ClouDiA: A Deployment Advisor for Public Clouds

Tao Zou, Ronan Le Bras, Marcos Vaz Salles[†], Alan Demers, Johannes Gehrke

Cornell University
Ithaca, NY

[†]University of Copenhagen
Copenhagen, Denmark

{taozou, lebras, ademers, johannes}@cs.cornell.edu, [†]vmarcos@diku.dk

ABSTRACT

An increasing number of distributed data-driven applications are moving into shared public clouds. By sharing resources and operating at scale, public clouds promise higher utilization and lower costs than private clusters. To achieve high utilization, however, cloud providers inevitably allocate virtual machine instances non-contiguously, i.e., instances of a given application may end up in physically distant machines in the cloud. This allocation strategy can lead to large differences in average latency between instances. For a large class of applications, this difference can result in significant performance degradation, unless care is taken in how application components are mapped to instances.

In this paper, we propose ClouDiA, a general deployment advisor that selects application node deployments minimizing either (i) the largest latency between application nodes, or (ii) the longest critical path among all application nodes. ClouDiA employs mixed-integer programming and constraint programming techniques to efficiently search the space of possible mappings of application nodes to instances. Through experiments with synthetic and real applications in Amazon EC2, we show that our techniques yield a 15% to 55% reduction in time-to-solution or service response time, without any need for modifying application code.

1. INTRODUCTION

With advances in data center and virtualization technology, more and more distributed data-driven applications, such as High Performance Computing (HPC) applications [23, 33, 50, 69], web services and portals [27, 43, 51, 58], and even search engines [5, 7], are moving into public clouds [3]. Public clouds represent a valuable platform for tenants due to their incremental scalability, agility, and reliability. Nevertheless, the most fundamental advantage of using public clouds is cost-effectiveness. Cloud providers manage their infrastructure at scale and obtain higher utilization by combining resource usages from multiple tenants over time, leading to cost reductions unattainable by dedicated private clusters.

Typical cloud providers adopt a pay-as-you-go pricing model in which tenants can allocate and terminate virtual machine instances at any time and pay only for the machine hours they use [2, 48, 64]. Giving tenants such freedom in allocating and terminating instances, public clouds face new challenges in choosing the placement of instances on physical machines. First, they must solve this problem at scale, and at the same time take into account different tenant needs regarding latency, bandwidth, or reliability. Second, even if a fixed goal such as minimizing latency is given, the placement strategy still needs to take into consideration the possibility of future instance allocations and terminations, not only for the current tenant, but also for other tenants who are sharing the resources.

Proceedings of the VLDB Endowment, Vol. 6, No. 2

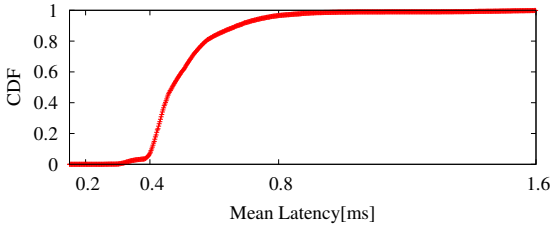

Figure 1: Latency Heterogeneity in EC2

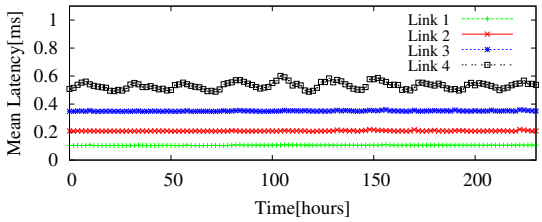

Figure 2: Mean Latency Stability in EC2

Given these difficulties, public cloud service providers do not currently expose instance placement or network topology information to cloud tenants.[1] While these API restrictions ease application deployment, they may cost significantly in performance, especially for latency-sensitive applications. Unlike bandwidth, which can be quantified in the SLA as a single number, network latency depends on message sizes and the communication pattern, both of which vary from one application to another.

In the absence of placement constraints by cloud tenants, cloud providers are free to assign instances to physical resources non-contiguously; i.e., instances allocated to a given application may end up in physically distant machines. This leads to heterogeneous network connectivity between instances: Some pairs of instances are better connected than other pairs in terms of latency, loss rate, or bandwidth. Figure 1 illustrates this effect. We present the CDF

[1]The only exception we know of is the notion of *cluster placement groups* in Amazon EC2 cluster instances. However, these cluster instances are much more costly than other types of instances, and only a limited number of instances can be allocated within a cluster placement group.

of the mean pairwise end-to-end latencies among 100 Amazon EC2 large instances (m1.large), obtained by TCP round-trip times of 1 KB messages. Around 10% of the instance pairs exhibit latency above 0.7 ms, while the bottom 10% are below 0.4 ms. This heterogeneity in network latencies can greatly increase the response time of distributed, latency-sensitive applications. Figure 2 plots the mean latencies of four representative links over a 10-day experiment, with measurements taken every two hours. The observed stability of mean latencies suggests that applications may obtain better performance by selecting "good" links for communication.

This paper examines how developers can carefully tune the deployment of their distributed applications in public clouds. At a high level, we make two important observations: (i) If we carefully choose the mapping from nodes (components) of distributed applications to instances, we can potentially prevent badly interconnected pairs of instances from communicating with each other; (ii) if we over-allocate instances and terminate instances with bad connectivity, we can potentially improve application response times. These two observations motivate our general approach: A cloud tenant has a target number of components to deploy onto x virtual machines in the cloud. In our approach, she allocates x instances plus a small number of additional instances (say $x/10$). She then carefully selects which of these $1.1 \cdot x$ instances to use and how to map her x application components to these selected virtual machines. She then terminates the $x/10$ over-allocated instances.

Our general approach could also be directly adopted by a cloud provider – potentially at a price differential – but the provider would need to widen its API to include latency-sensitivity information. Since no cloud provider currently allows this, we take the point of view of the cloud tenant, by whom our techniques are immediately deployable.

Contributions of this paper. In this paper, we introduce the problem of deployment advice in the cloud and instantiate a concrete deployment advisor called ClouDiA (*Clou*d *D*eployment *A*dvisor).

1. ClouDiA works for two large classes of data-driven applications. The first class, which contains many HPC applications, is sensitive to the worst link latency, as this latency can significantly affect total time-to-solution in a variety of scientific applications [1, 11, 21, 34]. The second class, represented by search engines as well as web services and portals, is sensitive to the longest path between application nodes, as this cost models the network links with the highest potential impact on application response time. ClouDiA takes as input an application communication graph and an optimization objective, automatically allocates instances, measures latencies, and finally outputs an optimized node deployment plan (Section 2). To the best of our knowledge, our methodology is the first to address deployment tuning for latency-sensitive applications in public clouds.

2. We formally define the two node deployment problems solved by ClouDiA and prove the hardness of these problems. The two optimization objectives used by ClouDiA – largest latency and longest critical path – model a large class of current distributed cloud applications that are sensitive to latency (Section 3).

3. We present an optimization framework that can solve these two classes of problems, each by different solvers based on mixed-integer and constraint programming. We discuss optimizations and heuristics that allow us to obtain high-quality deployment plans over the scale of hundreds of instances (Section 4).

4. We discuss methods to obtain accurate latency measurements (Section 5) and evaluate our optimization framework with both synthetic and real distributed applications in Amazon EC2. We observe 15%-55% reduction in time-to-solution or response times. These benefits come exclusively from optimized deployment plans, and require no changes to the specific application (Section 6).

We discuss related work in Section 7 and then conclude.

2. TUNING FOR LATENCY-SENSITIVE APPLICATIONS IN THE CLOUD

To give a high-level intuition for our approach, we first describe the classes of applications we target in Section 2.1. We then describe the architecture that ClouDiA uses to suggest deployments for these applications in Section 2.2.

2.1 Latency-Sensitive Applications

We can classify latency-sensitive applications in the cloud into two broad classes: high-performance computing applications, for which the main performance goal is *time-to-solution*, and service-oriented applications, for which the main performance goal is *response time for service calls*.

Goal: Time-to-solution. A number of HPC applications simulate natural processes via long-running, distributed computations. For example, consider the simulation of collective animal movement published by Couzin et al. in Nature [19]. In this simulation, a group of animals, such as a fish school, moves together in a two-dimensional space. Animals maintain group cohesion by observing each other. In addition, a few animals try to influence the direction of movement of the whole group, e.g., because they have seen a predator or a food source. This simulation can be partitioned among multiple compute nodes through a spatial partitioning scheme [60]. At every time step of the simulation, neighboring nodes exchange messages before proceeding to the next time step. As the end of a time step is a logical barrier, worst-link latency essentially determines communication cost [1, 11, 34, 69]. Similar communication patterns are common in multiple linear algebra computations [21]. Another example of an HPC application where time-to-solution is critical is dynamic traffic assignment [62]. Here traffic patterns are extrapolated for a given time period, say 15 min, based on traffic data collected for the previous period. Simulation must be faster than real time so that simulation results can generate decisions that will improve traffic conditions for the next time period. Again, the simulation is distributed over multiple nodes, and computation is assigned based on a graph partitioning of the traffic network [62]. In all of these HPC applications, time-to-solution is dramatically affected by the latency of the worst link.

Goal: Service response time. Web services and portals, as well as search engines, are prime cloud applications [5, 27, 58]. For example, consider a web portal, such as Yahoo! [51] or Amazon [58]. The rendering of a web page in these portals is the result of tens or hundreds, of web service calls [43]. While different portions of the web page can be constructed independently, there is still a critical path of service calls that determines the server-side communication time to respond to a client request. Latencies in the critical path add up, and can negatively affect end-user response time.

