Ostro: Scalable Placement Optimization of Complex Application Topologies in Large-Scale Data Centers

Gueyoung Jung, Matti Hiltunen, Kaustubh Joshi, Rajesh Panta, Richard Schlichting
AT&T Labs - Research
1 AT&T Way, Bedminster, New Jersey, USA
{gjung, hiltunen, kaustubh, rpanta, rick}@research.att.com

***Abstract*—A complex cloud application consists of virtual machines (VMs) running software such as web servers and load balancers, storage in the form of disk volumes, and network connections that enable communication between VMs and between VMs and disk volumes. The application is also associated with various requirements, including not only quantities such as the sizes of the VMs and disk volumes, but also quality of service (QoS) attributes such as throughput, latency, and reliability. This paper presents Ostro, an OpenStack-based scheduler that optimizes the utilization of data center resources, while satisfying the requirements of the cloud applications. The novelty of the approach realized by Ostro is that it makes holistic placement decisions, in which all the requirements of an application—described using an *application topology* abstraction—are considered jointly. Specific placement algorithms for application topologies are described including an estimate-based greedy algorithm and a time-bounded A* algorithm. These algorithms can deal with complex topologies that have heterogeneous resource requirements, while still being scalable enough to handle the placement of hundreds of VMs and volumes across several thousands of host servers. The approach is evaluated using both extensive simulations and realistic experiments. These results show that Ostro significantly improves resource utilization when compared with naive approaches.**

***Index Terms*—cloud; optimization; performance; scalability**

I. Introduction

A cloud application forms a logical topology that consists of multiple virtual machines (VMs) and disk volumes, together with the network links that interconnect them. Such applications are typically deployed as multiple loosely-coupled components that provide separate yet interdependent functions [1]. Instances of components then execute as VMs generating I/O to/from disk volumes, and communicate with other instances across a network. For example, Virtual Network Functions (VNFs) often consist of a number of network functions such as firewalls, routers, and CDN caches that are virtualized and interconnected into a logical topology [2].

In addition to a logical layout, a cloud application has *properties* associated with it. These include quantities such as the size of a VM or a disk volume, together with *quality of service* (QoS) attributes such as throughput, latency, and reliability [3][4][5]. Properties can also include requirements such as specific hardware or software affinities for VMs and disk volumes, and anti-affinities among VMs and disk volumes for better application reliability. We call the combination of an application's logical layout and properties the *application topology* of that application.

A critical aspect of cloud operation is the placement of applications onto the physical resources of a cloud infrastructure. This process is fundamentally a resource scheduling problem, and requires balancing two objectives: respecting the requirements represented by an application topology and maximizing resource utilization. The former is needed to provide appropriate service to applications, while the latter is needed to make efficient use of costly physical resources such as CPUs, disks, and networks. The placement decision may arise not just at application deployment time, but also at runtime if the infrastructure is being managed adaptively and the resource assignments to applications can be changed. These dynamic effects, the large scale of contemporary cloud infrastructures, and the need to handle the varying requirements of multiple applications simultaneously combine to make this a technically challenging multi-dimensional optimization problem.

This paper presents Ostro, a scalable OpenStack-based scheduler for cloud applications that addresses these challenges in the context of large-scale data centers. The main contribution of Ostro is making *holistic placement decisions*, i.e., how it considers each application topology as an "indivisible" unit and allocates all of the resources needed to execute the application. The goal is to maximize resource utilization across compute, storage, and network, while still meeting the QoS requirements and other properties associated with the application topology. This is in contrast with many current resource schedulers such as the one used in OpenStack, which treats each VM or disk volume request independently [6]. It also goes beyond placement algorithms in the literature such as [3][4][7][8], which focus only on network dependencies between application components or have restrictions such as uniformity of network flows or a single VM per host server.

In addition to the architectural details of Ostro, we describe the three placement algorithms currently supported by the scheduler. The first is an estimate-based greedy algorithm, the second is an algorithm based on A* graph search [9][10][11] that allows the search space to be extended efficiently, and the third is a deadline-bounded search algorithm that is designed to make a placement decision within a specified time bound. All three are evaluated using both extensive simulation and experiments with a realistic cloud storage application that contains multiple application components using a number of VMs and disk volumes. The current implementation of Ostro focuses on placement within a data center, but it can serve

as the basis for placement across multiple data centers in the wide area as well.

II. System Overview

We have integrated Ostro with OpenStack (www.openstack.org), a popular open-source cloud operating system consisting of a collection of services such as Nova (compute), Cinder (block storage), Neutron (networking), and Heat (orchestration). A cloud tenant can create a cloud application by invoking Nova and Cinder for each VM and disk volume the application requires. Nova and Cinder handle each request in isolation and use their own scheduling algorithms to decide placements individually. As a result, their placements are not optimized for the application, and any other QoS attributes are not considered.

In contrast, an application topology allows the cloud tenant to specify the elements and requirements of the entire application (see Figure 2). Specifically, it allows the specification of the VMs, disk volumes, and the network throughput between any two VMs and between a VM and a disk volume. An application topology also allows the specification of placement diversity requirements for a set of VMs or disk volumes with a *diversity zone* concept (i.e., anti-affinity). For example, using diversity zones a tenant can specify that 10 VMs running redundant database servers must be deployed across 10 different racks, or that 12 disk volumes must be placed on 12 separate disks for better reliability.

Ostro takes the application topology as input, and uses information about the available resources in the data center to determine the placement of all VMs and disk volumes of the topology so that all the requirements are met while optimizing resource utilization.

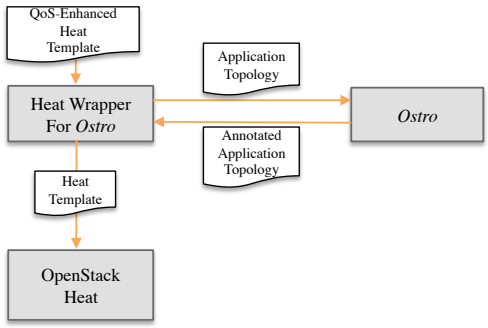


Fig. 1: Ostro integration with OpenStack

Figure 1 outlines the placement process of Ostro with OpenStack. In the current implementation, the application topology is described using a Heat template extended with diversity zones and a network pipe concept that specifies the VM to VM and VM to disk volume bandwidth requirements (*QoS-enhanced Heat template*). We provide a wrapper for the OpenStack Heat service. The wrapper takes the Heat template, and passes it to Ostro for placement optimization. Ostro returns the placement decision indicating for each VM and volume on which exact host and disk it should be deployed.

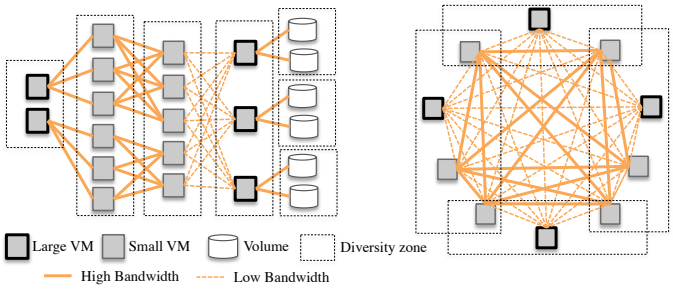


Fig. 2: Example application topologies: multi-tier (left) and mesh communication (right)

The modified Heat template is then passed to the Heat engine that calls Nova and Cinder to schedule the VMs and disk volumes on the designated cloud resources.

