# Declarative Automated Cloud Resource Orchestration

Changbin Liu
University of Pennsylvania
3330 Walnut St, Philadelphia
PA, USA
changbl@seas.upenn.edu

Boon Thau Loo
University of Pennsylvania
3330 Walnut St, Philadelphia
PA, USA
boonloo@seas.upenn.edu

Yun Mao
AT&T Labs - Research
180 Park Ave, Florham Park,
NJ, USA
maoy@research.att.com

## ABSTRACT

As cloud computing becomes widely deployed, one of the challenges faced involves the ability to orchestrate a highly complex set of subsystems (compute, storage, network resources) that span large geographic areas serving diverse clients. To ease this process, we present COPE (Cloud Orchestration Policy Engine), a distributed platform that allows cloud providers to perform declarative automated cloud resource orchestration. In COPE, cloud providers specify system-wide constraints and goals using COPElog, a declarative policy language geared towards specifying distributed constraint optimizations. COPE takes policy specifications and cloud system states as input and then optimizes compute, storage and network resource allocations within the cloud such that provider operational objectives and customer SLAs can be better met. We describe our proposed integration with a cloud orchestration platform, and present initial evaluation results that demonstrate the viability of COPE using production traces from a large hosting company in the US. We further discuss an orchestration scenario that involves geographically distributed data centers, and conclude with an ongoing status of our work.

## Categories and Subject Descriptors

K.6.4 [**Computing Milieux**]: Management Of Computing And Information Systems—*System Management*; C.2.4 [**Computer Systems Organization**]: Computer Communication Networks—*Distributed Systems*

## General Terms

Design, Languages, Management

## Keywords

Cloud computing, Resource orchestration, Declarative queries, Distributed optimizations

## 1. INTRODUCTION

Cloud resource orchestration [16] involves the creation, management, manipulation and decommissioning of cloud resources, i.e., compute, storage and network, in order to realize customer requests, while conforming to operational objectives of the cloud service providers at the same time. We argue that cloud orchestration is highly complex. First, as many recent proposals [22, 25, 4, 24, 5] have articulated, cloud management is inherently complicated due to the scale, heterogeneity, infrastructure, and concurrent user services that share a common set of physical resources. Second, configurations of various resource types interact with each other. For example, the locations of virtual machines (VMs) have an impact on storage placement, which in turn affect bandwidth utilization within and external to a data center. Third, cloud resources have to be deployed in a fashion that not only realizes provider operational objectives, but also guarantees that the customer service-level agreements (SLAs) can be constantly met as runtime conditions change. All in all, the orchestration process is complex and potentially error prone if performed manually, and motivates the need for better management tools that enable us to automate part or all of the decision process.

As a first step towards addressing these challenges, this paper presents our initial work on COPE (Cloud Orchestration Policy Engine), a platform that enables cloud providers to automate the process of cloud orchestration via the use of declarative policy languages. COPE allows cloud providers to formally model cloud resources and formulate orchestration decisions as a constraint optimization problem given goals and cloud constraints. In COPE, we envision a distributed network of controllers, each of which resides over a cloud orchestration platform [16], coordinating resources across multiple data centers. Each COPE node utilizes a constraint solver for efficiently generating the set of orchestration commands, and a distributed query engine [2, 17] for communicating policy decisions among different COPE nodes. Specifically, this paper makes the following contributions:

- **Cloud orchestration as a constraint optimization problem.** We propose a unified framework for mathematically modeling cloud resources orchestration as a constraint optimization problem (COP). Operational objectives and customer SLAs are specified in terms of *goals*, which are subjected to a number of *constraints* specific to the cloud deployment scenario. These specifications are then fed to a COPE constraint solver, which automatically synthesizes orchestration commands. We show that

this framework is general enough to capture a variety of cloud resource orchestration scenarios within and across data centers.

- **Declarative configuration.** To ease the process of specifying the above models, we provide a declarative policy language called *COPElog* that allows cloud providers to specify goals and constraints of resource orchestration. To validate COPE, we conduct an experiment based on actual hosting traces obtained from a large hosting company in the US. In 8 COPElog rules, we demonstrate the ability of COPE to perform automated VM migrations to ensure good load distribution within data centers.

- **Distributed optimizations.** To scale the above configuration process and provide autonomy to smaller groups of local administrators such as *federated cloud* [6] infrastructures, we extend the earlier formulation to consider a distributed optimizations-based approach. This approach allows different cloud operators to configure a smaller set of resources while coordinating amongst themselves to achieve a global objective. COPE utilizes a distributed query engine integrated with constraint solving capabilities for coordinating distributed optimizations. We demonstrate its feasibility via a case study involving dynamic load-balancing across data centers. We conclude with a discussion of open issues and challenges, and speculate how distribution optimizations may apply to a number of emerging cloud scenarios.