2.2 Architecture of ClouDiA

Figure 3 depicts the architecture of ClouDiA. The dashed line indicates the boundary between ClouDiA and public cloud tenants. The tuning methodology followed by ClouDiA comprises the following steps:

1. Allocate Instances: A tenant specifies the communication graph for the application, along with a maximum number of instances at least as great as the required number of application nodes. CloudDiA then automatically allocates cloud instances to

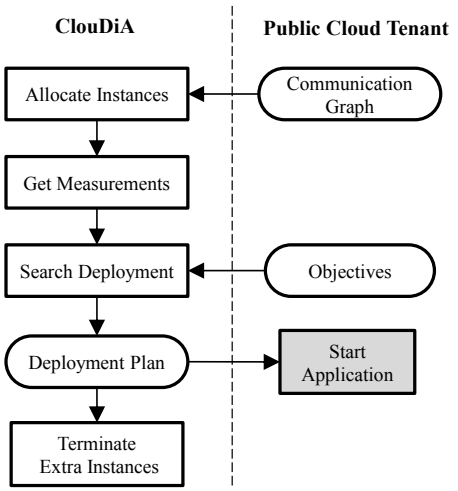

Figure 3: Architecture of ClouDiA

run the application. Depending on the specified maximum number of instances, ClouDiA will over-allocate instances to increase the chances of finding a good deployment.

2. Get Measurements: The pairwise latencies between instances can only be observed after instances are allocated. ClouDiA performs efficient network measurements to obtain these latencies, as described in Section 5. The main challenge reliably estimating the mean latencies quickly, given that time spent in measurement is not available to the application.

3. Search Deployment: Using the measurement results, together with the optimization objective specified by the tenant, CloudDiA searches for a "good" deployment plan: one which avoids "bad" communication links. We formalize this notion and pose the node deployment problem in Section 3. We then formulate two variants of the problem that model our two classes of latency-sensitive applications. We prove the hardness of these optimization problems in Appendix A. Given the hardness of these problems, traditional methods cannot scale to realistic sizes. We propose techniques that significantly speed up the search in Section 4.

4. Terminate Extra Instances: Finally, ClouDiA terminates any over-allocated instances and the tenant can start the application with an optimized node deployment plan.

Adapting to changing network conditions. The architecture outlined above assumes that the application will run under relatively stable network conditions. We believe this assumption is justified: The target applications outlined in Section 2.1 have long execution time once deployed, and our experiments in Figure 2 show stable pairwise latencies in EC2. In the future, more dynamic cloud network infrastructure may become the norm. In this case, the optimal deployment plan could change over time, forcing us to consider dynamic re-deployment. We envision that re-deployment can be achieved via iterations of the architecture above: getting new measurements, searching for a new optimal plan, and re-deploying the application.

Two interesting issues arise with iterative re-deployment. First, we need to consider whether information from previous runs could be exploited by a new deployment. Unfortunately, previous runs provide no information about network resources that were *not* used by the application. In addition, as re-deployment is triggered by changes in network conditions, it is unlikely that network conditions of previous runs will be predictive of conditions of future runs. Second, re-deployment should not interrupt the running application, especially in the case of web services and portals. Unfortunately, current public clouds do not support VM live migration [2, 48, 64]. Without live migration, complicated state migration logic would have to be added to individual cloud applications.

3. THE NODE DEPLOYMENT PROBLEM

In this section, we present the optimization problems addressed by ClouDiA. We begin by discussing how we model network cost (Section 3.1). We then formalize two versions of the node deployment problem (Section 3.2).

3.1 Cost Functions

When a set of instances is allocated in a public cloud, in principle any instance in the set can communicate with any other instance, possibly at differing costs. Differences arise because of the underlying physical network resources that implement data transfers between instances. We call the collection of network resources that interconnect a pair of instances the *communication link* between them. We formalize communication cost as follows.

DEFINITION 1 (COMMUNICATION COST). *Given a set of instances S, we define $C_L : S \times S \to \mathbb{R}$ as the communication cost function. For a pair of instances $i, j \in S$, $C_L(i, j)$ gives the communication cost of the link from instance i to j.*

C_L can be defined based on different criteria, e.g., latency, bandwidth, or loss rate. To reflect true network properties, we do not assume C_L to be a metric: Costs of links can be asymmetric and the triangle inequality does not necessarily hold. In this paper, given the applications we are targeting, we focus solely on using *network latency* as a specific instance for C_L. Extending to other network cost measurements is future work.

Our definition of communication cost treats communication links as essentially independent. This modeling decision ignores the underlying implementation of communication links in the datacenter. In practice, however, current clouds tend to organize their network topology in a tree-like structure [10]. A natural question is whether we could provide more structure to the communication cost function by reverse engineering this underlying network topology. Approaches such as Sequoia [52] deduce underlying network topology by mapping application nodes onto leaves of a virtual tree. Unfortunately, even though these inference approaches work well for Internet-level topologies, state-of-the-art methods cannot infer public cloud environments accurately [9, 15].

Even if we could obtain accurate topology information, it would be non-trivial to make use of it. First, there is no guarantee that the nodes of a given application can all be allocated to nearby physical elements (e.g., in the same rack), so we need an approach that fundamentally tackles differences in communication costs. Second, datacenter topologies are themselves evolving, with recent proposals for new high-performance topologies [42]. Optimizations developed for a specific tree-like topology may no longer be generally applicable as new topologies are deployed. Our general formulation of communication cost makes our techniques applicable to multiple different choices of datacenter topologies. Nevertheless, as we will see in Section 6, we can support a general cost formulation, but optimize deployment search when there are uniformities in the cost, e.g., clusters of links with similar cost values.

To estimate the communication cost function C_L for the set of allocated instances, CloudDiA runs an efficient network measurement tool (Section 5). Note that the exact communication cost is application-dependent. Applications communicating messages of different sizes can be affected differently by the network heterogeneity. However, for *latency-sensitive applications*, we expect application-independent network latency measurements can

be used as a good performance indicator, although they might not precisely match the exact communication cost in application execution. This is because our notion of cost need only discriminate between "good" and "bad" communication links, rather than accurately predict actual application runtime performance.

3.2 Problem Formulation

To deploy applications in public clouds, a mapping between logical application nodes and cloud instances needs to be determined. We call this mapping a *deployment plan*.

DEFINITION 2 (DEPLOYMENT PLAN). *Let S be the set of instances. Given a set N of application nodes, a deployment plan $\mathcal{D} : N \to S$ is a mapping of each application node $n \in N$ to an instance $s \in S$.*

In this paper, we require that $\mathcal{D}$ be an injection, that is, each instance $s \in S$ can have at most one application node $n \in N$ mapped to it. There are two implications of this definition. On one hand, we do not collocate application nodes on the same instance. In some cases, it might be beneficial to collocate services, yet we argue these services should be merged into application nodes before determining the deployment plan. On the other hand, it is possible that some instances will have no application nodes mapped to them. This gives us flexibility to over-allocate instances at first, and then shutdown those instances with high communication cost.

In today's public clouds, tenants typically determine a deployment plan by either a default or a random mapping. CloudDiA takes a more sophisticated approach. The deployment plan is generated by solving an optimization problem and searching through a space of possible deployment plans. This search procedure takes as input the communication cost function $\mathcal{C}_\mathcal{L}$, obtained by our measurement tool, as well as a *communication graph* and a *deployment cost function*, both specified by the cloud tenant.

DEFINITION 3 (COMMUNICATION GRAPH). *Given a set N of application nodes, the communication graph $G = (V,E)$ is a directed graph where $V = N$ and $E = \{(i, j) | i, j \in N \wedge \mathsf{talks}(i, j)\}$.*

The talks relation above models the application-specific communication patterns. When defining the communication graph through the talks relation, the cloud tenant should only include communication links that have impact on the performance of the application. For example, those links first used for bootstrapping and rarely used afterwards should not be included in the communication graph. An alternative formulation for the communication graph would be to add weights to edges, extending the semantics of talks. We leave this to future work.

We find that although such communication graph is typically not hard to extract from the application, it might be a tedious task for a cloud tenant to generate an input file with $O(|N|^2)$ links. ClouDiA therefore provides *communication graph templates* for certain common graph structures such as meshes or bipartite graphs to minimize human involvement.

In addition to the communication graph, a deployment cost function needs to be specified by the cloud tenant. At a high level, a deployment cost function evaluates the cost of the deployment plan by observing the structure of the communication graph and the communication cost for links in the given deployment. The optimization goal for CloudDiA is to generate a deployment plan that minimizes such cost. The formal definition of deployment cost is as follows:

DEFINITION 4 (DEPLOYMENT COST). *Given a deployment plan $\mathcal{D}$, a communication graph G, and a communication cost function $\mathcal{C}_\mathcal{L}$, we define $\mathcal{C}_\mathcal{D}(\mathcal{D}, G, \mathcal{C}_\mathcal{L}) \in \mathbb{R}$ as the deployment cost of $\mathcal{D}$. $\mathcal{C}_\mathcal{D}$ must be monotonic on link cost and invariant under exchanging nodes that are indistinguishable using link costs.*

Now, we can formally define the node deployment problem:

DEFINITION 5 (NODE DEPLOYMENT PROBLEM). *Given a deployment cost function $\mathcal{C}_\mathcal{D}$, a communication graph G, and a communication cost function $\mathcal{C}_\mathcal{L}$, the node deployment problem is to find the optimal deployment $\mathcal{D}_{OPT} = \arg\min_\mathcal{D} \mathcal{C}_\mathcal{D}(\mathcal{D}, G, \mathcal{C}_\mathcal{L})$.*

In the remainder, we focus on two classes of deployment cost functions, which capture the essential aspects of the communication cost of latency-sensitive applications running in public clouds.

