Application Placement and Demand Distribution in a Global Elastic Cloud: A Unified Approach

Hangwei Qian
VMWare
Palo Alto, CA, 94304

Michael Rabinovich
Case Western Reserve University
Cleveland, OH 44106

Abstract

Efficient hosting of applications in a globally distributed multi-tenant cloud computing platform requires policies to decide where to place application replicas and how to distribute client requests among these replicas in response to the dynamic demand. We present a unified method that computes both policies together based on a sequence of min-cost flow models. Further, since optimization problems are generally very large-scale in this environment, we propose a novel demand clustering approach to make them computationally practical. An experimental evaluation, both through large-scale simulation and a prototype in a testbed deployment, shows significant promise of our approach for the targeted environment.

1 Introduction

An important benefit of cloud computing is that it allows Internet application providers to obtain global footprint and elastic capacity without the need to deploy and maintain their own infrastructure. This service is often referred to as IaaS ("Infrastructure as a Service"). Cloud providers can offer IaaS efficiently by deriving the economy of scale though multiplexing their shared platforms among multiple applications. A number of cloud providers, including Google, Microsoft, and Amazon, offer some variation of this capability.

These geo-distributed multi-tenant hosting platforms need to be able to effectively distribute the hosted applications across multiple data centers and direct client demand to the appropriate application replicas. Specifically, this task involves the following two key policies: (i) At how many and which data centers should each application be deployed? We refer to this as the (global) application placement problem; and (ii) How should client demand be distributed to these application replicas? This is commonly referred to as demand distribution or, interchangeably, server selection problem.

Much prior work has targeted environments addressing one or the other of these aspects (see § 8). However, an elastic cloud must deal with both issues simultaneously because it distributes demand among dynamically changing sets of application instances. Consequently, we propose a unified framework to compute these two policies simultaneously. A comparison with an existing approach that also addressed both policies but computed them in isolation showed a significant advantage of our approach (§ 6.6).

Computing these policies in a hosting cloud brings an additional challenge. Because request processing involves accesses of application-specific back-end servers, the proximity of a request to data centers depends not just on the client's location but also on the location of the back-end servers and hence on the requested application. This increases the scale of the optimization problems by orders of magnitude (§ 4). We propose a novel demand clustering approach we call *permutation prefix clustering*, and show that it makes global optimization practical in many environments.

In summary, this work addresses the application placement and demand distribution problems in a geo-distributed and globally-shared cloud platform, and makes the following contributions: (i) We propose and evaluate a unified framework to jointly solve the application placement and demand distribution problems; (ii) A novel clustering technique is introduced to scale our optimization model to realistic platform sizes, which we believe will prove valuable for other optimization models as well; and (iii) We prototype our approach and demonstrate its operation in a testbed deployment.

2 System Overview

The high-level view of our targeted environment is shown in Fig. 1. Each client connects to a request-routing component (e.g., DNS server or HTTP redirector), which directs it to a data center hosting requested application. Known mechanisms (such as one provided in WebLogic [1]) ensure continued session state availability even if a client is redirected to a new instance mid-session. When processing requests, the application is assumed to access back-end database located at the premises of the application providers for security or legal reasons (the extended version of this paper also considers the hosted database scenario [27]). Thus, we aggregate the network distances from clients to data centers and from data centers to databases when calculating the network delay for the requests. Note, obtaining the distance information efficiently is a complicated task and an important part of the providers' know-how. We assume

this information is supplied by a separate measurement component (not considered here).

To effectively manage the proximity information of all the Internet clients, we assume the platform groups its clients by the IP prefixes found in multiple BGP tables [19], with each group dubbed a *client cluster (CC)*. A key part of our work is to aggregate demand further so that computing the application placement and demand distribution policies becomes tractable. Also, many cloud providers concentrate their platforms in a small number of strategically located mega data centers, leading to the factors of 5 to 7 decrease in the operational cost [12]. For example, Amazon EC2 is deployed in four locations (US-East, US-West, Ireland, and Singapore); other infrastructure providers, such as Limelight and AT&T, have a couple dozens data centers. We assume roughly this number of data centers (around 20). We do not target platforms such as Akamai with presence in thousands of locations.