## 2. SYSTEM OVERVIEW

Figure 1 presents a system overview of COPE, which is designed for a cloud environment comprising of geographically distributed data centers via dedicated backbone networks or the Internet. COPE can be deployed in either a *centralized* or *distributed* mode.
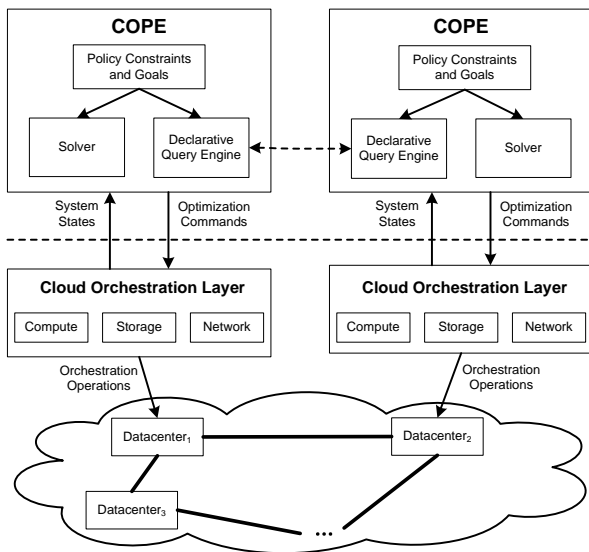


**Figure 1: COPE in the Cloud**

In a *centralized* deployment scenario, the entire cloud is configured by one centralized COPE node. It takes as in-

put system states (e.g. node CPU load, memory usage, network traffic statistics) gathered from the cloud, and a set of provider policy constraints and goals specified in a declarative policy language called COPElog. These specifications are then used by a *constraint solver* to automatically generate *optimization commands*. These commands are then fed into a *cloud orchestration layer* to generate physical orchestration operations to directly manipulate cloud resources. Our target orchestration layer is Data-centric Management Framework (DMF) [16], which provides high-level orchestration commands relieving operators from working with ad-hoc configuration scripts. DMF further provides a convenient *transactional semantics* for orchestration commands, where the ACID property is preserved for a series of orchestration commands grouped as a transaction.

In a *distributed* deployment scenario, there are multiple COPE instances, typically one for each data center. A distributed deployment brings two advantages. First, cloud environments like federated cloud [6] may be administered by different cloud providers. This necessitates each provider running its own COPE for its internal configuration, but coordinate with other COPE nodes for inter data center configurations. Second, even if the cloud is entirely under one administrative domain, for scalability reasons, each cloud operator may choose to configure a smaller set of resources using local optimization commands. A distributed query engine [2] is used to coordinate the exchange of system states and optimization output amongst COPE nodes, in order to achieve a global objective.

## 3. MOTIVATING EXAMPLES

To illustrate both deployment models, we present two motivating scenarios, *SimpleCloud* and *Follow-the-Sun*.

### 3.1 SimpleCloud

We consider a hypothetical cloud service called *Simple-Cloud*, a simplified version of what cloud service providers might offer. In SimpleCloud, a customer may spawn new VMs from an existing disk image, and later start, shutdown, or delete the VMs. SimpleCloud is realized within a data center, and used to host a variety of client applications. Given that SimpleCloud is located within one data center, one can deploy COPE as a centralized controller node for the entire data center, issuing commands to the cloud orchestration layer to manipulate VM placement.

COPE takes as input real-time system states (e.g., CPU and memory load, migration feasibility), and a set of policies specified by the cloud provider. An example optimization goal is to reduce the cluster-wide CPU load variance across all machines, so as to avoid hot-spots. Constraints can be tied to each machine's resource availability (e.g. each machine can only run up to a fixed number of VMs, run certain classes of VMs, and not exceed its own memory limits), or security concerns (VMs can only be migrated across certain types of machines). Another possible policy is to minimize the total number of VM migrations, as long as a load variance threshold is met across all machines. Alternatively, to consolidate workloads one can minimize the number of hosts that are hosting VMs, as long as each application receives sufficient resources to meet customer demands. Given these optimization goals and constraints, COPE can be executed periodically, or triggered whenever imbalance is observed, or whenever the VM's CPU and memory usage changes.

In today's deployment, providers typically perform load balancing in an ad-hoc fashion. For instance, VM migrations can be triggered at an overloaded host machine, whose VMs are migrated to a randomly chosen machine currently with light load. While such ad-hoc approaches may work for a specific scenario, they are unlikely to result in configurations that can be easily customized upon changing policy constraints and goals, and whose optimality cannot be easily quantified.

## 3.2 Follow-the-Sun

Our second motivating example is based on the *Follow-the-Sun* [22] scenario, which aims to migrate VMs across geographical distributed data centers based on customer dynamics. Here, the geographic location of the primary workload (i.e., majority of customers using the cloud service) derives demand shifts during the course of a day, and it is beneficial for these workload drivers to be in close proximity to the resources they operate on. The migration decision process has to occur in real-time on a live deployment with minimal disruption to existing services.

In this scenario, the cloud infrastructure service aims to optimize for two parties: enable service consolidation (for providers) to reduce operating costs, and improve application performance (for customers), while ensuring that customer SLAs of web services (e.g. defined in terms of the average end-to-end experienced latency of user requests) are met. In addition, they may be performed to reduce inter-data center communication overhead [27, 5]. Since data centers in this scenario may belong to different cloud providers (similar to federated cloud [6]), Follow-the-Sun may be best suited for a distributed deployment, where each COPE node is responsible for controlling resources within their data center.

In the rest of the paper, to illustrate COPE's capabilities, we first focus on the centralized deployment scenario in Section 4 using the SimpleCloud scenario, followed by a discussion of the distributed Follow-the-Sun scenario in Section 5.

## 4. CENTRALIZED ORCHESTRATION

Using the SimpleCloud scenario, we provide a complete example to illustrate COPE's capabilities: (1) expressing cloud orchestration as a *constraint optimization problem* (COP), (2) expressing COP using a declarative policy language, and (3) present a trace-driven evaluation of COPE using actual traces from a large hosting company. We focus on the centralized deployment, and defer discussion of distribution to Section 5.

## 4.1 COP Specifications

A COP takes as input a set of *constraints*, and attempts to find an assignment of values chosen from an domain to a set of *variables* to satisfy the constraints under an *optimization goal*. The goal is typically expressed as a minimization over a cost function of the assignments.

