-specific and cloud-specific built-in functions and keywords.

Workflow- and cloud-specific facts. We allow users to "import" two kinds of facts from external workflow abstraction files (DAX files in this study), or by specifying which cloud to use. This functionality saves users a lot of time on writing the tedious facts related to workflows and clouds, and also makes the WLog programs neat and easy to read/maintain.

From the DAX file, Deco imports the common facts of the workflow, including workflow structure, input/output data of each task in the workflow, and the executable file of the task. For example, from the DAX file shown in Figure 4, a workflow structural fact `child(ID01, ID02)` (representing task "ID02" is the child task of task "ID01") can be imported.

The built-in predicate `import(cloud)` covers the cloud-related facts, such as the cloud provided instance types, the pricing for each type of instances and their capabilities. For example, we can construct a fact for an Amazon EC2 instance type, m1.small: ⟨ `key="id1", cloud="ec2", instype="m1.small", price="0.044", cpu="1", mem="1.7", ...`). Some properties such as price and storage are constants, and other properties such as I/O and network performance can be modeled with some probabilistic distributions [48]. For example, the random I/O performance of the m1.medium instance type of Amazon EC2 can be modeled with a normal distribution (see Section 6 for details on the distributions and the way of obtaining them). For each dynamic performance component (i.e., network and I/O), we discretize the probabilistic performance distributions as histograms, and store the histograms in the *metadata store*. We have developed some micro-benchmarks and periodically perform calibrations on the target cloud, which is totally transparent to users.

Constraint built-ins. We find that *budget* and *deadline* are the two most common constraints in workflow optimizations on IaaS clouds [25, 26, 24]. WLog supports built-in functions `budget` and `deadline` with probabilistic notions. For example, a user can submit a probabilistic deadline requirement that the 95-th percentile of the workflow execution time distribution is no longer than 10 hours. This requirement can be easily expressed as `deadline(95%, 10h)` in WLog.

We also support the `enabled` built-in function, which enables more efficient $A^\star$ search algorithm when searching for a good solution. We explain the usage and benefit of this function in Section 5.

4.3 A Concrete Example

In Example 1, the WLog program implements a workflow scheduling problem (described in Section 3.1) that aims at minimizing the total monetary cost of executing a Montage workflow in Amazon EC2 cloud while satisfying user's probabilistic deadline requirement `deadline(95%, 10h)`.

Example 1. WLog program for workflow scheduling.

```
        import(amazonec2).
        import(montage).
goal    minimize Ct in totalcost(Ct).
cons    T in maxtime(Path,T) satisfies deadline(95%,
        10h).
var     configs(Tid,Vid,Con) forall task(Tid) and
        vm(Vid).

        /*calculate the time on the edge from X to Y*/
r1      path(X,Y,Y,Tp) :- edge(X,Y), exetime(X,Vid,T),
        configs(X,Vid,Con), Con==1, Tp is T.
        /*calculate the time on the path from X to Y,
        with Z as the next hop for X*/
r2      path(X,Y,Z,Tp) :- edge(X,Z), Z\==Y,
        path(Z,Y,Z2,T1), exetime(X,Vid,T),
        configs(X,Vid,Con), Con==1,Tp is T+T1.
        /*calculate the time on the critical path from
        root to tail*/
r3      maxtime(Path,T) :- setof([Z,T1],
        path(root,tail,Z,T1), Set), max(Set, [Path,T]).
        /*calculate the cost of Tid executing on Vid,
        where T includes the instance up time for data
        transfer*/
r4      cost(Tid,Vid,C) :- price(Vid,Up),
        exetime(Tid,Vid,T), configs(Tid,Vid,Con), C is
        T*Up*Con.
        /*calculate the total cost of all tasks*/
r5      totalcost(Ct) :- findall(C, cost(Tid,Vid,C),
        Bag), sum(Bag, Ct).
```

Program description. The program takes `task(Tid)`, `vm(Vid)`, `edge(X,Y)`, `exetime(X,Vid,T)` and `price(Vid,Up)` as input, which are imported from the workflow DAX file and the cloud metadata. The first two lines of import clauses specify the cloud- and workflow-related facts. `import(amazonec2)` indicates that the workflow is executed in the Amazon EC2 cloud, and the Amazon EC2 related facts are imported. For example, the `vm` entry stores the unique identifier of a VM type `Vid`. The `price` entry stores the unit price `Up` of instance `Vid`. `import(montage)` imports the workflow-related facts from the Montage DAX file, such as `task` and `edge`. Each `task` entry stores the unique identifier of a task `Tid`. The `edge` entry stores pairs of connected tasks `X` and `Y` (i.e., task `X` is the parent of task `Y`). `exetime` stores the execution time `T` of a task `X` on a VM `Vid`. Given these input entries, the program expresses the following. The WLog program is essentially equivalent to the optimization problem that we have formulated in Section 3.1.

- **Optimization goal:** Minimize the monetary cost `Ct` of executing a workflow in `totalcost`.
- **Constraints:** Satisfy the probabilistic deadline constraint that the 95-th percentile of the workflow execution time distribution is no longer than 10 hours.
- **Variables:** `configs(Tid,Vid,Con)` means configuring task `Tid` to instance type `Vid`. The value of *Con* is 1 if `Tid` is assigned to `Vid` and 0 otherwise. A solution to the workflow scheduling problem is a list of instance configurations for each task in the workflow.
- **Derivation rules:** Rule `r1` to `r3` calculate the overall execution time (or makespan) of a workflow. `path(X, Y, Z, Tp)` calculates the execution time (`Tp`) from task `X` to task `Y`, with task `Z` as the next hop of `X`. `maxtime(Path,T)` stores the execution time on the critical path of the workflow, i.e., the makespan of the workflow, in `T` (as in Equation (3) in Section 3.1). Note, we add task `root` and `tail` as two virtual tasks to represent the start and end of the workflow, respectively. Rules `r4` and `r5` calculate the overall monetary cost by summing up the monetary cost of all the tasks (as in Equation (1) in Section 3.1).

5. PROVISIONING PLAN GENERATION

This section describes the process of generating resource provisioning plan from WLog programs. A WLog program represents a constrained optimization problem for the resource provisioning of scientific workflows. When a WLog program is submitted, Deco first translates it into a *probabilistic* intermediate representation (IR) to capture the dynamics of the IaaS clouds. A WLog interpreter and a solver are implemented, where the interpreter is used to evaluate each solution searched by the solver with the probabilistic IR. Each searched solution is a potential resource provisioning plan for the optimization problem and the best found solution is returned as the output.