We focus on the business models (e.g., *auto-scaling* option in EC2 or Google's AppEngine), in which the cloud itself makes the decisions on the number and location of various application instances, to maximize the application performance and minimize the number of data centers where these applications are deployed. Removing underutilized application instances reduces the customer costs and frees up resources for other applications. Another objective is to reduce the number of placement changes in consecutive configurations. Despite recent advances in reducing overhead of starting a virtual machine [21] or an application server [9], deploying an application instance remains a heavy-weight operation in terms of CPU costs and system reconfiguration.

In making placement decisions, we only consider whether or not an application is deployed in a data center. Others have addressed the problem of resource allocation among applications within a data center [37, 26, 33]. In the rest of the paper, an "application instance" means that the application is deployed at the data center, regardless of the amount of resources it is assigned locally.

Resource allocation decisions require monitoring the demand and utilization of data centers. We assume a central controller collects this information periodically from each data center. Like any platform based on request routing, our target environment requires translation between requests and service demands; this so-called *application modeling* problem has been studied intensively (e.g., [32, 34, 35]) and we assume the use of one of these existing technologies. We also assume that our applications (i.e., web sites) are sufficiently popular so that even if different requests to a web site have different service demands, for a reasonable request rate (e.g., higher than the deletion threshold - see § 5), these requests will result in a representative request mix. (If this assumption

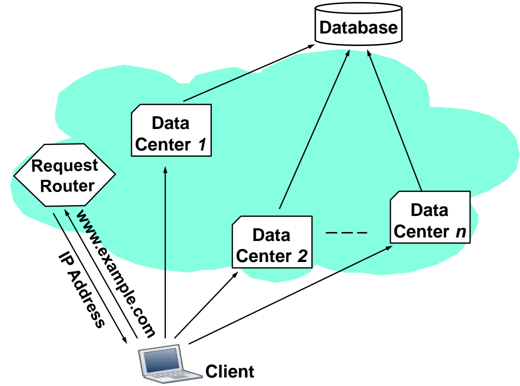


Figure 1: Overview

does not hold for an application, its requests must be split into classes with similar service demands and each class modeled separately.) Meanwhile, request rates for different applications are normalized so that the same (normalized) request rate will result in the same resource utilization regardless of the application. Thus, a request rate translates to the proportional resource usage and can be used to measure the capacity and the utilization of data centers. (This assumption is supported by our experience with prototype in § 7.)

2.1 Problem Statement

Let D be the number of data centers, A the number of applications and C the number of client clusters. The placement policy can be described as an $A \times D$ matrix P, with element $P_{ij} = 1$ if application i is deployed at data center j; $P_{ij} = 0$, otherwise. The demand distribution policy is an $A \times C \times D$ matrix R, whose element R_{amn} is the fraction of requests from client cluster m for application a to be directed to data center n. The system enacts the distribution policy by directing a request from client cluster m for application a to data center n with probability R_{amn}. Let r_{am} be the request rate for application a from client cluster m. Assume each request is associated with a cost C_{amn} if it is served at data center n, and u_n is the utilization of the data center. We formulate our problem as a *multi-objective optimization problem* [23] fulfilling the following competing objectives:

$$Minimize \sum_{a=1}^{A} \sum_{m=1}^{C} r_{am} \sum_{n=1}^{D} R_{amn} C_{amn} \tag{1}$$

$$Minimize \sum_{a=1}^{A} \sum_{n=1}^{D} P_{an} \tag{2}$$

$$Minimize \sum_{a=1}^{A} \sum_{n=1}^{D} |P_{an} - P_{an}^{prev}| \tag{3}$$

subject to

$$\sum_{a=1}^{A} \sum_{m=1}^{C} r_{am} R_{amn} \leq u_n, n = 1, 2, \ldots, D \tag{4}$$

$$0 \leq R_{amn} \leq 1; \sum_{n=1}^{D} R_{amn} = 1 \tag{5}$$

$$P_{an} \in \{0,1\};\ R_{amn} > 0 \ implies \ P_{an} = 1 \tag{6}$$

where P_{an}^{prev} is the previous placement policy. Objective (1) minimizes the overall cost. While C_{amn} is an abstract cost function, we use aggregate distance (measured as network latency) as the cost function thus trying to minimize the overall user-perceived network latency. Objective (2) minimizes the number of data centers with deployed application replicas and objective (3) minimizes the number of placement changes.

While multi-objective optimization problems are commonly handled by combining all the objectives into a single one with some weights assigned to each objective, in our case, this would transform the problem to a mixed integer programming formulation (in fact, a variant of a multicommodity capacitated facility location problem [8]), which is NP-hard. Further, choosing appropriate weights for different objectives is difficult in our context as their effect on the final policies is indirect and non-intuitive. Instead, we handle the problem heuristically as follows.