COPE uses a declarative policy language *COPElog* to concisely specify the COP formulation in the form of policy goals and constraints. Additional derivation rules are used to process intermediate system states for use by the solver in processing the COP program. Our declarative language COPElog is based on Datalog, a recursive query language used in the database community for querying graphs. Our

choice of Datalog as a basis for COPElog is driven by Datalog's conciseness in specifying dependencies among system states, including distributed system states that exhibit recursive properties. Its root in logic provides a convenient mechanism for expressing solver goals and constraints. Moreover, there exists distributed Datalog engines [2] that will later facilitate distributed COP computations.

COPElog extends traditional Datalog with constructs for expressing goals and constraints and also distributed computations. These specifications are compiled into execution plans executed by a distributed query engine that includes modules for constraint solving. To illustrate COPElog, we consider the following program, which expresses a COP that aims to achieve load-balancing within a data center. For simplicity, this example is centralized, and we will revisit the distributed extensions in the next section.

```
goal minimize C in stdevCPU(C).

var hostAssign(Hid,Vid) forall vm(Vid,CPU,Mem).

r1 aggCPU(Hid,SUM<CPU>) :- hostAssign(Hid,Vid),
                               vm(Vid,CPU,Mem).
r2 stdevCPU(STDEV<CPU>) :- aggCPU(Hid,CPU).
r3 aggMem(Hid,SUM<Mem>) :- hostAssign(Hid,Vid),
                               vm(Vid,CPU,Mem).

c1 aggMem(Hid,Mem) -> Mem<=mem_thres.
c2 hostAssign(Hid,Vid) -> vm(Vid,CPU,Mem),
                            CPU>load_thres.
c3 hostAssign(Hid,Vid) -> host(Hid).
```

In COPElog, regular Datalog rules are used to generate intermediate tables used by the solver. This is specified as regular Datalog rules of the form `p :- q1, q2, ..., qn`, resulting in the derivation of `p`, whenever the rule body (`q1` and `q2` and `...` and `qn`) is true. We adopt standard Datalog terminology throughout the paper, and refer to each term within a rule (e.g. `q1`, `q2`) as a *predicate*, and the corresponding derivation obtained (e.g. `p`) during rule body execution are referred to as *tuples*. Attributes in upper case refer to variables, while lower case refers to constants.

**Language extensions.** Two reserved keywords `goal` and `var` specify the goal and variables used by the constraint solver. *Constraint rules* of the form `F1 -> F2, F3, ..., Fn`, denotes the logical meaning that whenever `F1` is true, then the rule body (`F2` and `F3` and `...` and `Fn`) must also be true to satisfy the constraint. Unlike a Datalog rule, which derives new values for a predicate, a constraint *restricts* a predicate's allowed values, hence representing an invariant that must be maintained at all times. These are used by the solver to limit the search space when computing the optimization goal.

**Program description.** The above program takes as input `vm(Vid,CPU,Mem)` and `host(Hid)` tables. Each `vm` entry stores information of a virtual machine (VM) uniquely identified by `Vid`. Additional monitored information (i.e. its CPU utilization `CPU` and memory usage `Mem`) are also supplied in each entry. This monitored information can be provided by the cloud infrastructure, which regularly updates CPU and memory utilization attributes in the `vm` table. The `host` table stores the set of currently available machines that can run VMs. Given these input tables, the above program expresses the following:

- **Optimization goal:** Minimize the CPU standard deviation attribute C in `stdevCPU`.

- **Variables:** As output, the solver generates entries in `hostAssign(Hid,Vid)`, where each entry indicates VM `Vid` is assigned to host `Hid`. `Vid` is bounded via the `forall` to all existing VMs stored in `vm(Vid,CPU,Mem)`.

- **Derivation rules:** Rules `r1` aggregates the CPU of all VMs running on each host. Rule `r2` takes the output from `r1` and then computes the system-wide standard deviation of the aggregate CPU load across all hosts. The output from `r2` is later used by the constraint solver for exploring the search space that meets the optimization goal.

- **Constraints:** Constraint `c1` expresses that no host can accommodate VMs whose aggregate memory exceeds its physical limit (a predefined threshold `mem_thres`). Similarly, constraint `c2` restricts migration to only VMs whose CPU load is above a predefined threshold `load_thres`. `c2` is used to reduce unnecessary migrations, by removing the VMs with light load from the list. Constraint `c3` ensures that VMs are only assigned to hosts that are currently available.

**Orchestration.** The DMF orchestration layer executes the above program either as a one-time query, periodically, or in a continuous fashion. The continuous version would require running the program as a continuous query, and then triggering the solver whenever a rule body predicate is updated (e.g. reported changes to monitored CPU or memory usage for a given VM). The output of the solver is the `hostAssign(Hid,Vid)` table. For each entry, the DMF controller will invoke a migration operation if VM `Vid` currently resides in another location other than `Vid`.

Using COPElog, it is easy to customize policies simply by modifying the goals, constraints, and adding additional derivation rules. For instance, if the overhead of VM migration is too high, we can set an optimization goal that minimizes the number of VM migrations. We can also add a rule (continuous query) that triggers the COP program whenever load imbalance is observed (i.e. `C` in `stdevCPU` exceeds a threshold). Alternatively, we can optimize for the fewest number of unique hosts used for migration while meeting customer SLAs when consolidating workloads.