The interpretation of the probabilistic IR can induce high runtime overhead due to the large solution space in the workflow resource provisioning problem. In order to balance the overhead and the solution optimality, we implement the interpreter and solver with massive parallel implementations to search for a good provisioning plan efficiently. Particularly, the solver performs generic search by default and more efficient $A^\star$ search if users have specified search heuristics in the WLog program. Despite the previous studies on formulating constrained optimization problem as a search problem [3, 21], Deco has the following unique features.

First, the generic search in Deco is specifically designed for workflows and IaaS cloud features. The state transition in the solution space is driven by some common workflow transformation operations [46]. Second, the evaluation of each solution is based on the probabilistic IR of WLog programs to capture the dynamic cloud performance. We also leverage the available GPU power to accelerate the probabilistic evaluations. Note, the previous engines (e.g., [13]) can only support deterministic optimizations.

5.1 Probabilistic IR Translation

Due to cloud dynamics, we adopt probabilistic notion for the optimization goals and constraints. To accurately evaluate the probabilistic goals and constraints, given a WLog program, we first translate it into a probabilistic IR.

In order to generate the probabilistic IR, we first obtain the performance dynamics of the target cloud from the metadata store. We collect the task-related profiles from the specified DAX file to estimate the execution time of each task in the workflow. In Deco, we use an existing task execution time estimation approach for workflows [43]. The basic idea is that, given the input data size, the CPU execution time (we can define a scaling factor to scale the CPU time in multi-core system [31]) and the 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, since the I/O and network performance of the cloud are dynamic, the estimated task execution time is also a probabilistic distribution.

With the obtained cloud performance dynamics, we generate the probabilistic IR by translating each rule in the WLog program following the syntax in ProbLog [12]. Each rule in the probabilistic representation of a WLog program is in the form `p : h :- c1, c2, ..., cn.`, where `p` states the probability of the rule being true. For example, with the cloud performance histogram metadata, the execution time of task `Tid` on instance type `Vid` is represented by p_j `: exetime(Tid,Vid,`T_j`)`, where $j = 1, \ldots, n$ and n is determined by the number of bins in the performance histogram. Consequently, the rule shown in Section 4.1 can be translated to the probabilistic representation as below:

p_j `: cost(Tid,Vid,`C_j`) :- price(Vid,Up), exetime(Tid,Vid,`T_j`), configs(Tid,Vid,Con),` C_j `is` T_j`*Up*Con.`

Support for deterministic goals and contraints. Deco uses a WLog interpreter to evaluate each searched solution using the translated probabilistic IR, as we shall see later in this section. On the other hand, Deco also supports deterministic optimization goals and constraints, which are common in dynamic optimization problems such as Follow-the-cost. In order to provide a uniformed interface, we translate WLog programs with deterministic optimization requirements as follows: we obtain the runtime cloud performance from the metadata store and translate each WLog rule with probability of 1.0 in the same way as mentioned above.

5.2 Probabilistic IR Evaluation

The WLog interpreter answers the queries submitted by the solver to evaluate the quality of each solution found by the solver. There are two kinds of WLog queries, i.e., the ones on optimization goals (querying the value of the goals) and the ones on optimization constraints (querying whether a constraint is satisfied). Each WLog query is evaluated with the probabilistic IR, by calculating the probability that the query succeeds.

The probabilistic IR of WLog inherits many features from ProbLog [12], and we interpret the probabilistic IR in the way similar to the interpretation of ProbLog. Specifically, all rules in the probabilistic IR are denoted with an array Rule[1,...,n], where n is the number of rules in the program. The unification technique of ProLog [30] is adopted in the interpreter to find a series of proofs with certain succeeding probability for each submitted query. Exact inference for a query in ProbLog is quite complex or even impossible for large programs, as the number of proofs to be checked for the inference grows exponentially [20]. ProbLog provides several approximation methods [20] to reduce the inference time complexity and we adopt the Monte Carlo approximation which can benefit from the massive parallelism of GPU computations.

Given the probabilistic IR, Algorithm 1 describes the general process of answering a query q with WLog interpreter. In Line 1-4, we recursively look for the proofs of query q, using the unification policies defined in ProLog. Specifically, the *match* function uses the unification policies to find a match between the head of query q and the head of the rules in the probabilistic IR. If a match is found (Line 3), we generate a new query with the body of the matched rule and the body of q remaining to be matched. We recursively look for a match for the generated query. The function *append(X,Y)* concatenates two terms X and Y. When the newly generated query is nil, it means all terms of q have been answered.

In this way, we find a proof to query q. Given the found proofs and the probability p_i of each rule Rule[i], in Line 7-15, we use the Monte Carlo method [34] to calculate the approximate inference of the query. Specifically, *Max_iter* realizations are sampled from the found proofs. For each sample, we calculate the probability of the proof being true using the probabilities of the rules forming the proof. If query q is a query on constraint, we return the average probability of the constraint being true in the *Max_iter* samples. Otherwise, we return the mean value of the optimization goal as the answer to q.

GPU-accelerated evaluation implementation. GPUs can support a large number of threads running in parallel, but a single thread is not as powerful as a conventional CPU thread. GPU threads are grouped in *thread blocks*, where threads in the same block can cooperate via *shared memory*. Threads from different blocks communicate via the global memory and the data access bandwidth degrades a lot compared to the shared memory. We follow the following principles during the GPU implementation: 1) the work assigned to each thread should be light-weight; 2) threads in the same block should cooperate to reduce redundant computations; 3) communications across thread blocks should be avoided. Overall, our implementation features the GPU hardware features.

We adopt Monte Carlo approximation for the probabilistic IR evaluation. Since the work in one Monte Carlo iteration is light-weight, we use one GPU thread for each Monte Carlo iteration. We store the temporary results of each thread into the shared memory for fast synchronization.

Algorithm 1 Query evaluation function $WLogInterp(q)$.