2.2 Framework

Our heuristic approach to arbitrate among the competing objectives involves two steps. First, we compute optimal request distribution among data centers assuming every application is deployed at every data center (*full deployment*). Here any optimization technique can be applied. We explore a centralized approach based on a min-cost max-flow model (§ 3).

Second, given the optimal demand distribution policy, we attempt to remove underutilized instances. We introduce a *Deletion Threshold (DT)* as the level of demand that justifies the cost of running an application at a data center (note that the DT can be selected independently for each application and has an easily grasped intuitive meaning). We try to remove instances whose demand after the first step is below DT by reassigning their flows to remaining instances in an optimal manner (§ 5). We also attempt to reduce the number of placement changes in this step by assigning lower deletion threshold to already-deployed instances (§ 5.3).

3 Full Deployment

We begin by obtaining optimal demand distribution policy with full deployment. We use a min-cost max-flow optimization model for this purpose. This model represents the system as a directed network, with *source nodes* generating demand, *sink nodes* consuming this demand, and demand flowing from sources to sinks along edges labeled with $(cost, capacity)$. An edge label indicates the maximum amount of demand that can traverse this edge and the unit cost of such traversal. There are efficient algorithms that solve the min-cost max-flow problem, i.e., find the assignment of demand to edges that maximizes the total satisfied demand while minimizing the total cost. Refer to [7] for details on *min-cost* flow problem and [2] for transforming it to a *min-cost max-flow* problem; we use both terms interchangeably. We utilize the tool [4] in our implementation, which uses an algorithm with complexity $O(V^2 Elog(VCap))$ [16] where V and E are the number of nodes and edges and *Cap* is the maximum edge capacity.

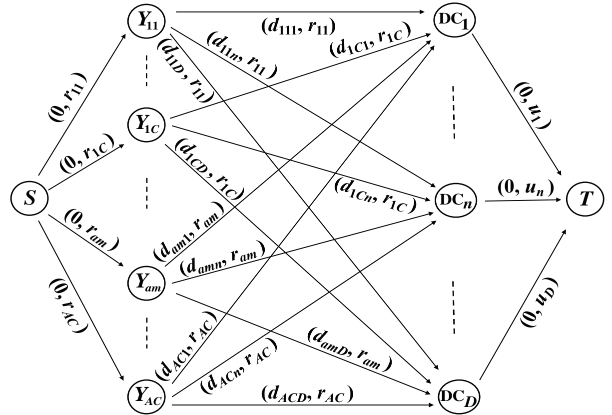


Figure 2: Min-cost network model

3.1 Problem Modeling

We would like to forward client requests to closest data centers and at the same time avoid overloading any data centers. We assume the service does not degrade appreciably as long as data center utilization is below its capacity. (In reality, this means that utilization must stay below a certain *watermark*, which for now we view as capacity but set as a parameter in the simulation – see § 6.) Under this notion, we model our problem as the following min-cost flow network.

Because of different back-end servers, requests for different applications from the same client may have different aggregate distances to the same data center. Thus our model can not simply consider all demand from the same client cluster as a whole. Therefore, as shown in Figure 2, we have a pair-node $Y_{am}, a = 1, 2, ...A, m = 1, 2, ...C$ for each application and client cluster pair (a, m). Also, each data center n has a node DC_n. Finally, we have a source node S and sink node T. From source S to each pair-node Y_{am}, we add an edge with cost 0 and capacity r_{am}, the latter being the request rate from client cluster m for application a. Then we add an edge from each pair-node Y_{am} to each data center node DC_n, with cost being the aggregate distance d_{amn} when client cluster m accesses the application a at data center n, and capacity equal to the full request

rate from this client cluster for this application (since the actual data center capacity is enforced by the subsequent edge), i.e., r_{am}. By connecting each pair-node with every data center, we allow the demand from the corresponding client cluster to be potentially split among any of the data centers. Finally, there is an edge from every data center node DC_n to sink T, with cost 0 and capacity equal to the capacity of data center u_n.

We try to move the total amount of flow $\sum_{a=1}^{A}\sum_{m=1}^{C} r_{am}$ from source S to sink T with minimum cost. After we obtain the solution, flow f_{amn} on the edge between nodes Y_{am} and DC_n represents the amount of requests from client cluster m for application a that should be forwarded to data center n.