A. Search Space for Optimal Placement Problem

Ostro optimizes the placement of a given application topology on a set of physical resources in a data center. We start by describing some of the specific challenges regarding the application topology and the data center.

1) Application Topology: Cloud applications may have large and complex topologies. The multi-tier [12][7] and mesh communication [3] topologies in Figure 2 are such examples of complex enterprise applications. Although Ostro can, in fact, handle any arbitrary application topology, the two topologies in the figure are used in our simulation with different scales to evaluate scalability (Section IV). The resource requirements associated with a given application topology such as those in Figure 2 typically vary across components and network links, i.e., are *heterogeneous* resource requirements. For example, VMs containing web servers of a three-tier application are mainly network intensive, while VMs containing database servers tend to be mainly compute intensive. This complicates the problem, and naive greedy approaches are insufficient. Ostro considers the co-existence of both compute- and network-intensivenesses in a single application topology.

Formally, the application topology is defined as a graph $T^a =< V, E >$, where a node $v_i \in V$ is a VM or a disk volume, and an edge $e_{i,j} \in E$ is a communication link between two nodes v_i and v_j. Each v_i and $e_{i,j}$ have property attributes defining resource requirements such as the number of vCPUs and the network bandwidth.

2) Data Center Topology: Analogously, the operating conditions of a data center can vary widely in terms of how much resources are available at each host at any given time. One possible scenario is that the available remaining compute and network capacities are spread non-uniformly across the hosts. This would be the case, for instance, if the compute resources of some hosts have enough capacity to accommodate multiple VMs, but their remaining network bandwidth can only accommodate a few, while the rest of the hosts are in the opposite situation. Conversely, a homogeneous condition would arise if all hosts and network links have sufficient and evenly available resources for a new application. This would allow multiple ways for placing the application onto

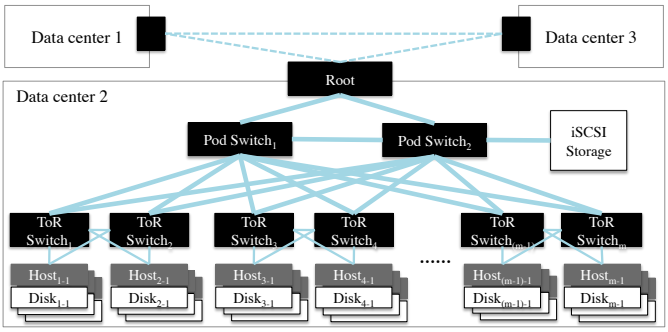


Fig. 3: Hierarchical data center

the available hosts, and each solution would result in a similar resource utilization. As might be expected, less uniformity makes it more difficult to solve the placement problem.

In this paper, we consider a hierarchical data center T^p that is common in modern data centers [3][12][4][5][8][7]. As illustrated in Figure 3, in T^p, each host $h_k \in H$ is placed in a rack with a top of the rack (ToR) switch, racks are grouped under a pod switch, and pods are connected to a root switch. A communication path between v_i and v_j may involve all three levels of switches and consume network bandwidth on all three levels of the network hierarchy depending on the placement of v_i and v_j. Note that Ostro is not limited to hierarchical network topologies, but accounts for any graphical topology representing multiple connected data centers.

B. Objective Function and Constraints

Given application topology T^a and data center T^p, the placement problem can be formulated as a constraint optimization problem. Here, we formally define the objective function and constraints.

1) Objective Function: Although any objective function can be defined, in this paper, we focus on minimizing the amount of bandwidth, u^{bw} to be used for the VMs and disk volumes of a given T^a, and the total number of hosts, u^c, after placing T^a. Formally, it can be defined as follows:

$$min(\theta^{bw}\frac{u^{bw}}{\tilde{u}^{bw}} + \theta^{c}\frac{u^{c}}{\tilde{u}^{c}})$$

where $\theta^{bw} + \theta^{c} = 1$. To unify two different usages, u^{bw} and u^c can be normalized against the corresponding worst case placements (i.e., $\tilde{u}^{bw}$ and $\tilde{u}^c$, respectively).

For the minimal u^{bw}, we place nodes of T^a as close as possible in T^p once all constraints are met. Meanwhile, to minimize u^c, we place as many nodes of T^a as possible to hosts that already contain existing nodes of this or other applications (i.e., they are not idle). By doing so, we may increase u^{bw} of T^a, since nodes of T^a may end up being more spread out across those hosts.

2) Constraints: In this paper, we consider two different types of constraints when solving the problem.

Capacity Constraints. Ostro accounts for capacity constraints of CPU, memory, disk space, and network bandwidth. Formally, all possible candidates hosts $\mathcal{H}_i \in H$ for v_i, must meet the inequality constraints defined as, $r^{cpu} \leq c^{cpu}$, $r^{mem} \leq c^{mem}$, and $r^{vol} \leq c^{disk}$, where r^x and c^x denote the resource requirements of v_i and the available capacity of each candidate $h_k \in \mathcal{H}_i$ for the resource type x, respectively. For network bandwidth, all links of the path between two hosts must have enough capacity. Consider the case where v_i communicates with v_j, which is placed in another location. Then, it must meet $r^{bw} \leq min\{c_k^{bw}, c_{tor}^{bw}, c_{pod}^{bw}, c_{root}^{bw}\}$. Here, c^{bw} is the available bandwidths of h_k or each switch (i.e., ToR, pod, or root) connected along the path to the data center where v_i is placed.

Diversity Zone. The diversity zone constraint defines how to segregate nodes of each application components (e.g., database replicas) when placing them (see the dashed gray line in Figure 2). We define d_k^z for a set of nodes, $\mathcal{V}_k \in V$ that must be placed at least across different z, where z can be host, rack, pod, or data center. Note that a v_i may belong to multiple diversity zones. We assume that d_k^z can be computed from the high-level reliability input provided by a cloud user (e.g., the reliability should be 99.99%) as described in our prior work [13].

C. Scalability

We consider the scalability of our approach. Due to the large search space that is combinatorial between nodes of T^a and hosts of T^p, the running time t can be very large, particularly when dealing with heterogeneous resource requirements and non-uniform resource availability. Hence, in such situations, we need to address the trade-off between t and the optimality of the solution by effectively pruning and bounding the large search space. Therefore, Ostro provides a time-driven optimization that computes an optimized solution by a given deadline $\mathcal{T}$ by controlling the pruning rate and bounding of the search. By allowing a longer search deadline, Ostro can generate a better solution.

III. Optimal Placement

Placing a graph topology onto a hierarchical topology with bandwidth constrained edges is a known NP-hard problem [14][4][5]. Hence, we have developed three heuristic approaches to address the tradeoff between scalability and optimality even under heterogeneous resource requirements and non-uniform resource availability.

A. Estimate-Based Greedy Search

We first describe our estimate-based greedy approach (EG) to efficiently solve the problem even under non-uniform resource availability in data centers.