In High Performance Computing (HPC) applications, such as simulations, matrix computations, and graph processing [69], application nodes typically synchronize periodically using either global or local communication barriers. The completion of these barriers depends on the communication link which experiences the longest delay. Motivated by such applications, we define our first class of deployment cost function to return the highest link cost.

CLASS 1 (DEPLOYMENT COST: LONGEST LINK). *Given a deployment plan $\mathcal{D}$, a communication graph $G = (V,E)$ and a communication cost function $\mathcal{C}_\mathcal{L}$, the longest link deployment cost $\mathcal{C}^{LL}_\mathcal{D}(\mathcal{D}, G, \mathcal{C}_\mathcal{L}) = \max_{(i,j)\in E} \mathcal{C}_\mathcal{L}(\mathcal{D}(i), \mathcal{D}(j))$.*

Another class of latency-sensitive applications is exemplified by search engines [5, 7] as well as web services and portals [27, 43, 51, 58]. Services provided by these applications typically organize application nodes into trees or directed acyclic graphs, and the overall latency of these services is determined by the communication *path* which takes the longest time [56]. We therefore define our second class of deployment cost function to return the highest path cost.

CLASS 2 (DEPLOYMENT COST: LONGEST PATH). *Given a deployment plan $\mathcal{D}$, an acyclic communication graph G and a communication cost function $\mathcal{C}_\mathcal{L}$, the longest path deployment cost $\mathcal{C}^{LP}_\mathcal{D}(\mathcal{D}, G, \mathcal{C}_\mathcal{L}) = \max_{\text{path } P\subseteq G} \left(\Sigma_{(i,j)\in P} \mathcal{C}_\mathcal{L}\big(\mathcal{D}(i), \mathcal{D}(j)\big)\right)$.*

Note that the above definition assumes the application is sending a sequence of causally-related messages along the edges of a path, and summation is used to aggregate the communication cost of the links in the path.

Although we believe these two classes cover a wide spectrum of latency-sensitive cloud applications, there are still important applications which do not fall exactly into either of them. For example, consider a key-value store with latency requirements on average response time or the 99.9$^{\text{th}}$ percentile of the response time distribution [20]. This application does not exactly match the two deployment cost functions above, since average response time may be influenced by multiple links in different paths. A natural extension to Longest Link returning k^{th} highest link cost as deployment cost can be useful in this case. While we leave this extension to future work, we discuss the applicability of our deployment cost functions to a key-value store workload further in Section 6.1. We then proceed to show experimentally in Section 6.4 that even though Longest-Link is not a perfect match for such a workload, use of this deployment cost function still yields a 15%-31% improvement in average response time for a key-value store workload. Considering this possibility of utilizing the above cost functions even if there is no exact match, CloudDiA is able to automatically improve response times of an even wider range of applications.

We prove that the node deployment problem with longest-path deployment cost is NP-hard. With longest-link deployment cost, it is also NP-hard and cannot be efficiently approximated unless

P=NP (Appendix A). Given the hardness results, our solution approach consists of Mixed-Integer Programming (MIP) and Constraint Programming (CP) formulations (Section 4). We show experimentally that our approach still brings significant performance improvement to real applications. In addition, from the insight gained from the theorems above, we also show that properly rounding communication costs to *cost clusters* can be heuristically used to further boost solution performance (Section 6).

4. SOLVER TECHNIQUES

In this section, we propose two encodings to solve the Longest Link Node Deployment Problem (LLNDP), as well as one formulation for the Longest Path Node Deployment Problem (LPNDP).

4.1 Mixed-Integer Program for LLNDP

Given a communication graph $G = (V,E)$ and a communication cost function $\mathcal{C}_\mathcal{L}$ defined over any pair of instances in S, the *Longest Link Node Deployment Problem* can be formulated as the following Mixed-Integer Program (MIP):

$$
\begin{aligned}
&(MIP)\ \textit{min}\ c \\
&s.t. \sum_{i \in V} x_{ij} = 1 && \forall j \in S \quad (1)\\
&\sum_{j \in S} x_{ij} = 1 && \forall i \in V \quad (2)\\
&c \geq \mathcal{C}_\mathcal{L}(j,j')(x_{ij} + x_{i'j'} - 1) && \forall (i,i') \in E, \forall j,j' \in S \quad (3)\\
&x_{ij} \in \{0,1\} && \forall i \in V, j \in S\\
&c \geq 0
\end{aligned}
$$

In this encoding, the boolean variable x_{ij} indicates whether the application node $i \in V$ is deployed on instance $j \in S$. The constraints (1) and (2) ensure that the variables x_{ij} represent a one-to-one mapping between the set V and S. Note that the set V might need to be augmented with dummy application nodes, so that we have $|V| = |S|$. Also, the constraints (3) require that the value of c is at least equal to $\mathcal{C}_\mathcal{L}(j,j')$ any time there is a pair (i,i') of communicating application nodes and that i and i' are deployed on j and j', respectively. Finally, the minimization in the objective function will make one of the constraints (3) tight, thus leading to the desired longest link value.

4.2 Constraint Programming for LLNDP

Whereas the previous formulation directly follows from the problem definition, our second approach exploits the relation between this problem and the subgraph isomorphism problem, as well as the clusters of communication cost values. The algorithm proceeds as follows. Given a goal c, we search for a deployment that avoids communication costs greater than c. Such a deployment exists if and only the graph $G_c = (S,E_c)$ where $E_c = \{(i,j) : \mathcal{C}_\mathcal{L}(i,j) \leq c\}$ contains a subgraph isomorphic to the communication graph $G = (V,E)$. Therefore, by finding such a subgraph, we obtain a deployment whose deployment cost c' is such that $c' \leq c$. Assume c'' is the largest communication cost strictly lower than c'. Thus, any improving deployment must have a cost of c'' or lower, and the communication graph $G = (V,E)$ must be isomorphic to a subgraph of $G_{c''} = (S,E_{c''})$ where $E_{c''} = \{(i,j) : \mathcal{C}_\mathcal{L}(i,j) \leq c''\}$. We proceed iteratively until no deployment is found. Note that the number of iterations is bounded by the number of distinct cost values. Therefore, clustering similar values to reduce the number of distinct cost values would improve the computation time by lowering the number of iterations, although it approximates the actual value of the objective function. We investigate the impact of *cost clusters* in Section 6. For a given objective value c, the encoding of the problem might be expressed as the following Constraint Programming (CP) formulation:

$$
\begin{aligned}
&(CP)\quad \texttt{alldifferent}\big((u_i)_{1 \leq i \leq |V|}\big)\\
&(u_i,u_{i'}) \neq (j,j') && \forall (i,i') \in E, \forall (j,j') : \mathcal{C}_\mathcal{L}(j,j') > c\\
&u_i \in \{1,...,|S|\} && \forall 1 \leq i \leq |V|
\end{aligned}
$$

This encoding is substantially more compact that the MIP formulation, as the binary variables are replaced by integer variables, and the mapping is efficiently captured within the `alldifferent` constraint. In addition, at the root of the search tree, we perform an extra filtering of the domains of the x_{ij} variables that is based on compatibility between application nodes and instances. Indeed, as the objective value c decreases, the graph G_c becomes sparse, and some application nodes can no longer be mapped to some of the instance nodes. For example, a node in G needs to be mapped to a node in G_c of equal or higher degree. Similar to [68], we define a labeling based on in- and out-degree, as well as information about the labels of neighboring nodes. This labeling establishes a partial order on the nodes and expresses compatibility between them. For more details on this labeling, please refer to [68].

4.3 Mixed-Integer Programming for LPNDP

As previously mentioned, the Node Deployment Problem is intrinsically related to the Subgraph Isomorphism Problem (SIP). In addition, the Longest Link objective function allows us to directly prune the graph that must contain the communication graph G and therefore, can be encoded as a series of Subgraph Isomorphism Satisfaction Problems. This plays a key role in the success of the CP formulation. On the contrary, the objective function of the *Longest Path Node Deployment Problem* (LPNDP) interferes with the structure of the SIP problem and rules out sub-optimal solutions only when most of the application nodes have been assigned to instances. As a result, this optimization function barely guides the systematic search, and makes it less suitable for a CP approach. Consequently, we only provide a MIP formulation for the LPNDP.

Given a communication graph $G = (V,E)$ and a communication cost function $\mathcal{C}_\mathcal{L}$ defined over any pair of instances in S, the *Longest Path Node Deployment Problem* (LPNDP) can be formulated as the following Mixed-Integer Program (MIP):

$$
\begin{aligned}
&(MIP)\ \textit{min}\ t\\
&s.t. \sum_{i \in V} x_{ij} = 1 && \forall j \in S\\
&\sum_{j \in S} x_{ij} = 1 && \forall i \in V\\
&c_{ii'} \geq \mathcal{C}_\mathcal{L}(j,j')(x_{ij} + x_{i'j'} - 1) && \forall (i,i') \in E, \forall j,j' \in S\\
&t \geq t_i, t_i \geq 0 && \forall i \in V\\
&t_{i'} \geq t_i + c_{ii'} && \forall (i,i') \in E\\
&x_{ij} \in \{0,1\} && \forall i \in V, j \in S\\
&c_{ii'} \geq 0 && \forall (i,i') \in E\\
&t \geq 0
\end{aligned}
$$

As in the previous MIP encoding, the boolean variable x_{ij} indicates whether the application node i is deployed on instance j in a one-to-one mapping. In addition, the variable $c_{ii'}$ captures the communication cost from application node i to node i' that would result from the deployment specified by the x_{ij} variables. The variable t_i represents the longest directed path in the communication graph G

that reaches the application node i. Finally, the variable t appears in the objective function, and corresponds to the maximum among the t_i variables.