4 Permutation Prefix Clustering

The size of the min-cost flow problem in Fig. 2 is extremely large, with $A \cdot C + D + 2$ nodes and $A \cdot C + A \cdot C \cdot D + D$ edges. According to [19], there were on the order of 400,000 client clusters in 2000. Then, for $C = 400,000$ client clusters, $A = 100$ applications, and $D = 20$ data centers, the number of nodes and edges are in the order of 4×10^7 and 8×10^8 respectively, making this problem intractable. We address the scalability problem in this section.

4.1 Basic Idea

With aggregate distance, each pair-node Y_{am} has its own preference of data centers in terms of proximity, producing a permutation of data centers. For example, permutation {1,4,2,3,6,5} means requests for application a from client cluster m are the closest to DC_1, the second closest to DC_4, and so on. We define each permutation as a *region*, and client requests with the same preference of data centers are in the same region. There is a region for each pair-node in Fig. 2. We propose *permutation prefix clustering* to reduce the number of regions and thus the number of edges in Fig. 2.

In this method, we merge regions if their permutations share the same prefix of certain length. For example, for six data centers, let $region_1$ have permutation {1,4,2,3,6,5} and $region_2$ have permutation {1,4,2,3,5,6}. With prefix length 4, we could merge them into $region_{12}$ with prefix {1,4,2,3}. (Note that requests from the *same* client cluster for different applications may end up in *different* regions since their data center preferences may be different due to different back-end servers.) After merging, we compute the distance from the new region to each data center, including those beyond the prefix, as the weighted average of the distances from $region_1$ and $region_2$, with request rate from each region as the weight.

Our observation is that unless most data centers are highly loaded, requests for an application will only go

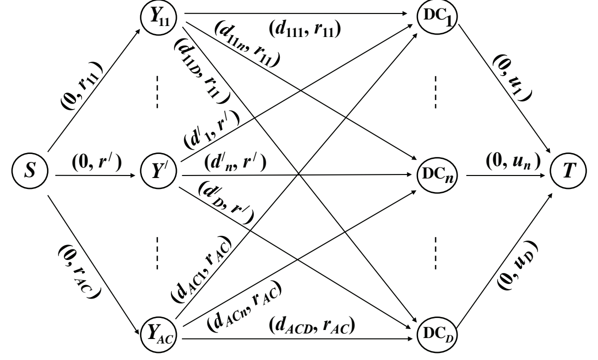


Figure 3: Clustered network model

to a few closest data centers. So for each client request, we only need to consider the front part of its corresponding permutation. Admittedly, there would be proximity penalty when the flows do need to go to the data centers beyond the prefix. However, this happens when most data centers are highly loaded, in which case the proximity becomes less of a priority as we need to satisfy all the demand first. Moreover, our use of the weighted average distance to *all* data centers, including those beyond the prefix, significantly reduces this penalty (see § 6.3).

4.2 Application to Min-Cost Model

To illustrate how *permutation prefix clustering* is applied in our min-cost flow model, suppose we want to merge the regions for pair-nodes Y_{1C} and Y_{am} in Fig. 2 because their permutations share a prefix. We remove nodes Y_{1C} and Y_{am} along with all their adjacent edges and replace them with a new node Y'. An edge is added from source node S to node Y', and from Y' to each node $DC_n, n = 1, 2, ...D$. The cost of the edge from S to Y' is still zero and capacity is the sum of the capacities of the edges (S, Y_{1C}) and (S, Y_{am}), or $r' = r_{1C} + r_{am}$. The cost of the edge $(Y', DC_n), n = 1, 2, ...D$ is the weighted average of cost of edges (Y_{1C}, DC_n) and (Y_{am}, DC_n), or $d'_n = \frac{d_{1Cn} * r_{1C} + d_{amn} * r_{am}}{r_{1C} + r_{am}}$, and capacity is r'. The updated network is shown in Fig. 3. This technique generalizes trivially to merging more than two pair-nodes.

Let L be the length of the permutation prefix. Then the total number of possible regions after merging is:

$$Min\{A * C, \prod_{i=0}^{L-1}(D - i)\}$$

which means the same number of merged pair-nodes Y'. Also, the dominant element of the total number of edges in Fig. 2 is reduced from $A \cdot C \cdot D$ to:

$$D * Min\{A * C, \prod_{i=0}^{L-1}(D - i)\}$$

Since $A * C$ is very large, the total number of nodes and edges in Fig. 3 are in the order of $\prod_{i=0}^{L-1}(D - i)$ and

$D * \prod_{i=0}^{L-1}(D-i)$ respectively, depending on D and L only. Generally, the smaller the prefix length, the smaller the problem size but the larger the potential proximity penalty. We study these effects in § 6.3.