## 4.2 COPElog Compilation and Execution

COPE's compiler and runtime system is implemented by integrating the RapidNet [2] distributed query engine with the Gecode [1] constraint solver. RapidNet was originally designed as a platform for declarative networks [17]. We adopted its usage in order to leverage its distributed Datalog engine. This allows us to execute the derivation rules in COPElog programs using standard query processing techniques involving database operators, such as joins (variable matching in rule body), aggregation (e.g. SUM, STDEV), selection filters, rule head renaming, etc.

Whenever an orchestration decision is required, COPElog programs are compiled into executable code in RapidNet, which invokes Gecode's high-performance constraint solving modules. Our compilation process maps COPElog's `goal`, `var`, and constraints into equivalent COP primitives in Gecode. These modules are invoked either as a *one-time* program submitted to COPE, *periodically* (via periodic timer events generated from COPElog rules), or in a *continuous* fashion triggered by incremental maintenance [18] as the body predicates are updated.

Gecode adopts the standard branch-and-bound searching approach to solve the optimization while exploring the space of variables under constraints (e.g. `c1-3`). In addition to these constraints, rules that use solver results as input (e.g. `r1` and `r3`) are rewritten into constraints to further prune the search space.

One of the interesting aspects of COPE from a query processing standpoint, is our integration of RapidNet (an incremental bottom-up distributed Datalog evaluation engine) and Gecode (a top-down goal-oriented constraint solver). This integration allows us to implement a distributed solver that can perform incremental and distributed constraint optimizations.

## 4.3 Evaluation

To demonstrate the capabilities of COPE, we perform a trace-driven experiment executing the COPElog program for the SimpleCloud scenario. As input to the experiment, we use data center traces obtained from a large hosting company in the US. The data contains up to 248 customers hosted on a total of 1,740 statically allocated physical processors (PPs). Each customer application is deployed on a subset of the PPs. The entire trace is one-month in duration, and the trace primarily consists of sampling of CPU and memory utilization at each PP gathered at 300 seconds interval.

**Workload generation:** From the trace, we generate a workload in a hypothetical cloud environment similar to SimpleCloud where there are 15 physical machines geographically dispersed across 3 data centers (5 hosts each). Each physical machine has 32GB memory. By default we configure two types of VMs: local and migratable. We preallocate 56 local VMs on each of 15 hosts, and 8 migratable VM on each of 12 hosts, where the other 3 hosts serve as storage servers for each of the three data centers. This allows us to simulate a deployment scenario involving about 1000 VMs. The workload is generated as follows:

- *VM spawn:* CPU demand (% PP used) is aggregated over all PPs belonging to a customer at every time interval. We compute the average CPU load, under the assumption that load is equally distributed among the allocated VMs. Whenever a customer's average CPU load per VM exceeds a predefined high threshold and there are no free VMs available, one additional VM is spawned on a random host by cloning from an image template.

- *VM stop and start:* Whenever a customer's average CPU load drops below a predefined low threshold, one of its VMs is powered off to save resources (e.g. energy and memory). We assume that powered-off VMs are not reclaimed by the cloud. Customers may bring their VMs back by powering them on when the CPU demands become high later.

Our COPE prototype periodically (every 20 minutes) execute the COPElog program in Section 4.1, to perform a COP computation that orchestrates load balancing via VM migration within each data center. The program takes as input the `vm(Vid,CPU,Mem)` table, which is updated by the workload generator, as well as the `host(Hid)` table. Figure 2 shows the average CPU standard deviation of three data centers achieved by the COPElog program over a 3.5 hours period. During this period, there are on average 16.8 VM migrations every interval.
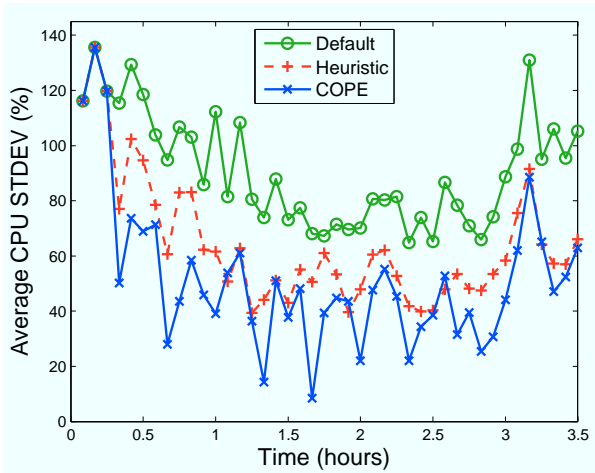
**Figure 2: Average CPU standard deviation of three data centers**

To validate that COPE is achieving good load balancing, we compare COPE against two strawman approaches *Default* and *Heuristic*. *Default* is the most naïve strategy, which simply does no migration after VMs are initially placed on a random host. *Heuristic* is a threshold-based policy that migrates VMs from the most loaded host (i.e. with the highest aggregate CPU of the VMs running on it) to the least one, until the most-to-least load ratio is below a threshold $K$ ($K$ is 1.2 in our experiment). *Heuristic* emulates an ad-hoc strategy that a cloud operator may adopt in the absence of COPE. As evidence of the effectiveness of COPE, we observe that COPE is able to more effectively perform load balancing, achieving a 44% and 18% reduction of the degree of CPU load imbalance as compared to *Default* and *Heuristic*, respectively.

Note that the above results validate the capability of COPElog in enforcing one particular load balancing policy. By customizing the COPElog rules, one could have flexibly implemented different policies that load balances memory usage, or achieve another optimization goal of minimizing the number of VM migrations, as previously described.