5. MEASURING NETWORK DISTANCE

Making wise deployment decisions to optimize performance for latency-sensitive applications requires knowledge of pairwise communication cost. A natural way to characterize the communication cost is to directly measure round-trip latencies for all instance pairs. To ensure such latencies are a good estimate of communication cost during application execution, two aspects need to be handled. First, the size of the messages being exchanged during application execution is usually non-zero. Therefore, rather than measuring pure round-trip latencies with no data content included, we measure TCP round-trip time of small messages, where message size depends on the actual application workload. Second, during the application execution, multiple messages are typically being sent and received at the same time. Such temporal correlation affects network performance, especially end-to-end latency. But the exact interference patterns heavily depend on low-level implementation details of applications, and it is impractical to require such detailed information from the tenants. Instead, we focus on estimating the quality of links without interference, as this already gives us guidance on which links are certain to negatively affect actual executions.

Although we observe mean latency heterogeneity is stable in the cloud (Figure 2), experimental studies have demonstrated that clouds suffer from high latency jitter [55, 59, 69]. Therefore, to estimate mean latency accurately, multiple round-trip latency measurements have to be obtained for each pair of instances. Since the number of instance pairs is quadratic in the number of instances, such measurement takes substantial time. On the one hand, we want to run mean-latency measurements as fast as possible to minimize the overhead of using ClouDiA. On the other hand, we need to avoid introducing uncontrolled measurement artifacts that may affect the quality of our results. We propose three possible approaches for organizing pairwise mean latency measurements in the following.

1. Token Passing. In this first approach, a unique token is passed between instances. When an instance i receives this token, it selects another instance j and sends out a probe message of given size. Once the entire probe message has been received by j, it replies to i with a message of the same size. Upon receiving the entire reply message, i records the round-trip time and passes the token on to another instance chosen at random or using a predefined order. By having such a unique token, we ensure that only one message is being transferred at any given time, including the message for token passing itself. We repeat this token passing process a sufficiently large number of times, so multiple round-trip measurements can be collected for each link. We then aggregate these measurements into mean latencies per link.

This approach achieves the goal of obtaining pairwise mean-latency measurements without correlations between links. However, the lack of parallelism restricts its scalability.

2. Uncoordinated. To improve scalability, we would like to avoid excessive coordination among instances, so that they can execute measurements in parallel. We introduce parallelism by the following simple scheme: Each instance picks a destination at random and sends out a probe message. Meanwhile, all instances monitor incoming messages and send reply messages once an entire probe message has been received. After one such round-trip measurement, each instance picks another probe destination and starts over. The process is repeated until we have collected enough round-trip measurements for every link. We then aggregate these measurements into mean latencies per link.

Given n instances, this approach allows up to n messages to be in flight at any given time. Therefore, this approach provides better scalability than the first approach. However, since probe destinations are chosen at random without coordination, it is possible that: 1) one instance needs to send out reply messages while it is sending out a probe message; or 2) multiple probe messages are sent to the same destination from different sources. Such cross-link correlations are undesirable for our measurement goal.

3. Staged. To prevent cross-link correlations while preserving scalability, coordination is required when choosing probe destinations. We add an extra *coordinator* instance and divide the entire measurement process into *stages*. To start a stage for n instances in parallel, the coordinator first picks $\lfloor \frac{n}{2} \rfloor$ pairs of instances $\{(i_1, j_1), (i_2, j_2), ..., (i_{\lfloor \frac{n}{2} \rfloor}, j_{\lfloor \frac{n}{2} \rfloor})\}$ such that $\forall p, q \in \{1..n\}, i_p \neq j_q$ and $i_p \neq i_q$ if $p \neq q$. The coordinator then notifies each $i_p, p \in \{1, .., \lfloor \frac{n}{2} \rfloor\}$, of its corresponding j_p. After receiving a notification, i_p sends probe messages to j_p and measures round-trip latency as described above. Finally, i_p ends its stage by sending a notification back to the coordinator, and the coordinator waits for all pairs to finish before starting a new stage.

This approach allows up to $\frac{n}{2}$ messages between instances in flight at any time at the cost of having a central coordinator. We minimize the cost of per-stage coordination by consecutively measuring round-trip times between the same given pair of instances K_s times within the same stage, where K_s is a parameter. With this optimization, the staged approach can potentially provide scalability comparable to the *uncoordinated* approach. At the same time, by careful implementation, we can guarantee that each instance is always in one of the following three states: 1) sending to one other instance; 2) receiving from one other instance; or 3) idle in networking. This guarantee provides independence among pairwise link measurements similar to that achieved by *token passing*.

Approximations. Even the Staged network latency benchmark above can take non-negligible time to generate mean latency estimates for a large number of instances. Given that our goal is simply to estimate link costs for our solvers, we have experimented with other simple network metrics, such as hop count and IP distance, as proxies for round-trip latency. Surprisingly, these metrics did *not* turn out to correlate well with round-trip latency. We provide details on these negative results in Appendix B.

6. EXPERIMENTAL RESULTS

In this section, we present experiments demonstrating the effectiveness of ClouDiA. We begin with a description of the several representative workloads used in our experiments. We then present micro-benchmark results for the network measurement tools and the solver of ClouDiA. Finally, we experimentally demonstrate the performance improvements achievable in public clouds by using ClouDiA as the deployment advisor.

6.1 Workloads

To benchmark the performance of ClouDiA, we implement three different workloads: a behavioral simulation workload, a query aggregation workload, and a key-value store workload. Each workload illustrates a different communication pattern.

Behavioral Simulation Workload. In behavioral simulations, collections of individuals interact with each other to form complex systems. Examples of behavioral simulations include large-scale traffic simulations and simulation of groups of animals. These simulations organize computation into ticks and achieve parallelism

by partitioning the simulated space into regions. Each region is allocated to a processor and inter-node communication is organized as a 2D or 3D mesh. As synchronization among processors happens every tick, the progress of the entire simulation is limited by the pair of nodes that take longest to synchronize. Longest-Link is thus a natural fit to the deployment cost of such applications. We implement a workload similar to the fish simulation described by Couzin et al [19]. The communication graph is a 2D mesh and the message size per link is 1 KB for each tick. To focus on network effects, we hide CPU intensive computation and study the time to complete 100K ticks over different deployments.

Synthetic Aggregation Query Workload. In a search engine or distributed text database, queries are processed by individual nodes in parallel and the results are then aggregated [7]. To prevent the aggregation node from becoming bottleneck, a multi-level aggregation tree can be used: each node aggregates some results and forwards the partial aggregate to its parent in the tree for further aggregation. The response time of the query depends on the path from a leaf to the root that has highest total latency. Longest-Path is thus a natural fit for the deployment cost of such applications. We implement a two-level aggregation tree of a top-k query answering workload. The communication graph is a tree and the forwarding message size varies from the leaves to the root, with an average of 4 KB. We hide ranking score computation and study the response time of aggregation query results over different deployments.

Key-Value Store Workload. We also implement a distributed key-value store workload. The key-value store is queried by a set of front-end servers. Keys are randomly partitioned among the storage nodes, and each query touches a random subset of them. The communication graph therefore is a bipartite graph between front-end servers and storage machines. However, unlike the simulation workload, the average response time of a query is not simply governed by the slowest link. To see this, consider a deployment with mostly equal-cost links, but with a single slower link of cost c, and compare this to a similar deployment with two links of cost $c-\varepsilon$. If Longest-Link were used as the deployment cost function, the second deployment would be favored even though the first deployment actually has lower average response time. Indeed, neither Longest-Link nor Longest-Path is the precisely correct objective function for this workload. We evaluate the query response time over different deployments by using Longest-Link, with a hope that it can still help avoid high cost links.

6.2 Network Micro-Benchmarks

Setup. The network measurement tools of ClouDiA are implemented in C++ using TCP sockets (SOCK_STREAM). We set all sockets to non-blocking mode and use *select* to process concurrent flows (if any). We disable the Nagle Algorithm.

We ran experiments in the Amazon Elastic Compute Cloud (Amazon EC2). We used large instances (m1.large) in all experiments. Each large instance has 7.5 GB memory and 4 EC2 Compute Units. Each EC2 Compute Unit provides the equivalent CPU capacity of a 1.0-1.2 GHz 2007 Opteron or 2007 Xeon processor [2]. Unless otherwise stated, we use 100 large instances for network micro-benchmarks, all allocated by a single *ec2-run-instance* command, and set the round-trip message size to 1KB. We show the following results from the same allocation so that they are comparable. Similar results are obtained in other allocations.