5 Partial Application Placement

A solution to the model of Fig. 3 provides a demand distribution policy assuming full deployment. Our next step is to remove underutilized application instances.

Let f_{an} be the amount of request flow of application a assigned to data center n. If $f_{an} \geq DT$, we call it *normal flow* and keep the instance of application a at data center n. We denote the set of data centers with these instances as U_a. We can immediately remove an instance with zero demand (e.g., if $f_{an} = 0$). The rest of this section handles instances with demand $0 < f_{an} < DT$. We call them *tiny instances* and their flows *tiny flows*.

5.1 Heuristics

Let set $V_a = \{DC_n | 0 < f_{an} < DT\}$ contain data centers with a tiny instance of application a. Also let h_n be the number of normal flows at data center DC_n. We first assume that all tiny instances are removed (unless all instances of an application are tiny, in which case one instance with the largest flow is retained) along with their flows and increase the residual capacities of the affected data centers accordingly. We then attempt to distribute these flows (referred to as residual demand) to data centers with residual capacities. Our procedure is guided by the following observations:

1. We should try to remove the instances with the smallest flows first because the reassignment of small flows will affect fewer requests. In particular, it means that (1a) demand for a tiny instance should not be reassigned to an even tinier instance, and (1b) we should try to accommodate smaller flows (across all applications) first.
2. If we must retain some tiny instances (because data centers in U_a reach their capacity), we should keep the tiny instances with the largest flows first. This is again motivated by the desire to keep the largest amount of demand assigned to the nearest data centers.
3. When selecting data centers in U_a to assign residual demand, we should favor those with smaller h_n because their residual capacity is harder to utilize (since a data center can only accept additional demand for the applications it hosts).

While the above set of heuristics may suggest a simple greedy procedure, where we reassign flows in the increasing size order and distribute them to normal instances first and then to the largest tiny instance with residual capacity, this may result in highly suboptimal flow assignment. Instead, we again build a min-cost flow model for this problem, so that we reassign the residual demand optimally, and at the same time manipulate the costs in the model to follow the above heuristics.

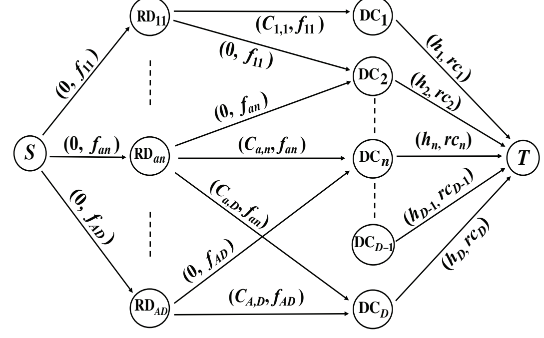


Figure 4: Residual demand distribution network

5.2 Tiny Flow Removal

Our min-cost flow model for tiny flow removal is shown in Fig. 4. Each tiny flow f_{an} has a corresponding node RD_{an}, referred to as *demand node*. From source S, we add an edge to every demand node. Also, from each demand node RD_{an}, there is an edge to data center node DC_k if the latter has an instance of application a and $f_{ak} >= f_{an}$. By not including edges to data centers with smaller flows (note the absence of edges from RD_{an} to DC_1 and DC_{D-1} in Fig. 4), we enforce heuristic 1a. Finally, each data center node is connected to sink T.

For edges from source node S to demand node RD_{an}, the capacity is f_{an}, and the cost is 0 - this represents the demand to be satisfied. All edges from demand node RD_{an} to data center nodes have capacity f_{an} (this demand could potentially be satisfied by any of these data centers), and the edges from data center nodes to the sink have capacities equal to the residual capacity rc_n of each data center. For data center $DC_k \in U_a$, the cost of the edge from node RD_{an} to DC_k is 0 (since it already has an instance and we would like to assign as much demand as possible to these nodes – see the edge from RD_{an} to DC_2 in the figure). The cost of other edges is chosen in a way such that:

1. For any two tiny flows f_{an} and $f_{a'n'}$, if $f_{an} < f_{a'n'}$ then $cost_{i,j}$ of edges going from demand node RD_{an} to data center nodes in V_a is larger than $cost_{i',j'}$ of edges going from $RD_{a'n'}$ to data center nodes in $V_{a'}$. In this way, flow f_{an} would have an advantage over $f_{a'n'}$ when competing for residual capacity of data centers with instances of both applications, thus following heuristic 1b.
2. The cost of edges going from residual node RD_{an} to $DC_k \in V_a$ is inversely proportional to f_{ak}. In this way, the min-cost flow algorithm will try to follow heuristic 2.
3. For the edge from data center node DC_n to sink T, the cost is the number of normal flows h_n at data center DC_n. This makes the algorithm follow heuristic 3.
4. Because heuristics 1 and 2 have higher priority than 3, we make sure that the cost of edges from demand nodes RD_{an} to data center nodes in V_a dominates the cost of edges from data center nodes in V_a to the sink node. In Fig. 4, $C_{a,n} >> h_k$ and $C_{a,D} >> h_k$ for all

$k = 1, 2, ...D$.

After solving this problem, we remove all the tiny instances that became idle (assigned no demand).

5.3 Hysteresis Placement

As described so far, our scheme computes a new placement policy only based on the current demand distribution, regardless of the previous placement. This can result in large number of placement changes.

We propose *hysteresis placement* to control the number of placement updates. We introduce a parameter *hysteresis ratio (HR)* when categorizing flows. If application a is deployed at data center n in the previous placement, we consider f_{an} as tiny instance only when $f_{an} < \frac{DT}{HR}$, where $HR \geq 1$. In this way, if application a is currently deployed at data center n, then it is more likely to be kept in place in the new placement. This added "stickiness" may result in some increase in the number of application instances and some response time penalty. We evaluate these effects in § 6.5.

5.4 Further Fine-Tuning

Our scheme so far aggregates demand into flows from coarse-grained regions and does not distinguish between requests within a flow. So, when a flow is assigned to multiple data centers, rather than sending requests to these data centers at random, we can split this flow among its assigned data centers according to request proximity preferences as long as this does not violate the overall demand distribution. We skip details due to space limitation but provide them in the extended version of this paper [27]. Our evaluation study includes this optimization in all the experiments.

6 Evaluation

We study the performance of our approach using large-scale simulation built on CSIM [6], a discrete-event simulation package. Mimicking the actual system, our simulator has a decision component and a request routing component. The decision component periodically updates application placement and server selection policy (every 30s by default). There is also a workload component that generates requests according to load patterns discussed later. The routing component forwards each request to the appropriate data center according to the policy generated by decision component.

6.1 Cloud Model

We simulate a global cloud platform across 20 data centers hosting 100 applications (except for the scalability experiments in § 6.7). We parameterize our model as follows. We got all pingable IP addresses – 157803 total – from the Gnutella peer list compiled at the University of Oregon [3] and found their geographical locations using the GeoIP database (commercial version)[5]. We "deploy" our 20 data centers in countries according to their client distribution, i.e., nine data centers in US, three in China, etc. For the US, we use a similar procedure to distribute the data centers among states.

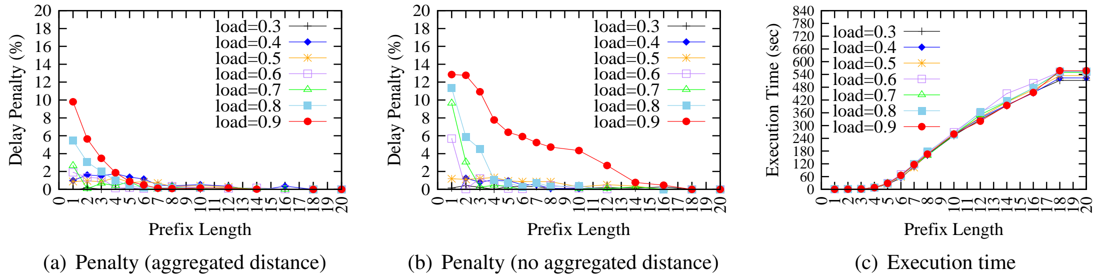
We then selected 20 PlanetLab nodes in the same locales as our data centers and measured ping latencies from each such PlanetLab node to each client. We were able to obtain complete distances to 100546 clients. We then used these clients to represent the locations of client clusters, the 20 PlanetLab nodes to mimic our data centers, and the measured ping latencies as the network distances. Since back-end databases are assumed to stay outside the cloud (see § 2), we randomly select 100 PlanetLab nodes to mimic the databases and use ping latencies from the 20 PlanetLab nodes representing data centers to these 100 PlanetLab nodes as distances between data centers and databases.