1) Overall Approach: A typical greedy search approach can compute a placement by taking one node v at a time from the list V and placing it on the first h where it fits. This approach can find a solution quickly but it requires that hosts of T^p (and nodes of T^a) are sorted in an appropriate way. If there is one bottleneck resource that is consumed consistently by all nodes, it is trivial to sort them based on the usage of this resource. For example, if all VMs of T^a are mainly network

Algorithm 1 Estimate-based Greedy Algorithm (EG)

1: $H^* \leftarrow \emptyset$;
2: `Sort`(V);
3: **while** $V \neq \emptyset$ **do**
4: $v_i \leftarrow V$.`Pop`();
5: $\mathcal{H}_i \leftarrow$ `GetCandidates`$(v_i,\ H)$;
6: **for** each $h_j \in \mathcal{H}_i$ **do**
7: $(u^{bw*},\ u^{c*}) \leftarrow$ `GetUsage`$(v_i,\ h_j,\ H^*)$
8: $(\hat{u}^{bw},\ \hat{u}^{c}) \leftarrow$ `GetHeuristic`$(v_i,\ h_j,\ H,\ H^*)$;
9: $u^{bw} \leftarrow (u^{bw*} + \hat{u}^{bw})$; $u^{c} \leftarrow (u^{c*} + \hat{u}^{c})$;
10: **end for**
11: $h^{best} \leftarrow$ `GetBest`$(v_i,\ \mathcal{H}_i)$;
12: $H^* \leftarrow H^* \cup (h^{best} \leftarrow v_i)$;
13: **end while**

intensive, hosts can be sorted by available network bandwidth. However, it may be more common that different VMs of a single T^a consume different resource types at different rates. Hence, there is no single deciding resource type to be used for sorting nodes.

In EG, shown in Algorithm 1, nodes are simply sorted by the sum of relative weights of resource types (i.e., $\sum_{x=cpu,mem,disk,bw}(r^x/\mathcal{R}^x)$, where $\mathcal{R}^x$ is the average total requirement of resource type x across all VMs and disk volumes) in `Sort` (line 2 of Algorithm 1). `GetCandidates()` (line 5) then chooses a list of candidate hosts $\mathcal{H}_i$ for the given v_i from all hosts H by applying constraints described in Section II-B2. We use $v^* \in V^*$ and $h^* \in H^*$ to denote a v of T^a that has been already placed and a host h, on which the v has been placed in the search, respectively. For the diversity constraint d_k^z, it computes the topological distance (e.g., the same rack or the same pod group) between each candidate host h_j and each h^* in H^* to check if the diversity constraint can be met (e.g., if d^{host}, h_j and h^* must be separated in at least different hosts). For the bandwidth constraint, it chooses all v^*'s in V^* that are connected to the given v_i, and then checks if all links between the candidate h_j and all hosts h^*'s, where those connected v^*'s are placed, have enough capacity.

Instead of sorting hosts, EG explores all available hosts and selects a h^{best} out of candidate hosts $\mathcal{H}_i$. The decision-making for h^{best} is based on the objective function (see Section II-B1), that consists of two utility values. The first is the accumulated resource usage (denoted by u^{bw*} and u^{c*}) resulting from the placement of nodes from the first v_0 to the previously placed node v_{i-1} in the sorted list V (i.e., V^*). The second is the heuristic lower bound of utility incurred by placing the nodes from the current v_i to the last node in V. We denote the lower bound of utility and the remaining nodes by $\hat{u}^{bw}$, $\hat{u}^{c}$, and $\hat{V}$, respectively.

2) Heuristic Lower Bound: `GetHeuristic()` (line 8 of Algorithm 1) estimates the lower bound of $\hat{u}^{bw}$ and $\hat{u}^{c}$. This can reduce the impact of sorting nodes (especially when completely consistent sorting is infeasible) on the placement of EG by approximately placing the remaining nodes in $\hat{V}$ and comparing it with other choices of h_j under given v_i. First, it places v_i onto the given candidate host h_j. Then, it sorts all nodes in $\hat{V}$ by the bandwidth requirement, and tries to place each $\hat{v} \in \hat{V}$ either any $h^* \in H^*$ or any imaginary host (denoted as $\hat{h} \in \hat{H}$). If the following conditions meet, it creates a new $\hat{h}$ and places the $\hat{v}$ on it: 1) if the capacity constraints of H^* and $\hat{H}$ are not enough, 2) if d^z's between the $\hat{v}$ and all nodes placed in H^* and $\hat{H}$ are violated, 3) if the $\hat{v}$ has no link to any node in H^* and $\hat{H}$, and 4) if the $\hat{v}$ has more bandwidth requirement with the rest of nodes in $\hat{V}$ than those in H^* and $\hat{H}$. With this way, the $\hat{v}$ is co-located with nodes that are linked with more bandwidth. To achieve the lower bound of utility, the $\hat{h}$ should have enough capacity (in our implementation, the capacity of $\hat{h}$ for each resource type is the max value of the resource type among all h's), and the imaginary hosts are not counted to $\hat{u}^c$.

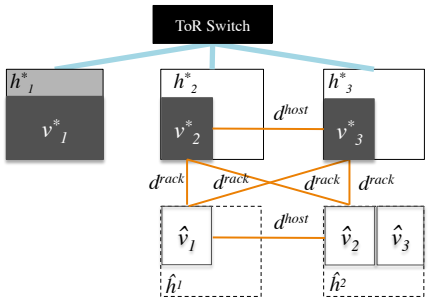


Fig. 4: Process of placing nodes

Figure 4 illustrates an example placement procedure that places v_1^*, v_2^*, v_3^*, $\hat{v}_1$, $\hat{v}_2$, and $\hat{v}_3$ (in this order). Three nodes (v_1^*, v_2^*, and v_3^*, all in V^*) have already been placed in h_1^*, h_2^*, and h_3^* (all in H^*). $\hat{v}_1$, $\hat{v}_2$, and $\hat{v}_3$ are the remaining nodes in $\hat{V}$ that will be approximately placed. $\hat{v}_1$ cannot be placed in h_1^* due to the lack of capacity. It cannot be placed in h_2^* and h_3^* due to the violation of diversity zone (i.e., $\hat{v}_1$ must be in a different rack from the rack where v_2^* and v_3^* are placed). Thus, an imaginary host $\hat{h}_1$ is created to place $\hat{v}_1$. With the same reason, $\hat{v}_2$ cannot be placed in any host in H^*. $\hat{v}_2$ cannot be placed in $\hat{h}_1$ as well due to the violation of the diversity zone with $\hat{v}_1$ (i.e., $\hat{v}_2$ must be in a host separated from $\hat{v}_1$) so that $\hat{v}_2$ is placed in another new imaginary host $\hat{h}_2$. Finally, $\hat{h}_2$ is chosen for $\hat{v}_3$ out of 4 candidate hosts (i.e., h_2^*, h_3^*, $\hat{h}_1$, and $\hat{h}_2$) since $\hat{v}_3$ is linked to $\hat{v}_2$ with the largest bandwidth requirement.