Round-Trip Latency Measurement. We run latency measurements with each of the three approaches proposed in Section 5 and compare their accuracy in Figure 4. To make sure *token passing* can observe each link a sufficiently large amount of times, we use 50 large instances in this experiment. We consider the mean latencies for 50^2 instance pairs as a 50^2-dimension vector of mean latencies, of which each dimension represents one link. Results are first normalized to the unit vector. Then, *staged* and *uncoordinated* are compared with the baseline *token passing*. The CDF of the relative error of each dimension is shown in Figure 4. Using *staged*, we find 90% of links have less than 10% relative error and the maximum error is less than 30%; whereas using *uncoordinated* we find 10% of links have more than 50% relative error. Therefore, as expected, *staged* exhibits higher measurement accuracy than *uncoordinated*.

Figure 5 shows convergence over time using the *staged* approach with 100 instances and $K_s = 10$. Again, the measurement result is considered as a latency vector. The result of the full 30 minutes observation is used as the *ground truth*. Each stage on average takes 2.75 ms. Therefore, we obtain about 3004 measurements for each instance pair within 30 minutes. We then calculate the root-mean-square error of partial observations between 1 and 30 minutes compared with the ground truth. From Figure 5, we observe the root-mean-square error drops quickly within the first 5 minutes and smooths out afterwards. Therefore, we pick 5 minutes as the measurement time for all the following experiments with 100 instances. For experiments with $n \neq 100$ instances, since the *staged* approach tests $\frac{n}{2}$ pairs in parallel whereas there are $O(n^2)$ total pairs, measurement time needs to be adjusted linearly to $5 \cdot \frac{n}{100}$ minutes.

6.3 Solver Micro-Benchmarks

Setup. We solve the MIP formulation using the IBM ILOG CPLEX Optimizer, while every iteration of the CP formulation is performed using IBM ILOG CP Optimizer. Solvers are executed on a local machine with 16 GB memory and Intel Core i7-2600 CPU (4 physical cores with hyper-threading). We enable parallel mode to allow both solvers to fully utilize all CPU resources. We use k-means to cluster link costs. Since the link costs are in one dimension, such k-means can be optimally solved in $O(kN)$ time using dynamic programming, where N is the number of distinct values for clustering and k is the number of *cost clusters*. After running k-means clustering, all costs are modified to the mean of the containing cluster and then passed to the solver. For comparison purposes, the same set of network latency measurements as in Section 6.2 is used for all solver micro-benchmarks. In each solver micro-benchmark, we set the number of application nodes to be 90% of the number of allocated instances. To find an initial solution to bootstrap the solver's search, we randomly generate 10 node deployment plans and pick the best one among those.

Longest-Link Node Deployment Problem. We provide both a MIP formulation and a CP formulation for LLNDP. Since the parameter of cost clustering may have different impacts on the performance of the two solvers, we first analyze the effect of cost clustering. Figure 6 shows the convergence of the CP formulation for 100 instances with different number of clusters. The communication graph for Figure 6 to Figure 8 is a 2D mesh from the simulation workload and the deployment cost is the cost of the longest link. We tested all possible k values from 5 to the number of distinct values (rounded to nearest 0.01 ms), with an increment of 5. We present three representative configurations: $k = 5$, $k = 20$ and *no clustering*. As we decrease the number of clusters, the CP approach converges faster. Indeed, with no clustering, the best solution is found after 16 minutes, whereas it takes 2 minutes and 30 seconds for the CP approach to converge with $k = 20$ and $k = 5$, respectively. This is mainly due to the fact that fewer iterations are needed to reach the optimal value. Such a difference demonstrates the effectiveness of cost clustering in reducing the search time. On the other hand, the smaller the value of k is, the coarser the cost clusters are. As a result, the CP model cannot discriminate among

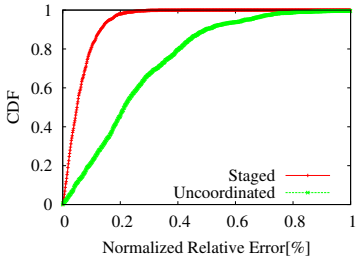

Figure 4: Staged vs Uncoordinated (against Token Passing)

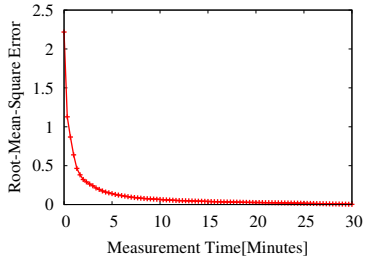

Figure 5: Latency Measurement Convergence over Time

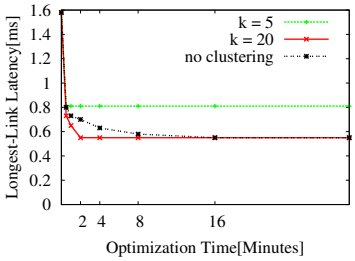

Figure 6: CP Convergence with k-means: Longest Link

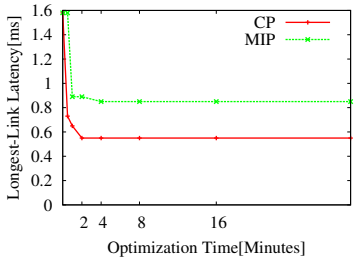

Figure 7: CP vs MIP Convergence: Longest Link (k=20)

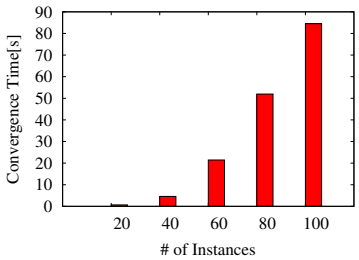

Figure 8: CP Solver Scalability: Longest Link

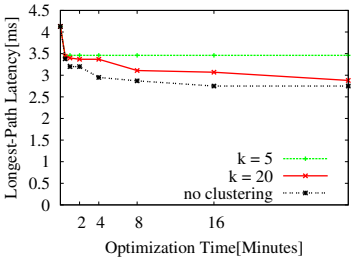

Figure 9: MIP Convergence with k-means: Longest Path

deployments within the same cost cluster, and this might lead to sub-optimal solutions. As shown in Figure 6, the solver cannot find a solution with a deployment cost smaller than 0.81 for $k = 5$, while both cases $k = 20$ and *no clustering* lead to a much better deployment cost of 0.55.

Figure 7 shows the comparison between the CP and the MIP formulations with $k = 20$. MIP performs poorly with the scale of 100 instances. Also, other clustering configurations do not improve the performance of MIP. One reason is that for LLNDP, the encoding of the MIP is much less compact than CP. Moreover, the MIP formulation suffers from a weak linear relaxation, as x_{ij} and $x_{i'j'}$ should add up to more than one for the relaxed constraint 3 to take effect.

Given the above analysis, we pick CP with $k = 20$ for the following scalability experiments as well as LLNDP in Section 6.4.

Measuring scalability of a solver such as ours is challenging, as problem size does not necessarily correlate with problem hardness. To observe scalability behavior with problem size, we generate multiple inputs for each size and measure the average convergence time of the solver over all inputs. The multiple inputs for each size are obtained by randomly choosing 50 subsets of instances out of our initial 100-instance allocation. The convergence time corresponds to the time the solver takes to not be able to improve upon the best found solution within one hour of search. Figure 8 shows the scalability of the solver with the CP formulation. We observe that average convergence time increases acceptably with the problem size. At the same time, at every size, the solver is able to devise node deployment plans with similar average deployment cost improvement ratios.

Longest-Path Node Deployment Problem. Figure 9 shows the convergence of the MIP formulation for 50 instances with different number of link cost clusters. The communication graph is an aggregation tree with depth less than or equal to 4. Similar to Figure 6, the solver performs poorly under the configuration of $k = 5$. Interestingly, clustering costs does not improve performance for LPNDP. This is because the costs are aggregated using summation over the path for LPNDP, and therefore the solver cannot take advantage of having fewer distinct values. We therefore use MIP with *no clustering* for LPNDP in Section 6.4.

Comparison with a Greedy Algorithm. A greedy algorithm can be used as a lightweight approach to quickly find a (usually sub-optimal) solution for the Node Deployment Problem: 1) Start with an empty deployment; 2) find a link (u,v) with the lowest cost between the current partial deployment and a node v not in it, and add the node v to the partial deployment via (u,v); and 3) repeat Step 2 until all nodes are included. We experiment with this algorithm over 20 different allocations with 50 nodes. On average, the deployment cost of the solution generated by the greedy algorithm is 65.7% higher than the solution returned by our solver using the CP formulation with a 2-minute optimization time. The reason is that although the links (u,v) explicitly picked by the greedy algorithm typically do not have high cost, the implicit additional links introduced often have substantial cost.

6.4 ClouDiA Effectiveness

Setup. We evaluate the overall ClouDiA system in EC2 with 100 to 150 instances over different allocations. Other settings are the same as in Section 6.2 (network measurement) and Section 6.3 (solver). We use a 10% over-allocation ratio for the first experiment and vary this ratio in the second experiment. The deployment decision made by ClouDiA is compared with the default deployment, which uses the instance ordering returned by the EC2 allocation command. Note that EC2 does not offer us any control on how to place instances, so the over-allocated instances we obtain are just the ones returned by the *ec2-run-instance* command.