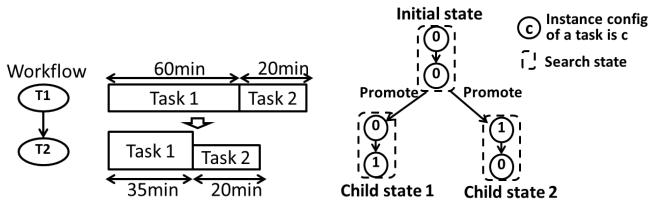
Require:
Probabilistic IR with goal g, variable *var*;
n rules: $p_i : Rule[i]$. where $i = 1,\ldots,n$;
m probabilistic constraints: $Cons(1,\ldots,m)$;
Query request: q.

1: **if** q is **not** *nil* **then**
2: **for** $i = 1$ **to** n **do**
3: **if** $match(head(Rule[i]), head(q))$ **then**
4: $WLogInterp(append(body(Rule[i]), body(q)))$;
5: **else**
6: $EvalResult = 0$;
7: **for each** Monte Carlo iteration in *Max_iter* iterations **do**
8: Randomly generate a realization from the found proofs and calculate the probability of the realization being true as α;
9: **if** q is a probabilistic constraint of the program and the inference of q from the sampled realization is true **then**
10: $EvalResult \mathrel{+}= \alpha$;
11: **else**
12: **if** q is a query on the optimization goal **then**
13: Infer the result r of the query from the sampled realization;
14: $EvalResult \mathrel{+}= r \times \alpha$;
15: $EvalResult \mathrel{/}= Max\_iter$;
16: **return** $EvalResult$;

5.3 Parallel Solver Design

Generic Search. The key challenge of designing a generic search is how to have a generic representation on the state as well as state transitions. Each state is a solution to the optimization problem. We evaluate each state using the probabilistic IR and formulate the search as a traversal of the search tree from the initial state. In Example 1, a state in the search tree is an instance configuration plan, where `configs(Tid, Vid, Con)` stores the instance configuration of each task `Tid` in the workflow.

We need a systematic and generic approach to define the transition between states in order to explore all the feasible states. As a start, we adopt the transformation operations [46] as transitions to traverse the search tree. In our previous work, we propose six general transformation operations for workflows, including Move, Merge, Promote, Demote, Split and Co-Scheduling. Suppose each task has an initial instance configuration (there are several ways to determine the initial configurations, see [46]). The transformation defines the change of instance configuration and the starting execution time of each task. We briefly describe the six transformation operations, and illustrate how we use them with an example.



(a) Promote operation. (b) Children of the initial state.

Figure 5: Examples of using transformation operations in the generic search.

The Move operation delays the execution of a task to a later time. The Merge operation merges two tasks with the same configuration onto the same instance to fully utilize the instance partial hour. The Promote/Demote operation changes the configuration of a task to a more/less powerful instance type. The Split operation suspends a running task and resumes its execution at a later time. The Co-scheduling operation assigns multiple tasks with the same instance configuration to the same instance. Figure 5a is an example of the Promote operation in the workflow scheduling problem. Consider a simple workflow with only two tasks. The deadline of the workflow is one hour and the execution time of the workflow under its initial instance configuration violates the deadline requirement. With the Promote operation, we can reduce the execution time of task 1 and satisfy the deadline constraint. Figure 5b shows the child states of the initial state generated by Promote. For example, the initial state configures each task with the cheapest instance type 0 (denoted as "0→0") and each child state configures one task with a better instance type 1 (denoted as "0→1" and "1→0"). Child state 2 is equivalent to the execution illustrated in Figure 5a.

Algorithm 2 General search process from initial state S to goal state D.

Require:
WLog program P with user specified objective *goal* and constraints *cons*;

1: $CurBest = S$;
2: Create a FIFO queue Q and push the initial state into Q;
3: **for each** state *current* in Q **do**
4: Assign the instance configurations in *current* to variables in P;
5: Query $feasibility = WLogInterp(cons)$ and $cur\_cost = WLogInterp(goal)$ from P, as in Algorithm 1;
6: **if** *feasibility* is **true then**
7: **if** $cur\_cost < CurBest.cost$ **then**
8: $CurBest = current$;
9: $CurBest.cost = cur\_cost$;
10: Generate all child states of *current* that have not been visited;
11: Add them into Q;
12: **return** $D = CurBest$;

Algorithm 2 shows the search process of the generic search. Deco evaluates each traversed state to check whether the target state is found. For evaluating a state, we first assign the values specified by the state to the variables in the WLog program and then submit queries regarding the goal and constraints (Line 5). The detailed algorithm of evaluating queries is shown in Algorithm 1.

We consider the balance between exploration and exploitation during the generic search. By exploitation, the depth first search is performed and the good partial solutions found are prioritized. By exploration, the breadth first search is performed and the threads

traverse the search tree individually. A good partial solution does not guarantee global optimality. On the other hand, the parallelism of exploration offers a good opportunity to leverage the power of the GPU and accelerate the search process. Due to the above reasons, for the generic search, we choose exploration over exploitation for parallelism.

$A^\star$ Search. Deco allows users to define heuristic predicates on the good partial solutions. By calling `enabled(astar)` (as shown in Table 1), Deco utilizes $A^\star$ search to benefit from its pruning capability for better search performance. Particularly, users can specify two predicates $cal\_g\_score(s)$ and $est\_h\_score(s)$, which calculate the g score of state s and estimates the h score of state s, respectively. The $g(s)$ score and $h(s)$ score 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. With the two heuristics, we can efficiently prune the solution space by not placing the states with high g and h scores into the candidate list.

Consider the workflow scheduling problem in Example 1, in order to enable $A^\star$ search, the user simply needs to add the following lines in her WLog program.

```
enabled(astar).
cal_g_score(C) :- totalcost(C).
est_h_score(C) :- totalcost(C).
```

In Example 1, both the g score and h score of a state s are the estimated monetary cost of state s. Since the optimization goal is to minimize monetary cost, if the monetary cost of a state is higher than the best found solution, all its child states can be pruned. Child states configure tasks with better instance types and thus always generate higher cost than their parent state, assuming the initial state to be all tasks on the cheapest instance type.

GPU-accelerated search implementation. According to our GPU implementation principles mentioned in Section 5.2, we use one thread block to handle each searched state since there is few data communication between two different states. We use N thread blocks to search the solution space at the same time, where N is the number of multiprocessors in the GPU. When evaluating a searched state, threads in the same block use the Monte Carlo method (see Algorithm 1) to verify the feasibility and calculate the optimization result of the state. Assume there are K threads in one block, ideally we can expect $K \times N$ times speedup comparing the parallel solver implementation to a single thread implementation. Note that, the GPU acceleration is transparent to users, which takes advantage of the available GPU computation power.