For each candidate placement, the total u^{bw} is estimated by the following function $f(v_i, h_j)$:

$$\sum_{v \in V} \sum_{v' \in V} (u'^{bw} \mid max\{d^z(h \leftarrow v, h' \leftarrow v'), h \neq h'\})$$
$$,\ for\ \forall (v, v')\ connected\ in\ T^a, \forall (h, h') \in (H^* \cup \hat{H})$$

u'^{bw} increases as hosts h and h' (where v and v' are placed, respectively) are separated further (e.g., z of diversity zone d^z increases from host to data center). u^c is computed by counting hosts used for getting u^{bw}. Finally, EG computes h^{best} (`GetBest()`, line 11) by,

$$\arg\min_{h_j \in \mathcal{H}} f(v_i, h_j)$$

Algorithm 2 Bounded A* Algorithm (BA*)

```
1: OQ ← OQ ∪ (V_0, H*_0, u_0);
2: CQ ← ∅; u^max ← 0;
3: (H*^upper, u^upper) ← RunEG();
4: while forever do
5:    (V_p, H*_p, u_p) ← OQ.Pop();
6:    if u_p ≥ u^upper then return H*^upper;
7:    if V_p = ∅ then return H*_p;
8:    P ← CreateCandidatePaths(V_p, H*_p, u_p);
9:    for each (V_q, H*_q, u_q) ∈ P do
10:      if ∃H'* ∈ CQ s.t. H*_q = H'* then continue;
11:      if u_q ≥ u^upper then continue;
12:      OQ.Insert(V_q, H*_q, u_q);
13:   end for
14:   CQ ← CQ ∪ (V_p, H*_p, u_p);
15:   if u_p > u^max then
16:      u^max ← u_p;
17:      (H*^upper, u^upper) ← RunEG();
18:   end if
19: end while
```

Note that this method estimates the $(u^* + \hat{u})$ as the achievable lower bound of u, and will be used as an "admissible" heuristic in the A* approach described in the next section.

The complexity of EG is polynomial (i.e., $O(|V|^3|H|)$, and to make the approach more efficient, EG computes the utility (lines 6–10) in parallel.

B. A* for Extended Search Space

Although EG can efficiently identify the placement through a single search path (i.e., linearly identify h^* for each v one by one), the results are less effective, especially in heterogeneous conditions (e.g., some are compute intensive, while others are network intensive) because the sorting may be infeasible. To resolve it, we have developed a bounded A* algorithm (BA^*, shown in Algorithm 2), that requires sorting of neither the hosts nor the nodes of T^a.

1) Overall Approach: Basically, BA^* explores all possible search paths (while EG searches along a single search path). Each individual path p includes all variables required for search (utilization u_p, placements H^*_p, and nodes V_p) that are used for the single search path. New paths are branched out by making all possible combinations of the current v_i in V_p and all candidate hosts (line 8 of Algorithm 2). These new paths are inserted into the open queue OQ and sorted by u (line 12). The new search starts with the path that has the least u in OQ (line 5). BA^* finishes the search once all nodes of currently selected V_p are completely placed (line 7). Otherwise, it adds the path into the closed queue CQ (line 14) and keep searching. Once it finds the best path (line 6 or 7), it can safely ignore the rest of the unfinished paths in OQ. This is because it uses the "admissible" heuristic (GetHeuristic() as described in the previous section) when creating new paths. That is, utility values of unfinished paths will be always equal or greater than u of the best path found.

2) Bounding the Search: BA^* can reduce the search space by pruning non-optimal paths, which utility values are even more than u of EG. Once it captures that the search is advanced by checking the current maximum u^{max}, it runs EG to compute the new upper-bound u^{upper} (line 3 and 17) and then, bounds the search by the u^{upper} (line 11). u^{upper} decreases over time (i.e., closer to the optimal placement), since remaining $\hat{V}_p$, which EG uses in the greedy search once called, gets smaller. Hence, we can prune more candidate paths, which $u_p \geq u^{upper}$ (line 11). Finally, once the first entry of OQ exceeds the current u^{upper}, the search can be safely finished with u^{upper} (line 6). Note that under homogeneous resource requirements and uniform operating conditions, the search can immediately be finished with u^{upper} since the initial estimate u_0 can be equal or greater than u^{upper}.

3) Further Performance Improvement: BA^* can further reduce the search time by removing redundant computation of u for candidate placements under certain assumption. Specifically, if we can assume that nodes of each diversity zone d^z_k have the same resource requirements, BA^* computes the candidate placement just for one of those nodes, since the resulting placements for those nodes will be identical. Thus, BA^* copies the candidate placements into the open queue OQ without any further computation. This assumption may be reasonable in multi-tier cloud applications as mentioned in [7].

C. Deadline-Bounded Search

BA^* performs well even under heterogeneous resource requirements and non-uniform resource availability. However, it is computationally expensive when we deploy a large T^a onto a large-scale T^p. In the worst case, the time complexity of BA^* can be $O(|V||H|^{|V|})$ without any assumption, where $|V||H|$ is the number of all possible branches[10][11]. Hence, it would be desirable to obtain a solution quicker while sacrificing optimality only minimally.

To address the trade-off between the running-time t and optimality, we have developed the deadline-bounded A* (DBA^*) that extends BA^*. It can find the near-optimal placement within a given computation time limit $\mathcal{T}$ by dynamically pruning paths over the progress towards the optimal u.

The pruning decision for a given path is based on the number of nodes that have already been placed in the path, (i.e., $|V^*_p|$). As a path is closer to the end of the search (i.e., its $|V^*_p|$ is closer to $|V|$), it has less chance to be pruned. Therefore, the search is biased to be depth first, and it avoids the generation of too many candidate paths that have small and identical $|V^*_p|$. DBA^* increases the pruning rate over time. It indicates that paths with larger u have more chance to be pruned, since DBA^* sorts paths in the open queue OQ by u and searches with the path having the least u first.

Formally, each path is pruned with the probability $p(x > s)$, where x is a random number uniformly selected from a range [0, r), and s is the progress rate that is computed by $(|V^*_p|)/|V|$. The upper bound r of the range directly affects the pruning rate, and it is dynamically adjusted depending on the time left before $\mathcal{T}$.

DBA^* periodically monitors the time left, i.e., $\mathcal{T}^{left} = \mathcal{T} - (t^{curr} - t^{start})$, where t^{curr} is the current time and t^{start} is the start time of the search. When half of the previously

estimated $\mathcal{T}^{left}$ has been consumed, it decides if the search can be completed by $\mathcal{T}^{left}$. If it cannot be done, DBA^* adjusts the pruning rate by increasing r by a parameter α (e.g., $\alpha = 0.2 * (\mathcal{T}/\mathcal{T}^{left})$ in our implementation).

For making a decision on pruning, DBA^* estimates the number of paths left in the open queue OQ to be explored. In particular, it computes the ratio ($|P^{left}|$ / $|P|$), where $|P^{left}|$ is the estimate of the number of paths to be generated in OQ along the search, and $|P|$ is the estimate of the number of paths DBA^* can handle before $\mathcal{T}^{left}$. $|P|$ can be estimated by dividing $\mathcal{T}^{left}$ by the average delay to handle a single path. Meanwhile, $|P^{left}|$ is computed as follows:

$$|P^{left}| = |P^{left}| + L[i] * (1 - p(x > s)),$$
$$L[i+1] = L[i+1] + L[i] * (1 - p(x > s))^2 * |\bar{\mathcal{P}}|$$
$$where\ \forall i, \quad l \leq i < |V|$$

$L[i]$ is the number of paths in OQ, where its length (i.e., $|V_p^*|$) is i, and $|\bar{\mathcal{P}}|$ is the average number of newly generated paths (line 8 of Algorithm 2). Then, the equation indicates that each path with length i will be pruned with the rate $p(x > s)$ when being selected to progress, and if not pruned, it will generate $|\bar{\mathcal{P}}|$ new paths of length $(i+1)$, but these new paths are pruned at the rate $p(x > s)$ as well before being inserted into OQ.