We divide the world into 20 geographic regions, each with a data center. Client clusters that share a common closest data center fall into the same geographic region with the data center. We also divide applications into two categories, regional and global. Regional applications are particularly popular within a specific geographic region (*hot region*), e.g., the website of a state government; global applications are universally popular. For a regional application, we define *regional rate* as the portion of requests it receives from its hot region. We use regional rate of 0.9.

6.2 Workload

Each data center can serve 10,000 requests peer second (req/s), resulting in the total capacity of all data centers of 200,000 req/s. These rates are dictated by the scalability of the simulator itself, but are sufficient to evaluate our approach. We define $load_factor$ as the ratio of the total request rate of all data centers to the total capacity. Each (normalized –see § 2) request is assumed to have service time 0.03 second, so every data center in the simulator has 300 CSIM facilities that mimic servers. We set the queue length of each facility to 150; requests are distributed among servers in a data center in a round-robin fashion and are dropped when arriving at a facility with full queue. In the optimization models, we assume the capacity watermark of 0.9, that is, the system tries to keep each data center utilization within 9,000 req/s.

Demand Generation. We assume applications' popularity follows Zipf law with parameter 1. The top application is global and the remaining 99 are regional, thus the global application generates around 20% of total demand. Given the target total request rate, r, determined by the load factor, the workload generates requests sequentially with exponentially distributed inter-arrival time with mean $t = 1/r$. For each request, it first



(a) Penalty (aggregated distance) (b) Penalty (no aggregated distance) (c) Execution time

Figure 5: Performance of prefix clustering

selects an application according to the power law probability distribution. Then if the selected application is regional, it assigns the request to a random client cluster from its hot region with probability of $regional_rate$ and to a randomly selected client cluster from outside its hot region otherwise. If the application is global, the request is assigned to a random client cluster.

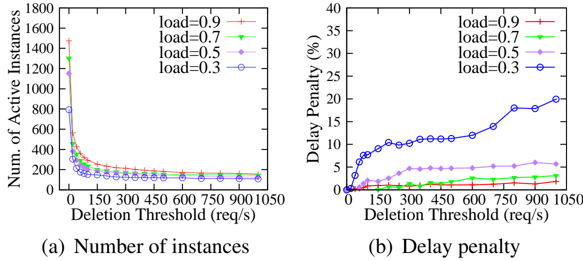
Dynamic Demand Patterns. During simulations, the demand pattern changes every T seconds. We use the following dynamic load patterns in our experiments:
1) *Vary-All-App:* starting from the initial distribution generated as described above, the demand for each application changes randomly within $\pm\Delta\%$, where Δ is a parameter controlling the extent of variability. This workload is an extended version of vary-all-apps in [33].
2) *Rank-Exchange:* popularity rankings of k randomly picked pairs of applications are swapped, where k controls the extent of demand variability. This workload mimics the change of popularity among applications.
3) *Reshuffle-All:* in each cycle, the rankings of the applications are reset to a random permutation and each regional application is remapped to a new random region. This workload mimics extreme case of change, where the demand pattern in each cycle is completely independent of the pattern in previous cycle.

6.3 Clustering Performance

We begin with the evaluation of permutation prefix clustering. In each experiment, we initially generate the requests that would occur in one second and re-send these requests repeatedly every logical second for ten logical seconds, at which point we recompute the demand to be used for the next ten seconds, and so on. While we use the same demand pattern, the demand will be different due to new random coin tosses during generation. To factor out the effects of stale demand data, the experiments in this subsection as well as § 6.4 and § 6.5 recompute the policy every time the demand is recomputed (every 10 second here) and use the upcoming demand data as input. We defer considering online policy computation based on prior demand until policy evaluation and prototype testing (§ 6.6 and § 7). The simulation lasts 50 logical seconds. To concentrate on clustering effect on server selection, all experiments in this subsection assume full deployment for each application, deletion threshold 0 req/s, and hysteresis ratio 1.