6. EVALUATION

We evaluate the performance dynamics observed on real clouds and then evaluate the effectiveness and efficiency of Deco.

6.1 Experimental Setup

We run Deco on a machine with 24GB DRAM, an NVIDIA K40 GPU and a 6-core Intel Xeon CPU. Workflows are executed on Amazon EC2 or a cloud simulator.

Clouds. We calibrate the cloud performance dynamics on Amazon EC2. The measurement results are used to model the probabilistic distributions of I/O and network performance, which are stored in the metadata store and used in `import(cloud)`. Specifically, we measure the performance of CPU, I/O and network for four frequently used instance types of Amazon EC2, namely m1.small, m1.medium, m1.large and m1.xlarge. We find that CPU performance is rather stable in the cloud, which is consistent with the previous studies [33]. In this paper, we focus on the calibration for 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 network bandwidth between any two types of instances with Iperf [17]. In particular, we repeat the performance measurement under each setting once a minute, and each experiment lasts for 7 days (in total 10,000 times). When an instance has been acquired for a full hour, it is released and a new instance of the same type is created to continue the measurement.

Workflows. There have been some studies on characterizing the performance behaviors of scientific workflows [18]. In this paper, we consider three common workflow structures, namely Ligo, Montage and Epigenomics. The three workflows have different structures and parallelism.

We create three instances of Montage workflows with different sizes using Montage source code. Their input data are the 2MASS J-band images covering 1-degree by 1-degree, 4-degree by 4-degree and 8-degree by 8-degree areas retrieved from the Montage archive [27]. We denote them as Montage-1, Montage-4, and Montage-8 accordingly. Initially, the input data are stored in Amazon S3 storage. All intermediate data during workflow executions are stored on local disks and the final output data are stored in S3 for persistence.

Since Ligo and Epigenomics are not open-sourced, we construct synthetic Ligo and Epigenomics workflows using the workflow generator provided by Pegasus [41].

Implementation details. 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 publically 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 [18].

On Amazon EC2, we create an AMI (Amazon Machine Image) installed with Pegasus and its prerequisites such as Condor. We acquire the four measured types of instances using the created AMI. We modify the Pegasus (release 4.3.2) scheduler to integrate Deco, such as scheduling the tasks onto instances of selected types. A script written with Amazon EC2 API is developed for acquiring and releasing instances at runtime.

We develop a simulator based on CloudSim [8]. We mainly present our new extensions, and more details on cloud simulations can be found in the original paper [8]. 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. The Instance component simulates cloud 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.

We implemented three existing algorithms for the three use cases as state-of-the-art comparisons with Deco. Comparing the implementation of the existing algorithms and the WLog programs for the use cases, WLog significantly simplifies the programming for users to solve the workflow optimization problems, and brings the desirable features such as easy maintenance and code readability.

Workflow scheduling problem: We implement Autoscaling [25] as the comparison algorithm for this problem. This approach

utilizes a series of heuristics such as deadline assignment to ensure the deadline while reducing the monetary cost.

Workflow ensemble: SPSS [24] is a state-of-the-art algorithm for scheduling workflow ensembles. SPSS is an offline provisioning and scheduling algorithm with heuristics to reduce resource waste on workflows that cannot be completed.

Follow-the-cost: Due to the large optimization overhead, most existing offline heuristics [21] can hardly solve this dynamic optimization problem at runtime. To make the workflow migration decisions, we design a simple and light-weight approach (denoted as Heuristic). At the offline stage, we consider the price differences among cloud data centers and determine the plan of migrating the workflows from their initial deployed data center to the more cost-efficient one. At runtime, we monitor the task execution time and make migration adjustments when the monitored execution time differs from the estimation by a threshold.

Parameter setting. We present the implementations in Deco and detailed experimental settings for each use case as follows.

Workflow scheduling problem: We study the average monetary cost and elapsed time for executing a workflow on Amazon EC2. Given the probabilistic deadline requirement, we run the compared algorithms for 100 times and record their monetary cost and execution time. For a fair comparison, we set the deadline of Autoscaling according to the QoS setting in Deco. For example, if user requires 90% of probabilistic deadline, the deadline setting for Autoscaling is the 90-th percentile of workflow execution time distribution. In the experiment, we consider the impact of different probabilistic deadline requirements of 90%, 92%, 94%, 96%, 98% to 99.9% (96% by default) on the optimization results. *All the results are normalized to those of Autoscaling [25].*

The deadline of workflows is an important factor for the candidate space of determining the instance configuration. We study the effectiveness of Deco under different deadline requirements. 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.small and m1.xlarge instances, respectively. In the experiments, we vary the deadline parameter from $1.5 \times D_{min}$ (denoted as tight deadline), $\frac{D_{min}+D_{max}}{2}$ (denoted as medium deadline and used as the *default* setting) to $0.75 \times D_{max}$ (denoted as loose deadline) to study its impact on the optimization results. *All results are normalized to those of Autoscaling under the tight deadline.*

Workflow ensemble: In Deco, a state in the search space is implemented as an array of boolean values, where each dimension of the array indicates whether to execute a workflow in the ensemble. We enable the $A^{\star}$ search by specifying the g and h score of a search state s as the Score metric of s. Initially, all dimensions are set to false, meaning that none of the workflows in the ensemble is executed. For state transitions, we consider executing each of the uncompleted workflows in the ensemble to generate child states.

For a fair comparison, we follow the experimental setup in the previous study [24]. Like the previous study [24], we use simulation to perform the comparison studies. We generate in total 180 synthetic workflows with Ligo, Montage and Epigenomics application types, each type with 3 different workflow sizes from 20, 100 to 1000 tasks and 20 different workflow instances for each different setting. We construct five different ensemble types: constant, uniform sorted, uniform unsorted, Pareto sorted and Pareto unsorted. Each ensemble is composed of 30 to 50 workflows. We generate different deadline and budget parameters. We first identify the smallest budget and amount of time required to execute one/all of the workflows in the ensemble and denote them as MinBudget/MaxBudget and MinDeadline/MaxDeadline. We generate 5 different budgets (denoted as Bgt1 to Bgt5, accordingly) and 5 different deadlines (denoted as D1 to D5, accordingly) equally distributed between the range [MinBudget, MaxBudget] and [MinDeadline, MaxDeadline], respectively. The default budget and deadline constraints are fixed at Bgt3 and Dl3, respectively. We perform sensitivity study on the probabilistic deadline requirement by varying it from 90% to 99.9% (96% by default). As for metrics, we compare the average monetary cost of workflows in the ensemble and the total score of completed workflows in ensembles. *All the metrics are normalized to those of SPSS*. Given each budget and deadline pair, we run the compared algorithms for 100 times.