Overall Effectiveness. We show the overall percentage of improvement over 5 different allocations in EC2 in Figure 10. The behavioral simulation and key-value store workloads use 100 application nodes, whereas the aggregation query workload uses 50 nodes due to the scalability limits of the LPNDP solver. For the simulation workload, we report the reduction in time-to-solution. For the aggregation query and key-value store workloads, we report the reduction in response time. Each of these is averaged based on an at least 10 minutes of observation for both. We compare the performance of ClouDiA optimized deployment to the default deployment. ClouDiA achieves 15% to 55% reduction in time-to-solution or response time over 5 allocations for 3 workloads. The reduction

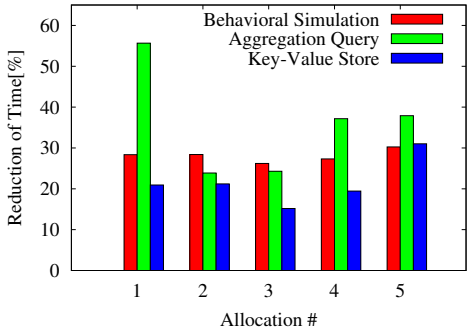

Figure 10: Overall Improvement in EC2

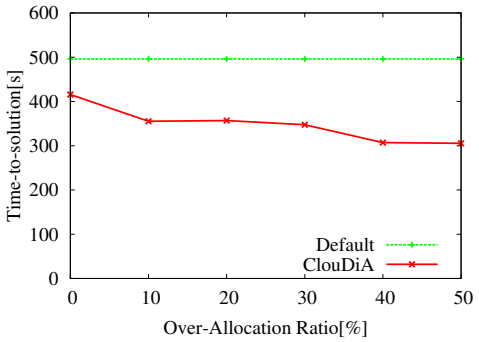

Figure 11: Effect of Over-Allocation in EC2: Behavioral Simulation

ratio varies for different allocations. Among three workloads, we observe the largest reduction ratio on average in the aggregation query workload, while the key-value store workload gets less improvement than the others. There are two reasons for this effect. First, the communication graph of the aggregation query workload has the fewest edges, which increases the probability that the solver can find a high quality deployment. Second, the Longest-Link deployment cost function does not exactly match the mean response time measurement of the key-value store workload, and therefore deployment decisions are made less accurately.

Effect of Over-allocation. In Figure 11, we study the benefit of over-allocating instances for increasing the probability of finding a good deployment plan. Note that although ClouDiA terminates extra instances once the deployment plan is determined, these instances will still be charged for at least one hour usage due to the round-up pricing model used by major cloud service providers [2, 64, 48]. Therefore, a trade-off must be made between performance and initial allocation cost. In this experiment, we use an application workload similar to Figure 10, but with 150 EC2 instances allocated at once by a single *ec2-run-instance command*. To study the case with over-allocation ratio x, we use the first $(1+x)\cdot 100$ instances out of the 150 instances by the EC2 default ordering. Figure 11 shows the improvement in time-to-solution for the simulation workload. The default deployment always uses the first 100 instances, whereas ClouDiA searches deployment plans with the $100x$ extra instances. We report 38% performance improvement with 50% extra instances over-allocated. Without any over allocation, 16% improvement is already achieved by finding a good injection of application nodes to instances. Interestingly, with only 10% instance over-allocation, 28% improvement is achieved. Similar observations are found on other allocations as well.

7. RELATED WORK

Subgraph Isomorphism. The *subgraph isomorphism problem* is known to be NP-Complete [26]. There is an extensive literature about algorithms for special cases of the subgraph isomorphism problem, e.g., for graphs of bounded genus [40], grids [41], or planar graphs [22]. Algorithms based on searching and pruning [17, 18, 57] as well as constraint programming [37, 68] have been used to solve the general case of the problem. In our Node Deployment Problem, a mapping from application nodes to instances needs to be determined as in the subgraph isomorphism problem, but in addition the mapping must minimize the deployment cost.

Overlay Placement. Another way to look at the Node Deployment Problem is to find a good overlay within the allocated instances. The networking community has invested significant effort in intelligently placing intermediate overlay nodes to optimize Internet routing reliability and TCP performance [29, 54]. This community has also investigated node deployment in other contexts, such as proxy placement [39] and web server/cache placement [35, 47]. In the database community, there have been studies in extensible definition of dissemination overlays [45], as well as operator placement for stream-processing systems [46]. In contrast to all of these previous approaches, ClouDiA focuses on optimizing the direct end-to-end network performance without changing routing infrastructure in a datacenter setting.

Virtual Network Embedding. Both the virtual network embedding problem [14, 16, 24, 65] and the testbed mapping problem [53] map nodes and links in a virtual network to a substrate network taking into account various resource constraints, including CPU, network bandwidth, and permissible delays. Traditional techniques used in solving these problems cannot be applied to our public cloud scenario simply because treating the entire datacenter as a substrate network would exceed the instance sizes they can handle. Recent work provides more scalable solutions for resource allocation at the datacenter scale by greedily exploiting server locality [8, 28]. However, this work does not take network latency into consideration. CloudDiA focuses on latency-sensitive applications and examines a different angle: We argue network heterogeneity is unavoidable in public clouds and therefore optimize the deployment as a cloud tenant rather than the infrastructure provider. Such role changing enables us to frame the problem as an application tuning problem and better capture optimization goals relevant to latency-sensitive applications. Also, we only need to consider instances allocated by a given tenant, which is a substantially smaller set than the entire data center. Of course, nothing precludes the methodology provided by ClouDiA being incorporated by the cloud provider upon allocation for a given tenant, as long as the provider can obtain latency-sensitivity information from the application.

Auto-tuning in the Cloud. In the database community, there is a long tradition of auto-tuning approaches, with AutoAdmin [13] and COMFORT [61] as some of its seminal projects. Recently, more attention has focused on auto-tuning in the cloud setting. Babu investigates how to tune the parameters of MapReduce programs automatically [6], while Jahani et al. automatically analyze and optimize MapReduce programs with data-aware techniques [32]. Lee et al. optimizes resource allocation for data-intensive applications using a prediction engine [38]. Conductor [63] assists public cloud tenants in finding the right set of resources to save cost. Both of the above two approaches are similar in spirit to ClouDiA. However, they focus on Map-reduce style computation with high bandwidth consumption. Our work differs in that we focus on latency-sensitive applications in the cloud, and develop appropriate auto-tuning techniques for this different setting.

Cloud Orchestration. AWS CloudFormation [4] allows tenants to provision and manage various cloud resources together using templates. However, interconnection performance requirements cannot be specified. AWS also supports cluster placement groups and guarantees low network latency between instances within the same placement group. Only costly instance types are supported and the number of instances that can be allocated to the same placement group is restricted. HP Intelligent Management Center Virtual Application Network Manager [30] orchestrates virtual machine network connectivity to ease application deployment. Although it allows tenants to specify an 'information rate' for each instance, there is no guarantee on pairwise network performance characteristics, especially network latency. Wrasse [49] provides a generic tool for cloud service providers to solve allocation problems. It does not take network latency into account.

Topology-Aware Distributed Systems. Many recent large-scale distributed systems built for data centers are aware of network topology. Cassandra [36] and Hadoop DFS [12] both provide policies to prevent rack-correlated failure by spreading replicas across racks. DyradLINQ [66] runs rack-level partial aggregation to reduce cross-rack network traffic. Purlieus [44] explores data locality for MapReduce tasks also to save cross-rack bandwidth. Quincy [31] studies the problem of scheduling with not only locality but also fairness under a fine-grained resource sharing model. The optimizations in these previous approaches are both rack- and application-specific. By contrast, ClouDiA takes into account arbitrary levels of difference in mean latency between instances. In addition, ClouDiA is both more generic and more transparent to applications.

Network Performance in Public Clouds. Public clouds have been demonstrated to suffer from latency jitter by several experimental studies [55, 59]. Our previous work has proposed a general framework to make scientific applications jitter-tolerant in a cloud environment [69], allowing applications to tolerate latency spikes. However, this work does little to deal with stable differences in mean latency. Zaharia et al. observed network bandwidth heterogeneity due to instance colocation in public clouds and has designed a speculative scheduling algorithm to improve response time of MapReduce tasks [67]. Farley et al. also exploit such network bandwidth as well as other types of heterogeneity in public clouds to improve performance [25]. To the best of our knowledge, ClouDiA is the first work that experimentally observes network *latency* heterogeneity in public clouds and optimizes application performance by solving the Node Deployment Problem.

8. CONCLUSIONS

We have shown how ClouDiA makes intelligent deployment decisions for latency-sensitive applications under heterogeneous latencies, which naturally occur in public clouds. We formulated the deployment of applications into public clouds as optimization problems and proposed techniques to speed up the search for high-quality deployment plans. We also presented how to efficiently obtain latency measurements without interference. Finally, we evaluated ClouDiA in Amazon EC2 with realistic workloads. ClouDiA is able to reduce the time-to-solution or response time of latency-sensitive applications by 15% to 55%, without any changes to application code.

As future work, we plan to extend our formulation to support weighted communication graphs. Another direction of practical importance is quantifying over-allocation cost and analyzing its impact on total cost-to-solution for scientific applications. Finally, we will investigate the deployment problem under other criteria, such as bandwidth, as well as other deployment cost functions.

Acknowledgments. The third author took inspiration for the title from the name of an early mentor, Claudia Bauzer Medeiros, to whom he is thankful for introducing him to database research during his undergraduate studies. We thank the anonymous PVLDB reviewers for their insightful comments which helped us to improve this paper. This research has been supported by the NSF under Grants CNS-1012593, IIS-0911036, by an AWS Grant from Amazon, the AFOSR under Award FA9550-10-1-0202 and by the iAd Project funded by the Research Council of Norway. Any opinions, findings, conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of the sponsors.