In terms of solver overhead, we note that on a Intel Quad core 2.33GHz PC with 4GB RAM running Ubuntu 10.04, the solver takes an average of 0.6 seconds (6.1 seconds at maximum) to complete each COP execution. The memory footprint is 217KB (on average), and 409KB (maximum). For larger-scale data centers with more migratable VMs, the solver will require exponentially more time to terminate. This makes it hard to reach the optimal solution in reasonable time. In the next section, we will discuss distributed deployment scenarios that can potentially improve scalability.

## 5. DISTRIBUTED ORCHESTRATION

As noted in Section 2, distributed deployments are required in cases where orchestration spans multiple cloud providers, or to improve scalability by partitioning an expensive COP operation into smaller operations (this typically results in an approximate solution). We use the *Follow-the-Sun* example to highlight these distributed capabilities.

## 5.1 COP Formulation

We first present a COP-based mathematical model of the Follow-the-Sun scenario. Recall that in the scenario, cloud providers decide to migrate VMs across data centers in order to consolidate resources and better meet customer demands. In this model, there are $n$ autonomous geographically distributed data centers $C_1, ..., C_n$ at location $1, 2, ..., n$. Each data center is managed by one COPE node. Each site $C_i$ has a resource capacity (set to the maximum number of VMs) denoted as $R_i$. Each customer specifies the number of VMs to be instantiated, as well as a preferred geographic location. We denote the aggregated resource demand at location $j$ as $D_j$, which is the sum of total number of VMs demanded by all customers at that location. Given the resource capacity and demand, $C_i$ currently allocates $A_{ji}$ resources (VMs) to meet customer demand $D_j$ at location $j$.

In the formulation, $M_{ijk}$ denotes the number of VMs migrated from $C_i$ to $C_j$ to meet $D_k$. When $M_{ijk} > 0$, the cloud orchestration layer will issue commands to migrate VMs accordingly. This can be periodically executed, or executed on demand whenever system parameters (e.g., demand $D$ or resource availability $R$) changes dramatically.

A naïve algorithm is to always migrate VMs to customers' preferred locations. However, it could be either impossible, when the aggregated resource demand exceeds resource capacity, or suboptimal, when the operating cost of a designated data center is much more expensive than neighboring ones, or when the traffic patterns incur high communication cost, e.g. the majority of the access clients for the VMs are far away from the preferred data center.

In contrast, COPE's COP approach attempts to optimize based on a number of factors captured in the cost function. In the model, we consider three main kinds of cost: (1) operating cost of data center $C_j$ is defined as $OC_j$, which includes typical recurring costs of operating a data center; (2) communication cost of meeting resource demand $D_i$ from data center $C_j$ is given as $CC_{ij}$; (3) migration cost $MC_{ij}$ is the communication overhead of moving a VM from $C_i$ to $C_j$. Given the above variables, the COP formulation is:

$$\min \quad (aggOC + aggCC + aggMC) \tag{1}$$

$$aggOC = \sum_{j=1}^{n} (\sum_{i=1}^{n} (A_{ij} + \sum_{k=1}^{n} M_{kji}) * OC_j) \tag{2}$$

$$aggCC = \sum_{j=1}^{n} \sum_{i=1}^{n} ((A_{ij} + \sum_{k=1}^{n} M_{kji}) * CC_{ij}) \tag{3}$$

$$aggMC = \sum_{i=1}^{n} \sum_{j=1}^{n} ((\sum_{k=1}^{n} max(M_{ijk}, 0)) * MC_{ij}) \tag{4}$$

subject to:

$$\forall j : R_j \geq \sum_{i=1}^{n} (A_{ij} + \sum_{k=1}^{n} M_{kji}) \tag{5}$$

$$\forall i, j, k : M_{ijk} + M_{jik} = 0 \tag{6}$$

**Optimization goal.** The COP aims to minimize the aggregate cost of cloud providers. In the above formula, it is defined as the sum of the aggregate operating cost $aggOC$ (2) across all data centers, aggregate communication cost $aggCC$ (3) to meet customer demands served at various data centers, and the aggregate VM migration cost $aggMC$ (4),

all of which are computed by summing up $OC_j$, $CC_{ij}$, and $MC_{ij}$ for the entire system.

**Constraints.** The COP is subjected to two constraints. In Constraint (5), each data center cannot allocate more resources than it possesses. Constraint (6) ensures the zero-sum relation between migrated resources between $C_i$ and $C_j$ for demand $k$.

In Appendix A, we validate the above COP formulation via performing a preliminary utility analysis using a synthetic workload. In the absence of actual traces that mirrors actual costs (operating, migration, communication), our synthetic workload provides an initial starting point for validating the formulation. Note that our COP formulation can be further customized based on provider/customer needs. For example, an alternative optimization goal is to minimize service latency in SLAs under the constraint that customer budget is fixed [27]. We can also impose restrictions on the maximum quantity of resources to be migrated due to high CPU load or router traffic in data centers, or impose constraints that the total cost after optimization should be smaller by a threshold than before optimization. This can prevent instability issues like recurring massive VM migration among specific data centers. Our model can also be extended to support other resource types (e.g. storage, network) beyond VM. We leave these extensions as future work.