Follow-the-Cost: In Deco, a state in the search space is implemented as an array of integers, where each dimension stands for a migration decision for a workflow. An integer in the array maintains the ID of the data center where the corresponding workflow will be migrated to. Generic search is used in the search engine of Deco. We make the migration decision periodically at runtime.

We consider two regions of the Amazon EC2, namely the US East Region and the Asia Pacific (Singapore) Region. These two regions have different instance prices. For example, the price difference of the m1.small instances is 33%. The number of workflows in each data center is randomly generated between 10 to 50. The threshold parameter for the runtime adjustment is set as 50% by default. We vary the threshold from 10%, 30%, 50%, 70% to 90% to show the trade-off between optimization results and overhead using Montage-8 workflows. As for metrics, we compare the overall monetary cost for workflow executions, including the operational cost spent on instance hours and the networking cost for transferring data across data centers. *All the metrics are normalized to those of Heuristic*.

6.2 Cloud Dynamics

We have observed performance dynamics on both I/O and network performances in the Amazon EC2 cloud. Figure 6a shows the network performance variance of m1.medium instances. The performance varies significantly, where the maximum variance can reach up to 50%.

Another important finding is that, we can model the performance dynamics with probabilistic distributions. Figure 6b shows the measurements of network performance of m1.medium instances. We verify the network performance with null hypothesis and find it can be modeled with a normal distribution.

We have also verified the performance of other instance types. Figure 7 shows the network performance between two m1.large instances and a m1.medium and a m1.large instance of Amazon EC2. The network performance dynamics of m1.medium instance type is much larger than that of m1.large instance type. Users can achieve better cloud performance by purchasing better types of instances.

On Amazon EC2, the sequential I/O performance of the four evaluated instance types follow Gamma distribution and the random I/O performance of the instance types follow Normal distribution. Table 2 presents the parameters of the distributions, which show that the I/O 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.

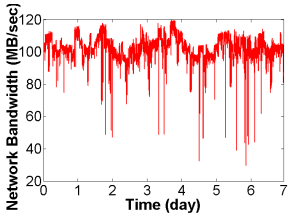
6.3 Evaluations on Use Cases

6.3.1 Workflow Scheduling Problem

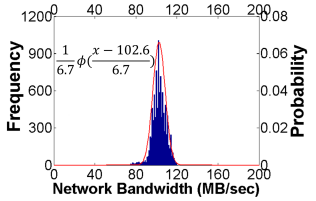
We present the average monetary cost and execution time under different probabilistic deadline requirements in Figure 8. Note, the

Table 2: Parameters of I/O performance distributions on Amazon EC2.

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	$k = 408.1, \theta = 0.26$	$\mu = 1034.0, \sigma = 146.4$



(a) Network performance variance



(b) The histogram and probabilistic distribution

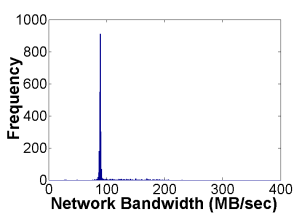
Figure 6: Network performance dynamics of m1.medium instances in Amazon EC2 US East region.

optimization goal is to minimize the monetary cost. Deco obtains better cost optimization results than Autoscaling [25] under all cases, with the monetary cost reduction by 30%–50%. We have several major observations.

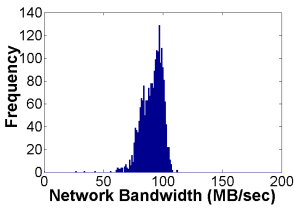
First, Deco reduces more monetary cost than Autoscaling on Montage workflows with larger sizes. The heuristics adopted in Autoscaling, such as deadline assignment, are not able to find the best instance configurations for tasks. On the other hand, our GPU-accelerated search engine can explore and exploit a large area of the search space in a reasonable time and thus is able to find better configurations. Second, when the probabilistic deadline requirement gets loose, Deco is able to save more cost compared to Autoscaling. This is because Autoscaling is static and thus cannot adjust the instance configurations accordingly. Although the average execution time optimized by Deco is larger than Autoscaling, all results can guarantee (be equal to or larger than) the probabilistic deadline requirement.

The optimization time of Deco is ignorable compared to the workflow execution time. Also, this use case is more for offline process. For comparison, we implement the CPU-based parallel algorithm in OpenMP, with a similar algorithm to the GPU algorithm. Nevertheless, Deco's GPU-based searches achieves 12X, 10X and 20X speed-up over CPU-based searches on six cores for the Montage-1, Montage-4 and Montage-8 degree workflows respectively. This demonstrates the efficiency of GPU accelerations.

Figure 11 shows the monetary cost and average execution time of Montage-8 workflow under different deadline settings. Deco obtains smaller monetary cost than Autoscaling under all settings. As the deadline gets loose, the monetary cost decreases and the average execution time increases. This is because more cheap instances are selected for tasks when the deadline gets loose.



(a) m1.large instances



(b) m1.large instance to m1.medium instance

Figure 7: Network performance histograms of different instance types of Amazon EC2.

6.3.2 *Workflow Ensemble*

Overall, for all ensemble settings, Deco obtains better than or the same scores as SPSS [24]. Figure 9 shows the scores with different ensembles of Ligo, with the deadline D3 and budget Bgt1 to Bgt5. Under budget Bgt1 and Bgt5, SPSS and Deco always obtain the same scores because they can only make the same decision, that is, to execute one smallest workflow and all the workflows in the ensemble, respectively. Under budget Bgt2, Bgt3 and Bgt4, Deco obtains better scores than SPSS. This is because the workflow transformation operations in Deco can highly reduce the monetary cost of a workflow, and allow more workflows to be executed within the budget and deadline constraints. The average monetary cost of a workflow optimized by the SPSS algorithm is 1.4 times as high as that optimized by Deco.