Conservatively, DBA^* may handle all paths in OQ just before it reaches a path with length $|V|$, so that $|P^{left}|$ will be accumulated over all $L[i]$'s, where $l \leq i < |V|$. It counts $|P^{left}|$ starting from $L[l]$ since paths with length $i < l$ are mostly handled at the moment of estimating. Note that $|\bar{\mathcal{P}}|$ changes over time so that DBA^* updates it periodically.

The larger the $\mathcal{T}$, the better the placement produced by DBA^*. This is because DBA^* can consider more candidate paths in OQ (i.e., do less pruning).

IV. Evaluation

We evaluate two aspects of Ostro. First, we demonstrate the impacts of application's heterogeneous resource requirements and non-uniform resource availability on placement optimization. Second, we evaluate how Ostro addresses the tradeoff between resource utilization and the running time on different application scales.

A. Experiment Setup

Using a cloud application in a small-scale testbed, we analyze the impact of application heterogeneity and non-uniform resource availability on the resource utilization resulted from our three algorithms. To compare against heuristics presented in prior work, we have created two versions of EG, one that minimizes the number of hosts used (EG^C) while the other minimizes bandwidth usage (EG^{BW}). EG^C minimizes u^c by always trying to choose first the host with the smallest remaining compute capacity. EG^{BW} minimizes only the u^{bw}—similar to the prior work such as [4][5][8][7]—by placing linked VMs as close to one another as possible. Note that we have not implemented the same greedy algorithms used in those prior works because they either use different application models ([4][5][8]) or different objective function ([7]).

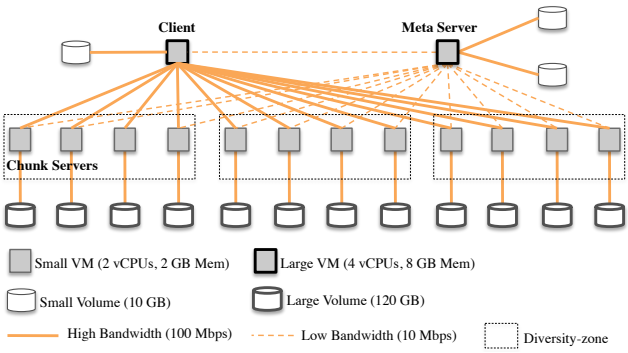


Fig. 5: Simple topology of the cloud-based QFS application

For the cloud application, we employ QFS (Quantcast File System [15]) consisting of chunk server VMs to store file chunks to disk volumes, meta server VMs to maintain the meta-data about the location of the file chunks, and client VMs to run a benchmarking program for the distributed file systems. Figure 5 illustrates a simple QFS application topology that includes 1 meta server, 1 client, 12 chunk servers, 15 disk volumes, and their resource requirements. As shown in the figure, the QFS application topology has heterogenous resource requirements (i.e., large and small VMs, volumes, and bandwidth requirements).

We place the QFS application onto a cluster that has 16 host servers connected by a ToR switch. Each host has 16 CPU cores with dual 2.67 GHz Intel Xeon processors, 32 GB memory, and a 1 TB disk. To establish non-uniform resource availability on the cluster, we have deployed several VMs and volumes to the cluster before computing the optimized placement of the QFS application. Specifically, the first four hosts of the cluster are relatively lightly utilized (i.e., 8 or 10 available CPU cores and more than 20 GB free memory), the next four have medium utilization (i.e., 5 or 6 available cores and 15–19 GB of available memory), the next four hosts are resource constrained (i.e., less than 5 cores and less than 15 GB memory), and the final four are idle with all the cores and memory available. The bandwidth between each host and ToR switch is 3200 Mbps.

B. Experimental Results

	EG^C	EG^{BW}	EG	BA^*	DBA^*
Bandwidth (Mbps)	4480	1980	2000	1980	1980
New active hosts	0	4	0	1	1
Run-time (sec)	0.058	0.082	0.084	7.842	0.513

TABLE I: Comparison under non-uniform resource availability.

We first set θ^{bw} and θ^c to 0.99 and 0.01, respectively, for EG, BA^*, and DBA^* to see how these algorithms can minimize bandwidth usage. Here, u^c is used as the tie-breaker in the search.

As shown in Table I, EG, BA^*, and DBA^* reserve less than half of the bandwidth used by EG^C because EG^C does not consider the communication links between VMs, but merely performs bin-packing based on available host resources. EG reserves a little more bandwidth than the minimum value achieved by EG^{BW}, BA^*, and DBA^*. However, EG does not utilize any new hosts, while EG^{BW} uses all the remaining idle hosts to minimize the bandwidth usage. This is because EG^{BW} tries to use the hosts that have the most available bandwidth first. Note that there are 12 hosts already used by other VMs in the testbed, and the number of new active hosts in the table indicates the number of formerly idle hosts that are now used to run components of the QFS application.

Optimizing the placement of the QFS application relies on placing as many chunk servers as possible together with the client VM on a single host, and placing each volume with corresponding linked VM. All algorithms except EG^C come up with such placement. EG^{BW}, BA^*, and DBA^* can save further 20 Mbps by placing the meta server in a new host.

This experiment shows that DBA^* can efficiently identify a placement that has the same resource utilization as that of BA^* but its running time is much shorter. Even under non-uniform resource availability, DBA^* (and BA^*) can bound the search space on the ideal co-placements of QFS components and prune many other non-optimal paths after it runs EG at the beginning of the search. Additionally, DBA^* prunes further to finish the search within the given deadline. In our experiments, we set $\mathcal{T}$ (running time) to 0.5 sec. DBA^* can identify the best placement after running EG two times in the search.

Finally, we increased θ^c to 0.4 increase the importance of minimizing the number of hosts used in the placement. The resource utilizations of EG^C, EG^{BW}, and EG remained the same as those reported in Table I, while the values of BA^* and DBA^* were changed to be identical with the values of EG. This shows that BA^* and DBA^* adjust the placement based on θ^c, while EG^C, EG^{BW}, and EG rely on a fixed initial setup by sorting nodes.

	EG^C	EG^{BW}	EG	BA^*	DBA^*
Bandwidth (Mbps)	2380	1980	1980	1980	1980
New active hosts	4	4	4	4	4
Run-time (sec)	0.055	0.082	0.082	0.882	0.185

TABLE II: Comparison under uniform resource availability

We also perform an experiment under uniform resource availability, where all hosts were idle before QFS was deployed. As shown in Table II, all algorithms except EG^C achieve the same resource utilization. This is because the available capacity of each host is large and identical and therefore, any host can be chosen as the best fit in the search process. Specifically, the number of hosts used is determined simply by the diversity zone requirements of the chunk servers in all algorithms and then, all algorithms except EG^C try to co-locate components of QFS that have a communication link with each other. Moreover, EG can achieve the same resource utilization as BA^*, and therefore DBA^* and BA^* can finish the search at an early stage by significantly bounding the search space using EG.