## 5.2 COPElog Specifications

We next demonstrate how COPE can be used to realize a distributed implementation of the above model. At a high level, we utilize an iterative distributed graph-based computation strategy, in which all nodes execute a *local* COP, and then iteratively exchange COP results with neighboring nodes until a stopping condition is reached. In this execution model, all data centers are represented as nodes in a graph, and a link exists between two nodes if resources can be migrated across them. The following COPElog program implements the local COP at a given node X:

```
// goal declaration
goal minimize C in totalCost(@X,C).

// variable declaration
var migVM(@X,Y,D,R) forall eSetLink(@X,Y,D).

// totalCost definition
r1 totalCost(@X,SUM<C>) :- link(@X,Y),
                          aggCost(@Y,C).

// aggCost definition
r2 aggCost(@X,C) :- aggOpCost(@X,C1),
    aggCommCost(@X,C2), aggMigCost(@X,C3),
    C=C1+C2+C3.

// aggregate operating cost
r3 aggOpCost(@X,SUM<Cost>) :- curVM(@X,D,R1),
    migVM(@X,Y,D,R2), opCost(@X,C),
    Cost=C*(R1-R2).

// aggregate communication cost
r4 aggCommCost(@X,SUM<Cost>) :- curVM(@X,D,R1),
    migVM(@X,Y,D,R2), commCost(@X,Y,C),
    Cost=C*(R1-R2).

// aggregate migration cost
r5 aggMigCost(@X,SUM<Cost>) :- migVM(@X,Y,D,R),
    migCost(@X,Y,C), Cost=R*C, Cost>0.
```

| COP | COPElog |
|:---:|:---:|
| symbol $R_i$ | `resource(I,R)` |
| symbol $A_{ij}$ | `curVM(I,J,R)` |
| symbol $M_{ijk}$ | `migVM(I,J,K,R)` |
| equation (1) | rule `goal, r1-2` |
| equation (2) | rule `r3` |
| equation (3) | rule `r4` |
| equation (4) | rule `r5` |
| equation (5) | rule `r6-7`, constraint `c1` |
| equation (6) | rule `r8` |

**Table 1: Mappings from COP to COPElog.**

```
// constraints on VM migration quantity
r6 aggCurVM(@X,SUM<R>) :- curVM(@X,D,R).
r7 aggMigVM(@X,SUM<R>) :- migVM(@X,Y,D,R).
c1 aggMigVM(@X,R2) -> resource(@X,R),
       aggCurVM(@X,R1), R1-R2>=R.

// propagate results to ensure symmetry
r8 migVM(@Y,X,D,R2):- migVM(@X,Y,D,R1), R2=-R1.
```

The above program is written using distributed variant of Datalog used in declarative networking [17], where the location specifier `@` denotes the source location of each corresponding tuple. This allows us to write rules where the input data spans multiple nodes, a convenient language construct for formulating distributed optimizations. Table 1 summarizes the mapping from COP symbols to COPElog tables, and COP equations to COPElog rules/constraints identified by the rule labels. In the table, $R_i$ is stored as a `resource(I,R)` tuple. Likewise, the R attribute in `mig(I,J,K,R)` stores the value of $M_{ijk}$.

The localized COP program works as follows. Instead of minimizing the global total cost of all data centers, the optimization goal is the total cost within a local region (node X and its neighbors). Periodically, an `eSetLink` event triggers the local COP operation at data center X, which then randomly selects one of its links to start a *link negotiation* process with its neighbor Y for resource demand at D. The `eSetLink(@X,Y,D)` tuple is generated periodically at node X via an extra COPElog rule, where Y and D denotes the neighbor randomly chosen for the current negotiation process and the resource demand location, respectively. To avoid conflicts, for any given `link(X,Y)`, the link negotiation protocol selects the node with the larger identifier (or address) to carry out the subsequent process.

As part of this negotiation process, each node exchanges its aggregate system state `aggCost` (derived using rules `r2-r5`) with its neighboring nodes. This is achieved by using rule `r1`, where node X collects its neighbors' `aggCost` tables, and then perform a `SUM` aggregate, stored in `totalCost`. The negotiation process then solves a *local* COP (by minimizing `totalCost`) to determine the quantity of resources that can be migrated to the specific neighbor. The COP is expressed by solving the optimization goal, under the specified constraint `c1`, together with rules `r6` and `r7`. The migration result `migVM` is then propagated to immediate neighbors using rule `r8` to ensure symmetry. Above process is then iteratively repeated until all links have been assigned values or when a stopping condition is reached.

**Distributed Execution.** The COPElog compilation process (in Section 4.2) requires minimal modification in or-

der to implement the above distributed program. COPE uses RapidNet for executing distributed Datalog rules, which already provides a runtime environment for implementing these rules. At a high level, each distributed rule or constraint (with multiple distinct location specifiers) is rewritten using a *localization* rewrite [17] step. This transformation results in rule bodies that can be executed locally, and rule heads that can be derived and sent across nodes. The beauty of this rewrite is that even if the original program expresses distributed properties and constraints, this rewrite process will realize multiple local COP operations at different nodes, and have the output of COP operations via derivations sent across nodes. For example, rule `r8` takes the output of a local COP execution at node `X`, and sends that to node `Y` to continue on the optimization process in the next iteration.

## 6. RELATED WORK

Prior to COPE, there have been a variety of systems that use declarative logic-based policy languages to express constraint optimization problems in resource management of computing systems. [23] proposes continuous optimization based on declaratively specified policies for autonomic computing. [21] describes a model for automated policy-based construction as a goal satisfaction problem in utility computing environments. The XSB engine [3] integrates a tabled Prolog engine with a constraint solver. Rhizoma [26] proposes to use rule-based language and constraint solving programming to optimize resource allocation. It focuses on self-deployed applications in overlay networks to maximize the utilization of well-connected, lightly loaded nodes. [19] uses a logic-based interface to a SAT solver to automatically generate configuration solution for a single data center.

Unlike the above systems, COPE is designed for a different application domain – one that aims to achieve cost reduction and maximize infrastructure utilization by automatically orchestrating the cloud based on policies configured by cloud providers. Another unique feature of COPE is its support for *distributed optimizations*, achieved by using the COPElog language which supports distribution, and the integration of a distributed query engine with a constraint solver.