When the probabilistic deadline requirement increases from 90% to 99.9%, Deco always obtains higher score than SPSS. The number of completed workflows in Deco is up to 60% higher than that in SPSS.

To evaluate the influence of memory access, we compared the performance of GPU-based search engine with different sizes of workflows. The average speedup of Deco over the CPU-based counterpart on the six cores is 36X, 22X, and 18X for 20-task, 100-task and 1000-task workflow ensembles respectively. As a result, the optimization overhead of Deco takes 4.3-63.17 ms per task for a workflow with 20-1000 tasks.

6.3.3 *Follow-the-Cost*

Figure 10a shows the normalized monetary cost obtained by Deco under different workflow sizes. Deco obtains the lowest monetary cost under all workflow sizes. As the workflow size increases, the cost reduced by Deco compared to the heuristic increases. The cost reduction by Deco mainly comes from two aspects. First, Deco dynamically changes the selected data center for the workflows to exploit the price difference between the two Amazon EC2 regions. The workflows deployed in the Singapore region are eventually migrated to the US East region due to the relatively low price in this region. Second, the cloud performance is dynamic and re-optimization at runtime generates more accurate configuration and migration solutions than offline optimizations. For example, when a task finishes earlier than its scheduled time, Deco chooses more cost effective and usually cheaper instance types for its child tasks after the re-optimization. Another case is, when the network performance between two clouds degrades at runtime, Deco changes the migration decision after the re-optimization. This is because the migration may increase the execution time of tasks and also increase the overall monetary cost.

Figure 10b shows the monetary cost results with different threshold settings. Deco obtains a smaller total cost under all threshold values. As the threshold decreases, the cost optimization result of the heuristic algorithm decreases. This is because the heuristic method performs more re-optimization operations under a smaller threshold. However, the optimization takes a long time, which cannot catch up with the workflow executions. In contrast, with the GPU power, Deco is still able to reduce the cost when the threshold is smaller than 10%.

7. RELATED WORK

There are a lot of works related to our study, and we focus on the most relevant ones on workflow optimizations on IaaS clouds, generalized optimization engines and GPU-based search strategies.

Workflow optimizations. Workflow management systems (WMSes) are a fruitful research area in the cluster and grid environments [13, 23, 42, 9, 2]. Compared with grid and cluster environ-

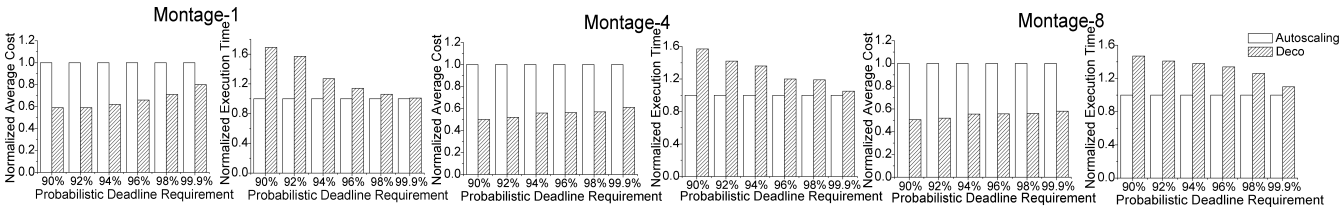


Figure 8: The average monetary cost and execution time of compared algorithms on Montage workflows.

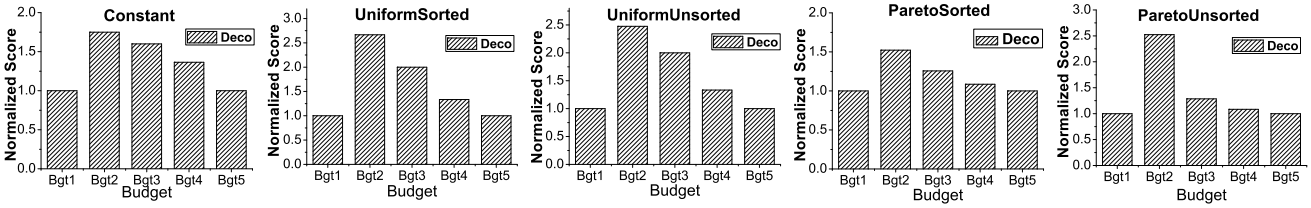
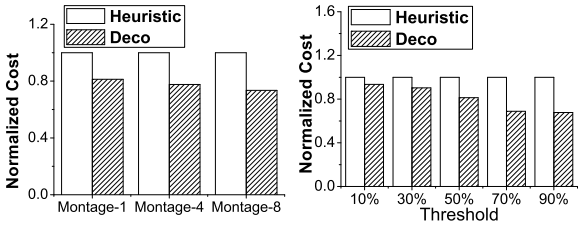
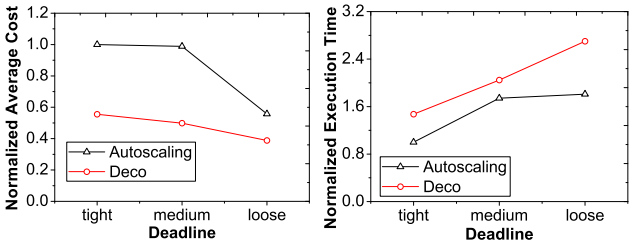


Figure 9: Scores of SPSS and Deco with different ensemble types under budget Bgt1 to Bgt5. Workflow type is Ligo and the deadline constraint is D3.



(a) Study on workflow size (b) Study on threshold

Figure 10: Monetary cost of Deco and Heuristic of follow-the-cost: (a) workflow sizes; (b) performance change threshold.



(a) Normalized average monetary cost (b) Normalized average execution time

Figure 11: Sensitivity study on the deadline parameter using Montage-8 workflow.

ments, IaaS clouds have their own unique features in many system and economic aspects [15]. Many WMSes (such as Pegasus [13] and Kepler [23]), originally designed for grid and local clusters, are redesigned to cloud environments [18, 38].

There have been a lot of studies working on various optimization problems for scientific workflows in the cloud (e.g., [25, 26, 24, 14, 6, 45]). However, most of them target at specific goals or constraints for performance and monetary cost optimizations for workflow. In contrast, we propose a declarative and flexible workflow optimization engine which can be applied to a wider range of workflow optimization problems. Also, we demonstrate that GPU accelerations are viable for workflow optimizations, whereas none of above-mentioned related work has attempted GPU accelerations.