C. Simulation Setup

To evaluate its scalability, we executed Ostro on simulated application topologies and data centers at different scales. Two types of application topologies, a multi-tier [12][7] and a mesh communication [3] topologies, as shown in Figure 2 are used. The multi-tier topology has 5 tiers, where each tier is populated with some number of nodes ranging from 5 to 40. Nodes of each tier are divided into 2 diversity zones d^{host}. For the mesh communication topology, the size of the topology ranges from 25 to 200 VMs. Each diversity zone d^{host} has 5 VMs, and we scale up the topology by increasing the number of disjoint diversity zones from 5 to 40. For each diversity zone, we randomly select around 80% of the other diversity zones and establish communication links between VMs in these separate diversity zones.

For both applications, we configure heterogeneous resource requirements by mixing compute intensiveness and networking intensiveness as shown in Table III. For the simulation with homogeneous requirements, we set all VMs with 2 vCPUs, 2 GB memory, and 50 Mbps.

	vCPUs	Memory (GB)	Bandwidth (Mbps)
40% of VMs	1	1	100
20% of VMs	2	2	50
40% of VMs	4	4	10

TABLE III: Heterogeneous resource requirements

We simulate a large-scale data center with a total of 2400 hosts arranged in 150 racks of 16 hosts each. The hierarchical structure is based on Figure 3, but without the pod switches for simplicity. The bandwidth between each host and the ToR switch is 10 Gbps, and between the ToR switch and the root switch is 100 Gbps. We configure the non-uniform resource availability of hosts in each rack as shown in Table IV. For the homogeneous case, we set all hosts as fully available. In the simulation, we set θ^{bw} and θ^c to 0.6 and 0.4, respectively.

	CPU cores	Memory (GB)	Bandwidth (Gbps)
25% of hosts	9–16	17–30	0–1.5
25% of hosts	6–8	8–16	2–5
25% of hosts	0–5	0–7	6–8
25% of hosts	16	32	10

TABLE IV: Configuration of non-uniform resource availability

D. Simulation Results

1) Time-Optimality Tradeoff: Figure 6 illustrates the tradeoff between running time and the optimality of the placement. In this simulation, we run DBA^* with different $\mathcal{T}$ (running time) for the multi-tier application with 200 VMs with heterogenous requirements. Each data point is the average of 20 executions.

As $\mathcal{T}$ increases, better resource utilization is achieved both in terms of bandwidth and the number of hosts. In particular, DBA^* dramatically decreases the bandwidth usage when $\mathcal{T}$

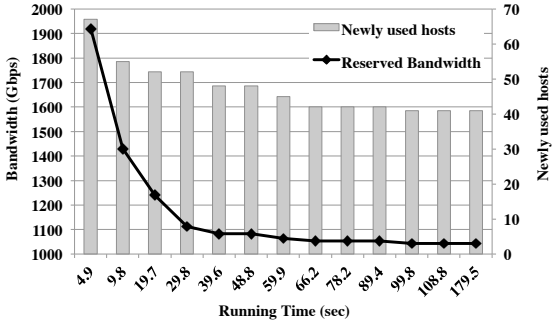
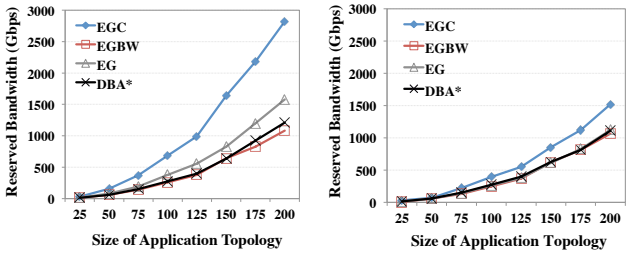


Fig. 6: Tradeoff between $\mathcal{T}$ and the optimality of placement

is increased from 4.9 to 29.8 seconds. After this point, the reduction is minimal. This indicates that the critical decisions of DBA^* for exploring the search space are made in the early stages of the search with high probability. Before exponentially expanding the search, DBA^* increasingly prunes many candidate paths, which are likely non-optimal. With $\mathcal{T} = 20$, the total pruning rate reaches around 90%.

The lower bound of $\mathcal{T}$ can be estimated as two times the running time of EG taken for the first bounding. This is because the number of paths handled at each stage in DBA^* (i.e., $|V||\mathcal{H}|$) is approximately the same as the number handled in EG.

2) Multi-Tier Application: We further examine the tradeoff between running time and optimality by comparing four algorithms with various configurations of the multi-tier application.



(a) Heterogeneous requirements (b) Homegeneous requirements

Fig. 7: Bandwidth reserved for multi-tier application

Similar to the QFS experiment, the difference of resource utilization between algorithms under heterogeneous requirements and non-uniform resource availability is higher than under homogeneous requirements and uniform availability. Moreover, the difference increases as the size of the topology increases, as shown in Figure 7. This indicates the sorting of $|V|$ significantly affects the placement. DBA^* does not use initial sorting, so it will perform well in any condition.

Figure 7a and Figure 8 also show the tradeoff between the bandwidth usage and the host usage. As EG^{BW} tries to minimize the bandwidth usage (Figure 7a), it uses many new hosts that have more available bandwidths (Figure 8). Meanwhile, EG^{C} consolidates VMs to fewer hosts but these VMs consume more bandwidth to communicate across hosts or racks. EG and DBA^* consider both bandwidth and host usage. Especially, DBA^* tries to identify the optimal balance

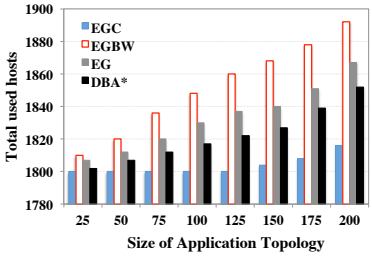
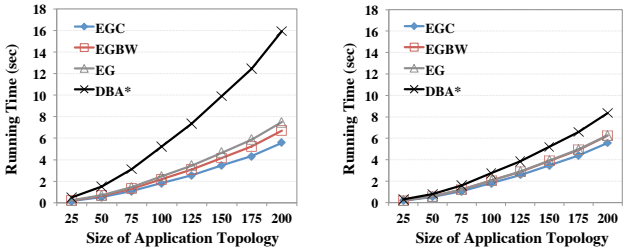


Fig. 8: The number of hosts for multi-tier application

between them as shown in the graphs.