## 7. ONGOING STATUS

This paper presents our initial work on the COPE system, which is a first step towards our eventual vision of enabling automated cloud resource orchestration realized and controlled using declarative languages. Resource allocation in the cloud is an inherent distributed optimization problem. We have demonstrated in this paper the viability of COPE in the centralized optimization scenario, using trace-driven simulations obtained from an actual hosting service. Realizing COPE in its full entirety requires a distributed policy language and optimization environment. Towards that goal, we introduce the COPElog framework, which combines unique features from declarative networking and constraint solving.

We are currently in the process of developing a full-fledged COPE prototype that will be integrated with the DMF cloud orchestration platform. We intend to evaluate the Simple-Cloud and Follow-the-Sun scenarios on the Shadownet [8] testbed. At this point of time, we have already successfully deployed DMF on Shadownet for two scenarios. The Simple-Cloud scenario consists of about 1000 VMs on 15 Shadownet hosts, while the Follow-the-Sun scenario is a smaller-scale limited deployment. A live demonstration [15] was carried out recently.

Building upon the basic policies described in this paper, we hope to incorporate into the optimization framework, recent models on resource provisioning and deployment analysis in the cloud. For example, [20] proposes a server operational cost optimization model for cloud computing service that involves scaling CPU frequency and powering on/off VMs over a time horizon. [13, 14] proposes a pricing model that offers cloud users the choice of different execution speeds corresponding to different prices. Similarly, [12] explores the feasibility of migrating enterprise applications to the cloud, based on a cost-benefit analysis. We plan to explore the applicability of COPE to enable such resource provisioning and cost-benefit analysis.

We also plan to release the COPE prototype as open-source, for use by other researchers to implement resource management policies in the cloud. Finally, we will explore more complex scenarios beyond resource migration, particularly those that involve network management tasks [7] incurred as a result of cloud orchestration, and orchestration procedures required for mobility [9], virtualized desktop [11] and database consolidation [10].

## 8. ACKNOWLEDGEMENTS

## 9. REFERENCES

[1] Gecode constraint development environment. http://www.gecode.org/.

[2] RapidNet. http://netdb.cis.upenn.edu/rapidnet/.

[3] XSB Project. http://xsb.sourceforge.net/.

[4] AGARWAL, S., DUNAGAN, J., JAIN, N., SAROIU, S., WOLMAN, A., AND BHOGAN, H. Volley: automated data placement for geo-distributed cloud services. In *NSDI* (2010).

[5] AL-FARES, M., RADHAKRISHNAN, S., RAGHAVAN, B., HUANG, N., AND VAHDAT, A. Hedera: dynamic flow scheduling for data center networks. In *NSDI* (2010).

[6] CAMPBELL, R., GUPTA, I., HEATH, M., KO, S. Y., KOZUCH, M., KUNZE, M., KWAN, T., LAI, K., LEE, H. Y., LYONS, M., MILOJICIC, D., O'HALLARON, D., AND SOH, Y. C. Open cirrus cloud computing testbed: federated data centers for open source systems and services research. In *HotCloud* (2009).

[7] CHEN, X., MAO, Y., MAO, Z. M., AND VAN DER MERWE, J. Declarative Configuration Management for Complex and Dynamic Networks. In *CoNEXT* (2010).

[8] CHEN, X., MAO, Z. M., AND VAN DER MERWE, J. ShadowNet: A Platform for Rapid and Safe Network Evolution. In *Proc. USENIX ATC* (2009).

[9] CHUN, B.-G., IHM, S., MANIATIS, P., NAIK, M., AND PATTI, A. Clonecloud: Elastic execution between mobile device and cloud. In *EuroSys* (2011).

[10] CURINO, C., JONES, E., MADDEN, S., AND BALAKRISHNAN, H. Workload-aware database monitoring and consolidation. In *SIGMOD* (2011).

[11] DAS, T., PADALA, P., PADMANABHAN, V. N., RAMJEE, R., AND SHIN, K. G. Litegreen: saving energy in networked desktops using virtualization. In *USENIX ATC* (2010).

[12] HAJJAT, M., SUN, X., SUNG, Y.-W. E., MALTZ, D., RAO, S., SRIPANIDKULCHAI, K., AND TAWARMALANI, M. Cloudward bound: planning for beneficial migration of enterprise applications to the cloud. In *SIGCOMM* (2010).

[13] HENZINGER, T. A., SINGH, A. V., SINGH, V., WIES, T., AND ZUFFEREY, D. FlexPRICE: Flexible provisioning of resources in a cloud environment. In *IEEE Conference on Cloud Computing* (2010).

[14] Henzinger, T. A., Singh, A. V., Singh, V., Wies, T., and Zufferey, D. A marketplace for cloud resources. In *EMSOFT* (2010).

[15] Liu, C., Mao, Y., Chen, X., Fernandez, M. F., Loo, B. T., and der Merwe, K. V. Towards transactional cloud resource orchestration. In *NSDI (poster and demo)* (2011).

[16] Liu, C., Mao, Y., Van der Merwe, J., and Fernandez, M. Cloud Resource Orchestration: A Data-Centric Approach. In *CIDR* (2011).

[17] Loo, B. T., Condie, T., Garofalakis, M., Gay, D. E., Hellerstein, J. M., Maniatis, P., Ramakrishnan, R., Roscoe, T., and Stoica, I. Declarative Networking. In *Communications of the ACM (CACM)* (2009).