9. REFERENCES

[1] R. D. Alpert and J. F. Philbin. cBSP: Zero-cost synchronization in a modified BSP model. Technical report, NEC Research Institute, 1997.

[2] Amazon web services, elastic compute cloud (ec2). http://aws.amazon.com/ec2/.

[3] Amazon web services, case studies. http://aws.amazon.com/solutions/case-studies/.

[4] Amazon web services, cloudformation. http://aws.amazon.com/cloudformation/.

[5] Amazon web services, search engines & web crawlers. http://aws.amazon.com/search-engines/.

[6] S. Babu. Towards automatic optimization of mapreduce programs. In *SOCC*, 2010.

[7] C. S. Badue, R. A. Baeza-Yates, B. A. Ribeiro-Neto, and N. Ziviani. Distributed query processing using partitioned inverted files. In *SPIRE*, pages 10–20, 2001.

[8] H. Ballani, P. Costa, T. Karagiannis, and A. I. T. Rowstron. Towards predictable datacenter networks. In *SIGCOMM*, 2011.

[9] D. Battré, N. Frejnik, S. Goel, O. Kao, and D. Warneke. Evaluation of network topology inference in opaque compute clouds through end-to-end measurements. In *IEEE CLOUD*, 2011.

[10] T. Benson, A. Akella, and D. A. Maltz. Network traffic characteristics of data centers in the wild. In *Internet Measurement Conference*, 2010.

[11] O. Bonorden, B. Juurlink, I. von Otte, and I. Rieping. The paderborn university BSP (PUB) library. *Parallel Computing*, 29(2):187–207, 2003.

[12] D. Borthakur. The hadoop distributed file system: Architecture and design. http://hadoop.apache.org/core/docs/current/hdfs design.pdf.

[13] S. Chaudhuri and V. Narasayya. An efficient, cost-driven index selection tool for microsoft SQL server. In *VLDB*, 1997.

[14] X. Cheng, S. Su, Z. Zhang, H. Wang, F. Yang, Y. Luo, and J. Wang. Virtual network embedding through topology-aware node ranking. *SIGCOMM CCR*, 41(2):38–47, 2011.

[15] M. Chowdhury, M. Zaharia, J. Ma, M. I. Jordan, and I. Stoica. Managing data transfers in computer clusters with orchestra. In *SIGCOMM*, 2011.

[16] N. M. M. K. Chowdhury, M. R. Rahman, and R. Boutaba. Virtual Network Embedding with Coordinated Node and Link Mapping. In *INFOCOM*, 2009.

[17] L. P. Cordella, P. Foggia, C. Sansone, and M. Vento. Performance evaluation of the VF graph matching algorithm. In *International Conference on Image Analysis and Processing*, 1999.

[18] L. P. Cordella, P. Foggia, C. Sansone, and M. Vento. A (sub)graph isomorphism algorithm for matching large graphs. *IEEE Transactions on PAMI*, 26:1367–1372, 2004.

[19] I. Couzin, J. Krause, N. Franks, and S. Levin. Effective leadership and decision-making in animal groups on the move. *Nature*, 433(7025):513–516, 2005.

[20] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's highly available key-value store. In *SOSP*, 2007.

[21] J. Demmel, M. Hoemmen, M. Mohiyuddin, and K. A. Yelick. Avoiding communication in sparse matrix computations. In *IPDPS*, 2008.

[22] D. Eppstein. Subgraph isomorphism in planar graphs and related problems. In *SODA*, 1995.

[23] C. Evangelinos and C. N. Hill. Cloud computing for parallel scientific HPC applications. In *Cloud Computing and Its Applications*, 2008.

[24] I. Fajjari, N. Aitsaadi, G. Pujolle, and H. Zimmermann. VNE-AC: Virtual network embedding algorithm based on ant colony metaheuristic. In *IEEE International Conference on Communications*, 2011.

[25] B. Farley, V. Varadarajan, K. Bowers, A. Juels, T. Ristenpart, and M. Swift. More for your money: Exploiting performance heterogeneity in public clouds. In *SOCC*, 2012.

[26] M. R. Garey and D. S. Johnson. *Computers and intractability*. Freeman, 1979.

[27] R. Geambasu, S. D. Gribble, and H. M. Levy. Cloudviews: Communal data sharing in public clouds. In *HotCloud*, 2009.

[28] C. Guo, G. Lu, H. J. Wang, S. Yang, C. Kong, P. Sun, W. Wu, and Y. Zhang. Secondnet: a data center network virtualization architecture with bandwidth guarantees. In *CoNEXT*, 2010.

[29] J. Han, D. Watson, and F. Jahanian. Topology aware overlay networks. In *INFOCOM*. IEEE, 2005.

[30] HP intelligent management center virtual application network manager. http://h17007.www1.hp.com/us/en/products/network-management/IMC_VANM_Software/index.aspx.

[31] M. Isard, V. Prabhakaran, J. Currey, U. Wieder, K. Talwar, and A. Goldberg. Quincy: fair scheduling for distributed computing clusters. In *SOSP*, 2009.

[32] E. Jahani, M. J. Cafarella, and C. Ré. Automatic optimization for mapreduce programs. *PVLDB*, 2011.

[33] G. Juve, E. Deelman, K. Vahi, G. Mehta, B. Berriman, B. P. Berman, and P. Maechling. Scientific workflow applications on amazon EC2. In *IEEE International Conference on e-Science.*, 2009.

[34] J.-S. Kim, S. Ha, and C. S. Jhon. Efficient barrier synchronization mechanism for the BSP model on message-passing architectures. In *IPPS/SPDP*, 1998.

[35] P. Krishnan, D. Raz, and Y. Shavitt. The cache location problem. *IEEE/ACM Transactions on Networking*, 8(5):568–582, 2000.

[36] A. Lakshman and P. Malik. Cassandra: a decentralized structured storage system. *SIGOPS OSR*, 44(2):35–40, 2010.

[37] J. Larrosa and G. Valiente. Constraint satisfaction algorithms for graph pattern matching. *Mathematical. Structures in Comp. Sci.*, 12(4):403–422, Aug. 2002.

[38] G. Lee, N. Tolia, P. Ranganathan, and R. H. Katz. Topology-aware resource allocation for data-intensive workloads. *SIGCOMM CCR*, 2011.

[39] B. Li, M. J. Golin, G. F. Italiano, X. Deng, and K. Sohraby. On the optimal placement of web proxies in the internet. In *INFOCOM*, 1999.

[40] G. L. Miller. Isomorphism testing for graphs of bounded genus. In *STOC*, 1980.

[41] Z. Miller and J. B. Orlin. Np-completeness for minimizing maximum edge length in grid embeddings. *J. Algorithms*, pages 10–16, 1985.

[42] R. N. Mysore, A. Pamboris, N. Farrington, N. Huang, P. Miri, S. Radhakrishnan, V. Subramanya, and A. Vahdat. Portland: a scalable fault-tolerant layer 2 data center network fabric. In *SIGCOMM*, 2009.

[43] C. O'Hanlon. A conversation with werner vogels. *ACM Queue*, 4(4):14–22, May 2006.

[44] B. Palanisamy, A. Singh, L. Liu, and B. Jain. Purlieus: locality-aware resource allocation for mapreduce in a cloud. In *SC*, 2011.

[45] O. Papaemmanouil, Y. Ahmad, U. Çetintemel, J. Jannotti, and Y. Yildirim. Extensible optimization in overlay dissemination trees. In *SIGMOD*, 2006.

[46] P. R. Pietzuch, J. Ledlie, J. Shneidman, M. Roussopoulos, M. Welsh, and M. I. Seltzer. Network-aware operator placement for stream-processing systems. In *ICDE*, 2006.

[47] L. Qiu, V. N. Padmanabhan, and G. M. Voelker. On the placement of web server replicas. In *INFOCOM*, 2001.

[48] Rackspace. http://www.rackspace.com/.

[49] A. Rai, R. Bhagwan, and S. Guha. Generalized resource allocation for the cloud. In *SOCC*, 2012.

[50] L. Ramakrishnan, K. R. Jackson, S. Canon, S. Cholia, and J. Shalf. Defining future platform requirements for e-Science clouds. In *SOCC*, 2010.

[51] R. Ramakrishnan. Data serving in the cloud. In *LADIS*, 2010.

[52] V. Ramasubramanian, D. Malkhi, F. Kuhn, M. Balakrishnan, A. Gupta, and A. Akella. On the treeness of internet latency and bandwidth. In *SIGMETRICS*, 2009.

[53] R. Ricci, C. Alfeld, and J. Lepreau. A solver for the network testbed mapping problem. *SIGCOMM CCR*, 33(2):65–81, 2003.

[54] S. Roy, H. Pucha, Z. Zhang, Y. C. Hu, and L. Qiu. Overlay node placement: Analysis, algorithms and impact on applications. In *ICDCS*, 2007.

[55] J. Schad, J. Dittrich, and J.-A. Quiané-Ruiz. Runtime measurements in the cloud: Observing, analyzing, and reducing variance. *PVLDB*, 3(1), 2010.

[56] Y. J. Song, M. K. Aguilera, R. Kotla, and D. Malkhi. RPC chains: Efficient client-server communication in geodistributed systems. In *NSDI*, 2009.

[57] J. R. Ullmann. An algorithm for subgraph isomorphism. *Journal of the ACM*, 23, 1976.

[58] W. Vogels. Data access patterns in the amazon.com technology platform. In *VLDB*, page 1, 2007.