Previous works have demonstrated significant variances on the cloud performance [33, 16]. Machine learning techniques are used to adapt various applications in the changing execution environment [10]. In this paper, our declarative engine exposes the probabilistic performance distributions to users and abstracts the dynamic performance details from users with a declarative interface.

Generalized optimization engines. There have been some general optimization frameworks proposed to solve domain-specific optimization problems in the cloud. Alvaro et al. [3] propose to use a declarative language called Overlog to reduce the complexity of distributed programming in the cloud. Cologne [21] is another declarative system to solve a wide range of constrained optimization problems in distributed systems. They model the optimization problems in the form of Datalog with extensions for constraints and goals. Inspired by their design, this paper has the similar spirit in extending ProLog to hide the internal system details from users. Differently, this paper further makes a series of extensions that go beyond the previous studies [3, 21]. The significant ones include 1) formulate probabilistic extensions of cloud dynamics to ProbLog, 2) workflow specific constructs for reducing the complexity of developing workflow applications, and 3) a GPU-accelerated parallel solver for workflow optimizations.

There are some other studies on resource provisioning. Zhang et al. [44] proposed to formulate the mapping of users resource requirement to cloud resources with SQL queries. ClouDiA provides instance deployment solutions for users [49]. Rai et al. [32] proposed a novel bin-ball abstraction for the resource allocation problems. Different from bin-ball abstractions, workflows have more complicated structures. Moreover, bin-ball abstractions are static, which cannot capture cloud performance dynamics.

GPU-based search strategies. There have been a number of previous studies [5, 37, 32] that use the GPU to accelerate the performance of local search algorithms by means of exploring multiple neighbors in parallel. Different from those studies, our GPU-based implementation are specially designed for general workflow transformation-based optimizations.

8. CONCLUSION

In this paper, we have developed a declarative optimization engine *Deco* for WMSes to address the resource provisioning problem of scientific workflows in IaaS clouds. Deco works as an effective alternative to the existing schedulers of WMSes. With Deco, users can implement their workflow optimizations in a declarative manner, without involving the tedious and complicated details on dynamic cloud features and workflow processing. The declarative engine provides a series of practical constructs as building blocks to facilitate users to develop resource provision-

ing mechanisms for workflows. Moreover, a novel approach of probabilistic optimizations on goals and constraints is developed to address cloud dynamics. Deco also takes advantage of the available GPU power to accelerate the probabilistic evaluations. We integrate Deco into Pegasus (a popular workflow management system) and evaluate Deco with real scientific workflows on both Amazon EC2 and simulator, in comparison with the state-of-the-art heuristics based approaches. Our experiments show that 1) Deco can achieve more effective performance/cost optimizations than the state-of-the-art approaches, with the monetary cost reduction by 30-50%. 2) The optimization overhead of Deco takes 4.3-63.17 ms per task for a workflow with 20-1000 tasks. We have made Deco open-sourced at http://goo.gl/jZatcF.

9. ACKNOWLEDGEMENT

The authors would like to thank anonymous reviewers for their valuable comments. We 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).

10. REFERENCES

[1] L. Abeni and G. Buttazzo. Qos guarantee using probabilistic deadlines. In *ECRTS 1999*, pages 242 – 249.

[2] S. Abrishami, M. Naghibzadeh, and D. H. J. Epema. Deadline-constrained workflow scheduling algorithms for infrastructure as a service clouds. *FGCS*, pages 158–169, 2013.

[3] P. Alvaro, T. Condie, N. Conway, K. Elmeleegy, J. M. Hellerstein, and R. Sears. Boom Analytics: Exploring Data-centric, Declarative Programming for the Cloud. In *EuroSys '10*, pages 223–236.

[4] Amazon Case Studies. `http://aws.amazon.com/solutions/case-studies/`. accessed on July 2014.

[5] A. Arbelaez and P. Codognet. A GPU Implementation of Parallel Constraint-based Local Search. In *PDP '14*, pages 648 – 655.

[6] E.-K. Byun, Y.-S. Kee, J.-S. Kim, and S. Maeng. Cost Optimized Provisioning of Elastic Resources for Application Workflows . *FGCS*, pages 158–169, 2011.

[7] R. N. Calheiros and R. Buyya. Meeting Deadlines of Scientific Workflows in Public Clouds with Tasks Replication. *IEEE TPDS*, pages 1787 – 1796, 2013.

[8] R. N. Calheiros, R. Ranjan, A. Beloglazov, C. A. De Rose, and R. Buyya. CloudSim: A Toolkit for Modeling and Simulation of Cloud Computing Environments and Evaluation of Resource Provisioning Algorithms. *Softw. Pract. Exper.*, pages 23–50, 2011.

[9] J. Cao, S. Jarvis, S. Saini, and G. R. Nudd. GridFlow: Workflow Management for Grid Computing. In *CCGrid '03*, pages 198 – 205.

[10] Z. Chen, S. W. Son, W. Hendrix, A. Agrawal, W. keng Liao, and A. Choudhary. Numarck: Machine learning algorithm for resiliency and checkpointing. In *SC '14*, pages 733–744.

[11] DAGMan. `http://research.cs.wisc.edu/htcondor/dagman/dagman.html`. accessed on May 2014.

[12] L. De Raedt, A. Kimmig, and H. Toivonen. ProbLog: a Probabilistic Prolog and its Application in Link Discovery. In *IJCAI '07*, pages 2468–2473.

[13] E. Deelman, G. Singh, M.-H. Su, J. Blythe, Y. Gil, C. Kesselman, G. Mehta, K. Vahi, G. B. Berriman, J. Good, et al. Pegasus: A Framework for Mapping Complex Scientific Workflows Onto Distributed Systems. *Sci. Program.*, pages 219–237, 2005.

[14] K. Deng, J. Song, K. Ren, and A. Iosup. Exploring Portfolio Scheduling for Long-term Execution of Scientific Workloads in IaaS Clouds. In *SC '13*, pages 55:1–55:12.

[15] I. Foster, Y. Zhao, I. Raicu, and S. Lu. Cloud Computing and Grid Computing 360-Degree Compared. In *GCE '08*, pages 1 – 10.