[18] Mengmeng Liu, Nicholas Taylor, Wenchao Zhou, Zachary Ives, and Boon Thau Loo. Recursive Computation of Regions and Connectivity in Networks. In *ICDE* (2009).

[19] Narain, S., Levin, G., Malik, S., and Kaul, V. Declarative infrastructure configuration synthesis and debugging. *J. Netw. Syst. Manage. 16* (September 2008), 235–258.

[20] Qian, H., and Medhi, D. Server operational cost optimization for cloud computing service providers over a time horizon. In *HotICE* (2011).

[21] Sahai, A., Singhal, S., Machiraju, V., and Joshi, R. Automated policy-based resource construction in utility computing environments. In *IEEE/IFIP Network Operations and Management Symposium* (2004).

[22] Van der Merwe, J., Ramakrishnan, K., Fairchild, M., Flavel, A., Houle, J., Lagar-Cavilla, H. A., and Mulligan, J. Towards a ubiquitous cloud computing infrastructure. In *LANMAN* (2010).

[23] White, S. R., Hanson, J. E., Whalley, I., Chess, D. M., Segal, A., and Kephart, J. O. Autonomic computing: Architectural approach and prototype. *Integr. Comput.-Aided Eng. 13* (April 2006), 173–188.

[24] Wood, T., Gerber, A., Ramakrishnan, K., Shenoy, P., and der Merwe, J. V. The case for enterprise-ready virtual private clouds. In *HotCloud* (2009).

[25] Wood, T., Shenoy, P., Venkataramani, A., and Yousif, M. Black-box and gray-box strategies for virtual machine migration. In *NSDI* (2007).

[26] Yin, Q., Schuepbach, A., Cappos, J., Baumann, A., and Roscoe, T. Rhizoma: a runtime for self-deploying, self-managing overlays. In *Proceedings of Middleware* (2009).

[27] Zhang, Z., Zhang, M., Greenberg, A., Hu, Y. C., Mahajan, R., and Christian, B. Optimizing cost and performance in online service provider networks. In *NSDI* (2010).

# APPENDIX

# A. FOLLOW-THE-SUN SIMULATIONS

Using the Follow-the-Sun scenario, we present a preliminary validation of our COP formulation. The goal of our evaluation is to validate that executing the COP results in VM placement which minimizes the total cost (as described in Section 5.1). We also examine the scalability of COPE.

Our experiment setup consists of 3 data centers geographically distributed at three locations, with the number of migratable VMs all set to 60. Initially each data center has 35, 35, and 30 VMs respectively. This roughly corresponds to a random placement of VMs across the 3 data centers.

We consider three time phases within a single day in which there are drastic changes in user locations, i.e. morning, afternoon, and evening, denoted as $time = 0, 1, 2$. Given an aggregate customer demand of 100, we modify customer locations as follows. At $time = 0$, all customer are located near data center $DC_0$. This means that demand at $DC_0$ is $D_0 = 100$. As the Sun moves, at $time = 1$, customers move closer to $DC_1$, hence $D_1 = 100$. Finally, at $time = 2$, demand shifts again to $DC_2$.

Based on the above customer mobility pattern, at the beginning of each time phase, we execute the COP that determines the VM migrations necessary to minimize the cloud providers' total cost. As a strawman, we compare against

*Default*, where all the VMs continue to reside in their original location regardless of time phase. In our analysis, we consider abstract cost values as follows. We set the operating cost for all data center to be 4. The migration cost between data centers are $MC = \begin{bmatrix} 0 & 1 & 1 \\ 1 & 0 & 1 \\ 1 & 1 & 0 \end{bmatrix}$, where the value in row $i$ and column $j$ denote $MC_{ij}$. Finally, $CC = \begin{bmatrix} 0 & 3 & 5 \\ 3 & 0 & 4 \\ 5 & 4 & 0 \end{bmatrix}$. Given that data centers may span geographic regions, communication costs between data centers may differ.

Table 2 summarizes our evaluation results, where we compare the total cost of *COPE* against the strawman *Default* approach. We further provide a breakdown of VMs per data center at each time interval.

| | Time | 0 | 1 | 2 |
|---|---|---|---|---|
| **Default** | $DC_0$ | 35 | 60 | 40 |
| | $DC_1$ | 35 | 40 | 60 |
| | $DC_2$ | 30 | 0 | 0 |
| | **Total Cost** | 655 | 580 | 840 |
| **COPE** | $DC_0$ | 60 | 40 | 0 |
| | $DC_1$ | 40 | 60 | 40 |
| | $DC_2$ | 0 | 0 | 60 |
| | **Total Cost** | 548 | 540 | 618 |

**Table 2: Evaluation results of the Follow-the-Sun scenario.**

We observe that the output from our solver produced the following migration patterns: at $time = 0$, 25 VMs are migrated from $DC_2$ to $DC_0$, and 5 VMs from $DC_2$ to $DC_1$; at $time = 1$, 20 VMs are migrated from $DC_0$ to $DC_1$; and at $time = 2$, 1 VM is migrate from $DC_0$ to $DC_1$, 39 VMs are migrated from $DC_0$ to $DC_2$, and 21 VMs from $DC_1$ to $DC_2$. While the absolute number of VMs migrated are highly dependent on cost parameters, we note that the general trend of shifting VMs based on demand is followed. Based on this migration pattern, we observe that *COPE* achieved a lower total cost compared to *Default*.

In terms of solver execution time, we note that on a Intel Quad core 2.33GHz PC with 4GB RAM running Ubuntu 10.04, the solver takes less than 2.6 seconds to complete each COP execution. For larger number of data centers up to 6, the solver returns the answer within seconds.