[59] G. Wang and T. S. E. Ng. The impact of virtualization on network performance of Amazon EC2 data center. In *INFOCOM*, 2010.

[60] G. Wang, M. A. V. Salles, B. Sowell, X. Wang, T. Cao, A. J. Demers, J. Gehrke, and W. M. White. Behavioral simulations in mapreduce. *PVLDB*, 2010.

[61] G. Weikum, C. Hasse, A. Moenkeberg, and P. Zabback. The COMFORT automatic tuning project, invited project review. *Inf. Syst.*, 19(5), 1994.

[62] Y. Wen. *Scalability of Dynamic Traffic Assignment*. PhD thesis, Massachusetts Institute of Technology, 2008.

[63] A. Wieder, P. Bhatotia, A. Post, and R. Rodrigues. Orchestrating the deployment of computations in the cloud with conductor. In *NSDI*, 2012.

[64] Windows azure. http://www.windowsazure.com/.

[65] M. Yu, Y. Yi, J. Rexford, and M. Chiang. Rethinking virtual network embedding: substrate support for path splitting and migration. *SIGCOMM CCR*, 38(2):17–29, 2008.

[66] Y. Yu, M. Isard, D. Fetterly, M. Budiu, Ú. Erlingsson, P. K. Gunda, and J. Currey. DryadLINQ: A system for general-purpose distributed data-parallel computing using a high-level language. In *OSDI*, 2008.

[67] M. Zaharia, A. Konwinski, A. D. Joseph, R. H. Katz, and I. Stoica. Improving mapreduce performance in heterogeneous environments. In *OSDI*, 2008.

[68] S. Zampelli, Y. Deville, and C. Solnon. Solving subgraph isomorphism problems with constraint programming. *Constraints*, 15(3):327–353, 2010.

[69] T. Zou, G. Wang, M. V. Salles, D. Bindel, A. Demers, J. Gehrke, and W. White. Making time-stepped applications tick in the cloud. In *SOCC*, 2011.

APPENDIX

A. PROBLEM COMPLEXITY

THEOREM 1. *The Longest-Link Node Deployment Problem (LLNDP) is NP-hard.*

PROOF. We reduce the *Subgraph Isomorphism Problem* (SIP), which is known to be NP-Hard [26], to the LLNDP. Consider an instance of SIP, where $G_1 = (V_1, E_1), G_2 = (V_2, E_2)$, and we look for a mapping $\sigma : V_1 \to V_2$ such that whenever $(i, j) \in E_1$, then $(\sigma(i), \sigma(j)) \in E_2$. We build an instance of the LLNDP as follows. We set $N = V_1$, $S = V_2$, $G = (V_1, E_1)$, and the costs $\mathcal{C}_{\mathcal{L}}(i, j)$ to 1 whenever the edge (i, j) belongs to E_2, and to 2 otherwise. By solving LLNDP, we get a deployment plan $\mathcal{D}$. G_2 contains a subgraph isomorphic to G_1 whenever $\mathcal{C}_{\mathcal{D}}^{\mathsf{LL}} = 1$ and $\sigma = \mathcal{D}$. □

In order to show hardness of approximation, we will assume in the next two theorems that all communication costs are distinct. This assumption is fairly realistic, even more so as these costs are typically real numbers that are experimentally measured.

THEOREM 2. *There is no α-absolute approximation algorithm to the Longest-Link Node Deployment Problem in the case where all communication costs are distinct, unless* P=NP.

PROOF. Consider an instance I of the LLNDP, consisting of $G = (N, E)$, $\mathcal{C}_{\mathcal{L}}(i, j)$ where $i, j \in S$, and $\mathcal{C}_{\mathcal{L}}(i, j) = \mathcal{C}_{\mathcal{L}}(i', j')$ if and only if $i = i'$ and $j = j'$. We strictly order the all communication links $\big((i_1, j_1), (i_2, j_2) \ldots, (i_{|N|^2}, j_{|N|^2})\big)$ in increasing order of their communication costs. Let (i_w, j_w) be the Longest Link used in the optimal solution for I, with optimal value $\mathcal{C}_{\mathcal{L}}(i_w, j_w)$. Notice that any instance of LLNDP that shares the same G and S as well as the same exact ordering of the communication links will also have an optimal value equal to $\mathcal{C}_{\mathcal{L}}(i_w, j_w)$. Now, assume that there is an α-absolute approximation algorithm $\mathcal{A}$ to LLNDP. We build an instance I' by changing the communication costs in I to $\mathcal{C}_{\mathcal{L}}(i_k, j_k) = (\alpha + 1)k$. Note that I and I' share the same ordering of the communication links, and any two links in I' have communication costs that are at least $\alpha + 1$ apart. Since $\mathcal{A}$ returns a longest link $(i_{w'}, j_{w'})$ for I' such that $\mathcal{C}_{\mathcal{L}}(i_{w'}, j_{w'}) \leq \mathcal{C}_{\mathcal{L}}(i_w, j_w) + \alpha$, $\mathcal{A}$ actually solves I' optimally. The fact that LLNDP is NP-hard completes the proof. □

THEOREM 3. *There is no ε-relative approximation algorithm to the Longest-Link Node Deployment Problem in the case where all costs are distinct, unless* P=NP.

PROOF. As in the previous proof, we build an instance I' that differs from I only by the costs of the links. We set these costs to be $\mathcal{C}_{\mathcal{L}}(i_k, j_k) = (\varepsilon + 1)^k$ for every link (i, j) where $i, j \in S$. In that case, for any two links (i_p, j_p) and (i_q, j_q) where $p < q$, we have $\mathcal{C}_{\mathcal{L}}(i_p, j_p) < \varepsilon \cdot \mathcal{C}_{\mathcal{L}}(i_q, j_q)$. The fact that a ε-relative approximation algorithm would return a longest link $(i_{w'}, j_{w'})$ of I' such that $\mathcal{C}_{\mathcal{L}}(i_{w'}, j_{w'}) \leq \varepsilon \cdot \mathcal{C}_{\mathcal{L}}(i_w, j_w)$ completes the proof. □

THEOREM 4. *The Longest-Path Node Deployment Problem (LPNDP) is NP-hard.*

PROOF. The proof is otherwise identical to the proof of Theorem 1 except when (i, j) does not belong to E_2, we set $\mathcal{C}_{\mathcal{L}}(i, j)$ to $|E_1| + 1$ and the final check is now $\mathcal{C}_{\mathcal{D}}^{\mathsf{LP}} \leq |E_1|$. □

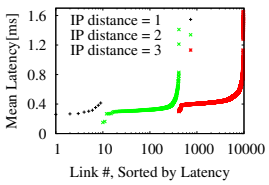

Figure 12: Latency order by IP distance

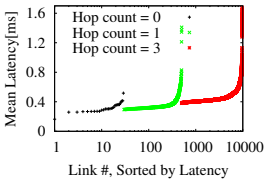

Figure 13: Latency order by Hop Count

B. DISTANCE APPROXIMATIONS

All techniques in Section 5 may require non-negligible measurement time to obtain pairwise mean latencies. We also experimented with the following two approximations to network distance, which are both simple and intuitively related to mean latency.

1. IP Distance. Our first approximation makes use of internal IPv4 addresses in the cloud. The hypothesis is that if two instances share a common /24 address prefix, then these instances are more likely to be located in the same or in a nearby rack than if the two instances only share a common /8 prefix. We can therefore define *IP distance* as a measure of the dissimilarity between two IP addresses: Two instances sharing the same /x address prefix but not /x+1 address prefix have IP distance 32 - x. We can future adjust the sensitive of this measurement by considering $g (1 \leq g < 32)$ consecutive bits of the IP addresses together.

2. Hop Count. A slightly more sophisticated approximation is the hop count between two instances. The hop count is the number of intermediate routers through which data flows from source to destination. Hop count can be obtained by sending packets from source to destination and monitoring the Time To Live (TTL) field of the IP header.

Experimental Evaluation. We evaluated the two approximations above with the same experimental setup described in Section 6.2. We compare both IP distance and hop count against the mean round-trip latency measurement results obtained using the *staged* approach described in Section 5.

In Figure 12, we show the effect of using IP distances as an approximation. In this experiment, we consider 8 consecutive bits of the IP address together: two instances sharing a /24 address prefix have IP distance 1; two instances with the same /16 prefix but not /24 prefix have IP distance 2, and so on. We also experimented with other sensitivity configurations and the results are similar. The x axis is in log scale and divided into 3 groups based on the value of IP distance. Since we used the internal IP addresses of EC2, all of which share the same /8 prefix, we do not observe IP distance of 4. Within each group, links are ordered by round-trip latency measurements. If IP distance were a desirable approximation, we would expect that pairs with higher IP distance would also have higher latencies. Figure 12 shows that such monotonicity does not always hold. For example, within the group of IP distance $= 2$, there exist links having lower latency than some links of IP distance $= 1$, as well as links having higher latency than some links of IP distance$= 3$. Interestingly, the lowest latencies are observed in pairs with IP distance$= 2$.

The effect of using hop count as an approximation is shown in Figure 13. Similarly, the x axis is in log scale and divided into 3 groups based on hop count. We do not observe any pair of instances that are 2 hops apart. Within each group, links are ordered by round-trip latencies. As in Figure 12, there exists a significant number of link pairs ordered inconsistently by hop count and measured latency.

The above results demonstrate that IP distance and hop count, though easy to obtain, do not effectively predict network latency.