(a) Heterogeneous requirements (b) Homegeneous requirements

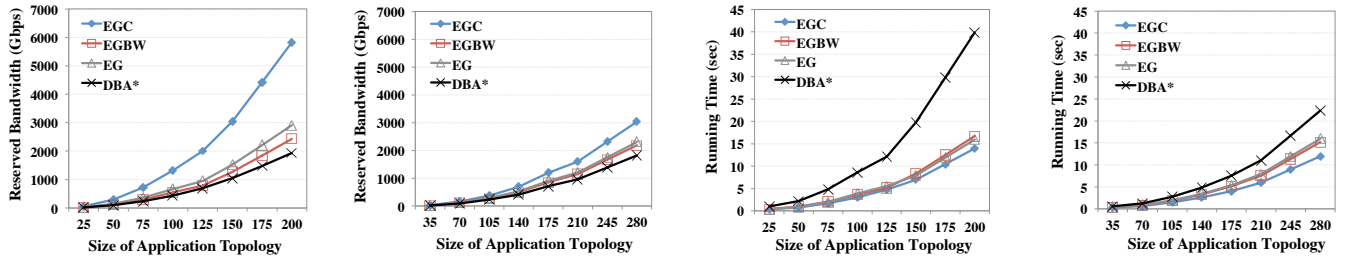
Fig. 9: Run time comparison for multi-tier application

The running time of EG is similar with those of EG^{C} and EG^{BW} while it identifies a better balance of resource utilization even under heterogeneous requirements (Figure 9a). However, DBA^* spends more time than the other algorithms to identify the best placement. This is because the bounding ratio driven by the first EG run in DBA^* is not so effective, so that DBA^* should explore more candidate placements until the second run of EG is performed. As shown in Figure 9b, the running time of DBA^* is close to the other algorithms under homogeneous resource requirements and uniform resource availability case since the first EG run in this case tightly bounds the search space.

3) Mesh Communication Application: Figure 10 and 11 show the results of the mesh communication application case. The overall trend is consistent, but the bandwidth usage is significantly larger than with the multi-tier application, since each VM has more bandwidth requirements. Moreover, because the application topology is more complex, and the resource requirements of VMs are more varied than in the previous case, the running time is higher in all algorithms. The results also indicate that DBA^* identifies better placement for the bandwidth usage than all other greedy algorithms including EG^{BW} when dealing with such a complex application.

E. Online Adaptation

An application topology can be updated online by, for example, adding or removing VMs, or changing resources requirements. In this case, Ostro also re-computes the placement online. We observe that such updates are typically made in an incremental way on a small portion of the application topology over time, rather than drastic changes on a large portion in a short period. Hence, computing the new optimized placement



(a) Heterogeneous requirements (b) Homegeneous requirements (c) Heterogeneous requirements (d) Homogeneous requirements

Fig. 10: Bandwidth and run time comparisons for mesh communication application

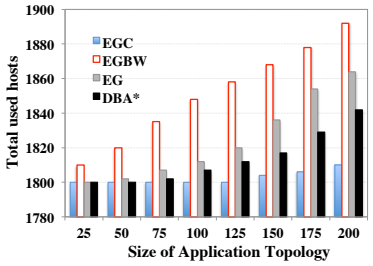


Fig. 11: Number of hosts for mesh application

can be done very efficiently. We have simulated adding 10% more small VMs on the first or second tier of the multi-tier application topology (200 VMs), and the new placement optimization is completed within 0.3 sec using DBA^*.

This simulation also shows that adding additional VMs can trigger the re-positioning of previously placed nodes in the application, and it can in fact spread out to a large portion of the application nodes if we are adding more than 10% VMs. We will continue to investigate incremental update strategies and tradeoffs as part of our future work.

V. Related Work

The problem of mapping a logical topology (such as our application topology) into a physical topology has been addressed in many research areas including distributed memory parallel computing, peer-to-peer systems, and grid computing. In this section, we introduce some recent research that has tackled the similar problem but in the context of data center and cloud computing.

Oktopus [4] and SecondNet [3] use a virtual topology as the means for explicitly exposing QoS requirements, and use heuristic algorithms to reduce the cost for cloud providers when scheduling the virtual topology. Oktopus assumes a virtual cluster topology resembling the traditional hierarchical data center (i.e., one- or two-level tree structure rooted with a virtual switch), while SecondNet can deal with more complex topologies. SecondNet uses a greedy algorithm that places one link between two VMs at a time, but assumes that one host server can contain only one VM of the virtual topology. Authors of [8] use a virtual data center topology similar to Oktopus, and propose a heuristic algorithm to satisfy stochastic bandwidth requirements under demand uncertainty. CloudMirror [7] considers application reliability for complex application topologies similar to Ostro's, but it mainly focuses on the bandwidth resource. Rather than assuming limited virtual topologies and limited homogenous and uniform resource usage environments, Ostro supports any application topology in arbitrary environments across compute, storage, and network resources. Therefore, the placement algorithm of Ostro is more general.

Proteus [5] maximizes the network bandwidth utilization between two VMs, but deals with it dynamically. Its dynamic mechanism can be complementary to our approach. A heuristic approach that minimizes the number of host servers to save energy while placing VMs is proposed in [16]. Similarly, authors of [17] attempt to intensively consolidate VMs based on resource usage patterns. However, those approaches do not consider the dependencies between application components.

Another problem relevant to this research area is service composition for cloud applications. For example, [18] addresses the optimal selection of an instance for each component from multiple clouds, [19] takes the network distance and load balancing into consideration when selecting servers, Cloud-GPS [20] tries to minimize the network traffic and latency, and [21] accounts for energy consumption when tackling the service selection problem. However, these proposals do not address large application topologies, but consider only single tier applications.

Another problem that explicitly considers the networking between components is the service routing problem. Its goal is to use data center networks efficiently by choosing the network routes for application-level flows between existing VMs. Thus, the assumption is that VMs have already been placed and the scheduling algorithm only considers the routing of communication paths. Hedera [22] routes long-lived flows to achieve high bandwidth utilization in a data center by leveraging the fact that there are multiple paths between any two host servers. A load balancing algorithm integrated with an OpenFlow controller is presented in [23]. It makes routing decisions aimed at evenly distributing the network traffic in a data center network. A similar mechanism to consolidate the network traffic to a few links is presented in [24]. Unlike these approaches, we focus on the placement of a given application topology that considers not only the network resources in decision making, but also compute and storage.

Many heuristic algorithms can be used to solve the type of optimization problems addressed here. The A* Prune algo-

rithm was proposed to find *K* shortest paths in a graph subject to multiple constraints [25]. An A* graph search algorithm to place parallel service workflow into multiple data centers was proposed in [26]. Load balancing for packet routing using a mixed integer programming with the goal of minimizing the network energy consumption is presented in [27]. Evolutionary approaches can also be used to determine the optimal solution in this domain, including Simulated Annealing [28], Genetic Algorithm [29], Particle Swarm Optimization [30], and Artificial Bee Colony [31]. However, it is non-trivial to guarantee an optimal solution in a tight time bound for complex application topologies using such heuristic algorithms.

VI. Conclusions

In this paper, we have addressed the problem of placing complex applications in large-scale data centers, and presented an OpenStack-based solution called Ostro. The goal of Ostro is to meet the dual goals of optimizing resource utilization and meeting application requirements, including properties related to resource requirements, diversity of placement for reliability, and communication dependencies. To do this, we define *application topology* as our fundamental and indivisible unit of scheduling. This abstraction captures the reality that a given application is a complex entity consisting of multiple different types of resources, QoS requirements, and resource usage patterns that all need to be considered holistically when making placement decisions. Ostro does this, while also dealing with the complexities introduced by multi-tenancy such as non-uniform resource availability. We have demonstrated the effectiveness of Ostro under multiple scenarios using both simulation and experimental results with a real application. We also show the impacts of heterogenous resource requirements and non-uniform resource availability on the optimized placement, and evaluated how Ostro effectively addresses the tradeoff between scalability and optimality.