[16] A. Iosup, S. Ostermann, M. N. Yigitbasi, R. Prodan, T. Fahringer, and D. H. Epema. Performance Analysis of Cloud Computing Services for Many-Tasks Scientific Computing. *IEEE TPDS*, pages 931 – 945, 2011.

[17] Iperf. `http://iperf.sourceforge.net`. accessed on July 2014.

[18] G. Juve, A. Chervenak, E. Deelman, S. Bharathi, G. Mehta, and K. Vahi. Characterizing and Profiling Scientific Workflows. *FGCS*, pages 1–10, 2013.

[19] K. Keahey, R. Figueiredo, J. Fortes, T. Freeman, and M. Tsugawa. Science Clouds: Early Experiences in Cloud Computing for Scientific Applications. In *CCA '08*.

[20] A. Kimmig, B. Demoen, L. De Raedt, V. S. Costa, and R. Rocha. On the Implementation of the Probabilistic Logic Programming Language Problog. *Theory Pract. Log. Program.*, pages 235–262, 2011.

[21] C. Liu, B. T. Loo, and Y. Mao. Declarative Automated Cloud Resource Orchestration. In *SoCC '11*, pages 26:1–26:8.

[22] J. W. Lloyd. *Foundations of Logic Programming*. 1984.

[23] B. Ludäscher, I. Altintas, C. Berkley, D. Higgins, E. Jaeger, M. Jones, E. A. Lee, J. Tao, and Y. Zhao. Scientific Workflow Management and the Kepler System: Research Articles. *Concurr. Comput. : Pract. Exper.*, pages 1039–1065, 2006.

[24] M. Malawski, G. Juve, E. Deelman, and J. Nabrzyski. Cost- and Deadline-constrained Provisioning for Scientific Workflow Ensembles in IaaS Clouds. In *SC '12*, pages 22:1–22:11.

[25] M. Mao and M. Humphrey. Auto-scaling to Minimize Cost and Meet Application Deadlines in Cloud Workflows. In *SC '11*, pages 49:1–49:12.

[26] M. Mao and M. Humphrey. Scaling and Scheduling to Maximize Application Performance within Budget Constraints in Cloud Workflows. In *IPDPS '13*, pages 67 – 78.

[27] Montage Archive. `http://hachi.ipac.caltech.edu:8080/montage/`. accessed on July 2014.

[28] Montage Workflow. `http://montage.ipac.caltech.edu/docs/download2.html`. accessed on July 2014.

[29] D. Nurmi, R. Wolski, C. Grzegorczyk, G. Obertelli, S. Soman, L. Youseff, and D. Zagorodnov. The Eucalyptus Open-Source Cloud-Computing System. In *CCGRID '09*, pages 124–131.

[30] R. A. O'Keefe. *The Craft of Prolog*. 1990.

[31] I. Pietri, G. Juve, E. Deelman, and R. Sakellariou. A Performance Model to Estimate Execution Time of Scientific Workflows on the Cloud. In *WORKS '14*, pages 11–19.

[32] A. Rai, R. Bhagwan, and S. Guha. Generalized Resource Allocation for the Cloud. In *SoCC '12*, pages 15:1–15:12.

[33] J. Schad, J. Dittrich, and J.-A. Quiané-Ruiz. Runtime Measurements in the Cloud: Observing, Analyzing, and Reducing Variance. *Proc. VLDB Endow.*, pages 460–471, 2010.

[34] P. Sevon, L. Eronen, P. Hintsanen, K. Kulovesi, and H. Toivonen. Link Discovery in Graphs Derived from Biological Databases. In *DILS '06*, pages 35–49.

[35] W. Tang, J. Wilkening, N. Desai, W. Gerlach, A. Wilke, and F. Meyer. A Scalable Data Analysis Platform for Metagenomics. In *BigData '13*, pages 21–26.

[36] J. Van der Merwe, K. Ramakrishnan, M. Fairchild, A. Flavel, J. Houle, H. A. Lagar-Cavilla, and J. Mulligan. Towards a Ubiquitous Cloud Computing Infrastructure. In *LANMAN '10*, pages 1–6.

[37] T. Van Luong, N. Melab, and E.-G. Talbi. GPU Computing for Parallel Local Search Metaheuristic Algorithms. *IEEE TC*, pages 173 – 185, 2013.

[38] J. Wang and I. Altintas. Early Cloud Experiences with the Kepler Scientific Workflow System. In *ICCS '12*, pages 1630–1634.

[39] L. Wang, J. Tao, M. Kunze, D. Rattu, and A. C. Castellanos. The Cumulus Project: Build a Scientific Cloud for a Data Center. In *CCA '08*, 2008.

[40] Windows Azure Case Studies. `http://azure.microsoft.com/en-us/case-studies/`. acessed on July 2014.

[41] Workflow Generator. `https://confluence.pegasus.isi.edu/display/pegasus/WorkflowGenerator`. accessed on July 2014.

[42] J. Yu and R. Buyya. A Taxonomy of Scientific Workflow Systems for Grid Computing. *SIGMOD Rec.*, pages 44–49, 2005.

[43] J. Yu, R. Buyya, and C. K. Tham. Cost-based Scheduling of Scientific Workflow Applications on Utility Grids. In *e-Science '05*, pages 140–147.

[44] M. Zhang, R. Ranjan, S. Nepal, M. Menzel, and A. Haller. A Declarative Recommender System for Cloud Infrastructure Services Selection. In *GECON'12*, pages 102–113.

[45] A. Zhou and B. He. Simplified resource provisioning for workflows in iaas clouds. In *CloudCom'14*, pages 650–655, 2014.

[46] A. C. Zhou and B. He. Transformation-based Monetary Cost Optimizations for Workflows in the Cloud. *IEEE TCC*, pages 85–98, 2013.

[47] A. C. Zhou and B. He. A Declarative Optimization Engine for Resource Provisioning of Scientific Workflows in IaaS Clouds. Technical Report 2015-TR-Deco, http://pdcc.ntu.edu.sg/xtra/tr/2015-TR-Deco.pdf, 2015.

[48] A. C. Zhou, B. He, and C. Liu. Monetary Cost Optimizations for Hosting Workflow-as-a-Service in IaaS Clouds. *IEEE TCC*, page PrePrints, 2015.

[49] T. Zou, R. Le Bras, M. V. Salles, A. Demers, and J. Gehrke. ClouDiA: a Deployment Advisor for Public Clouds. In *PVLDB'13*, pages 121–132.