As future work, we plan to add other property attributes to application topologies, including latency requirements for the communication links between nodes, as required by real world applications. We also envision specifying additional properties for VMs. For example, a VM could have a guaranteed or best effort CPU reservation. These demands do not affect the main approach used by Ostro, but do broaden the requirements for the optimized placement.

References

[1] M. Ramachandran, "Component-based development for cloud computing architectures," in *Cloud Computing for Enterprise Architectures*. Springer London, 2011, pp. 91–114.

[2] N. Feamster, J. Rexford, and E. Zegura, "The road to SDN: An intellectual history of programmable networks," *ACM Queue - Large-scale implementations*, vol. 11, no. 12, p. 20, 2013.

[3] C. Guo, G. Lu, H. J. Wang, S. Yang, C. Kang, P. Sun, W. Wu, and Y. Zhang, "SecondNet: A data center network virtualization architecture with bandwidth guarantees," in *Proc. of ACM CoNEXT*, 2010.

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

[5] D. Xie, N. Ding, Y. C. Hu, and R. Kompella, "The only constant is change: Incorporating time-varying network reservations in data centers," in *Proc. ACM SIGCOMM*, 2012, pp. 199–210.

[6] "Openstack Nova scheduler," http://www.rackspace.com/blog/openstack-nova-scheduler-and-vsphere-drs/.

[7] J. Lee, Y. Turner, M. Lee, L. Popa, S. Banerjee, J. Kang, and P. Sharma, "Application-driven bandwidth guarantees in datacenters," in *Proc. ACM SIGCOMM*, 2014, pp. 467–478.

[8] L. Yu and H. Shen, "Bandwidth guarantee under demand uncertainty in multi-tenant clouds," in *Proc. IEEE ICDCS*, 2014, pp. 258–267.

[9] R. Dechter and J. Pearl, "Generalized best-first search strategies and the optimality of a*," *ACM Journal*, vol. 32, no. 3, pp. 505–536, 1985.

[10] A. Koll and H. Kaindl, "A new approach to dynamic weighting," in *Proc.* 10^{th} *European Conference on AI*, 1992, pp. 16–17.

[11] J. Pearl and J. H. Kim, "Studies in semi-admissible heuristics," *IEEE TPAMI*, vol. 4, no. 4, pp. 392–399, 1982.

[12] G. Jung, M. Hiltunen, K. Joshi, R. Schlichting, and C. Pu, "Mistral: Dynamically managing power, performance, and adaptation cost in cloud infrastructures," in *Proc.* 30^{th} *IEEE ICDCS*, 2010, pp. 62–73.

[13] G. Jung, K. Joshi, M. Hiltunen, R. Schlichting, and C. Pu, "Performance and availability aware regeneration for cloud based multi-tier applications," in *Proc. of IEEE DSN*, 2010, pp. 497–506.

[14] N. G. Duffield, P. Goyal, A. Greenberg, P. Mishra, K. K. Ramakrishnan, and J. E. van der Merive, "A flexible model for resource management in virtual private networks," in *Proc. ACM SIGCOMM*, 1999.

[15] M. Ovsiannikov, S. Rus, D. Reeves, P. Sutter, S. Rao, and J. Kelly, "The quantcast file system," *Proceedings of the VLDB Endowment*, vol. 6, no. 11, pp. 1092–1101, 2013.

[16] J. Famaey, T. Wauters, F. D. Turck, B. Dhoedt, and P. Demeester, "Network-aware service placement and selection algorithms on large-scale overlay networks," *Computer Communications*, vol. 34, no. 15, pp. 1777–1787, 2011.

[17] L. Chen and H. Shen, "Consolidating complementary vms with spatial/temporal-awareness in cloud datacenters," in *Proc. IEEE INFOCOM*, 2014, pp. 1033–1041.

[18] X. Li, J. Wu, and S. Lu, "QoS-aware service selection in geographically distributed clouds," in *Proc.* 22^{nd} *Int. Conf. on Computer Communications and Networks*, 2013, pp. 1–5.

[19] P. Wendell, J. W. Jiang, M. J. Freedman, and J. Rexford, "Donar: Decentralized server selection for cloud services," in *Proc. ACM SIGCOMM*, 2010, pp. 231–242.

[20] C. Ding, Y. Chen, T. Xu, and X. Fu, "CloudGPS: A scalable and ISP-friendly server selection scheme in cloud computing environments," in *Proc.* 20^{th} *IEEE/ACM IWQoS*, 2012.

[21] Z. Liu, M. Lin, A. Wierman, S. H. Low, and L. Andrew, "Greening geographical load balancing," in *ACM SIGMETRICS*, 2011, p. 233.

[22] M. Al-Fares, S. Radhakrishnan, B. Ragavan, N. Huang, and A. Vahdat, "Hedera: Dynamic flow scheduling for data center networks," in *Proc.* 7^{th} *USENIX NSDI*, 2010.

[23] N. Handigol, M. Flajslik, S. Seetharaman, N. McKeown, and R. Johari, "Aster*x: Load-balancing as a network primitive," in *Architectural Concerns in Large Datacenters*, 2010.

[24] B. Heller, S. Seetharaman, P. Mahadevan, Y. Yiakoumis, P. Sharma, S. Banerjee, and N. McKeown, "Elastictree: Shaving energy in data center networks," in *Proc.* 7^{th} *USENIX NSDI*, 2010.

[25] G. Liu and K. Ramakrishnan, "A*Prune: an algorithm for finding K shortest paths subject to multiple constraints," in *Proc.* 12^{th} *IEEE INFOCOM*, vol. 2, 2001, pp. 743–749.

[26] G. Jung and H. Kim, "Optimal time-cost tradeoff of parallel service workflow in federated heterogeneous clouds," in *Proc.* 20^{th} *IEEE Int. Conf. on Web Services*, 2013, pp. 499–506.

[27] D. Erickson, B. Heller, S. Yang, J. Chu, J. Ellithorpe, S. Whyte, S. Stuart, N. McKeown, G. Parulkar, and M. Rosenblum, "Optimizing a virtualized data center," in *Proc. ACM SIGCOMM*, 2011.

[28] A. YarKhan and J. Dongarra, "Experiments with scheduling using simulated annealing in a grid environment," in *Proc.* 3^{rd} *Int. Workshop on Grid Computing*, 2002, pp. 232–242.

[29] J. Yu and R. Buyya, "Scheduling scientific workflow applications with deadline and budget constraints using genetic algorithms," *Sci. Program*, vol. 14, no. 3,4, pp. 217–230, 2006.

[30] S. Pandey, L. Wu, S. Guru, and R. Buyya, "A particle swarm optimization-based heuristic for scheduling workflow applications in cloud computing environments," in *Proc.* 24^{th} *IEEE Int. Conf. on Advanced Information Networking and Applications*, 2010, pp. 400–407.

[31] A. Banharnsakun, B. Sirinaovakul, and T. Achalakul, "Job shop scheduling with the best-so-far ABC," *Engineering Applications of Artificial Intelligence*, vol. 25, no. 3, pp. 583–593, 2012.