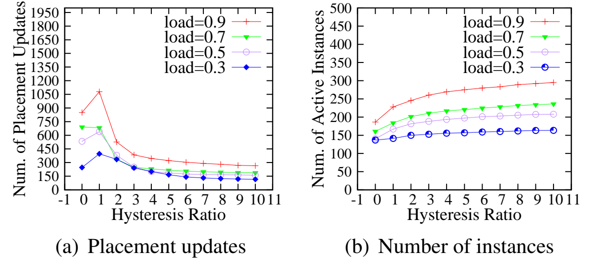
Fig. 5 shows performance effects as clustering level changes from the extreme case when only the closest server is considered (prefix 1) to no clustering (prefix 20, although no clustering occurred beyong prefix 18). We measure the number of dropped requests (although we did not observe any) and the average response time. Fig. 5a shows the response time penalty from clustering (also called *delay penalty* below), expressed as the relative difference between average response times with and without clustering. As seen from the figure, for a given level of clustering (i.e., the prefix length value), the penalty is smaller for lower load factors. (The line for load factor 0.4 deviates slightly from this trend for initial values of the prefix length. Since the penalty variations involved are very small - within 1% - we view it as a statistical aberration.) This makes sense because with low load, most demand is satisfied by the closest server, and the discrimination among more distant servers becomes unimportant. However, even for high loads, the clustering penalty is small, never exceeding 10%, and drops quickly with the prefix length. We attribute this to the effect of our distance aggregation for all members of the cluster: even when client-application pairs are clustered, their proximity to servers beyond the common prefix is still accounted for through aggregated distances. Indeed, Fig. 5b shows the delay penalty increases significantly when all distances to data centers beyond the prefix are assumed equal. Finally, Fig. 5c depicts the effect of clustering on the algorithm execution time. It shows that clustering trades these small delay penalties for a dramatic reduction in the execution time. For instance, for load factor 0.6, going from no clustering (prefix 20) to clustering with prefix 3 reduces the execution time from 552.4s to 2.3s, at the expense of only 1.3% delay penalty. We study the scalability of our approach further in § 6.7.

In summary, our experiments show that prefix clustering is a promising general technique for aggregating demand. Given these results, we use prefix size 3 for sub-



(a) Number of instances (b) Delay penalty

Figure 6: The effects of the deletion threshold



(a) Placement updates (b) Number of instances

Figure 7: The effects of the hysteresis ratio

sequent experiments, which allows us to solve the min-cost problem efficiently while keeping the delay penalty small - within 4% in the above experiments.

6.4 Deletion Threshold

We now study the deletion threshold (DT) effect. A higher DT tends to remove more instances but leads to greater performance penalty, as requests that used to go to the underutilized instances will now be routed to more distant data centers, while DT = 0 means no tiny instance removal, i.e., only completely idle instances are dropped. We use prefix 3 (see § 6.3) and hysteresis ratio 1 for these experiments.

Fig. 6 quantifies these effects by showing the total number of instances and delay penalty for different DT values. The workload is the same as in the previous subsection. Since each simulation run involves five recomputations of the demand, each data point represents the average total number of application instances across the whole run. The figure shows that as the deletion threshold increases, the number of total instances plunges in the beginning, but then decreases very slowly. The delay penalty behaves the opposite way, although at low load the penalty does not flatten. In general, this result indicates that with an appropriate deletion threshold, our scheme can drastically reduce the number of application instances with small performance penalty. We choose deletion threshold 150 req/s throughout our subsequent experiments as it obtains factor of 5-7 reduction in the number of application instances while keeping the delay penalty under 8% for all loads.

6.5 Hysteresis Placement Effects

We now evaluate the hysteresis ratio effects, using deletion threshold 150 req/s (see § 6.4) and prefix length 3 (see § 6.3) in the experiments.

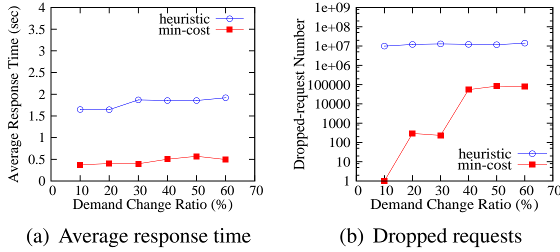
We use the following workload. At the first logical second, we generate a demand. At the next second, we remap the regional applications randomly to regions and recompute the demand. At all the subsequent seconds, we recompute the demand with new random coin tosses but keep the same pattern. So the workload changes dramatically in the second second, but keeps stable (except for statistical variations) in the remaining time. The application placement is computed every second: with our focus on placement changes, this allows us to shorten the experiment without affecting the results. The experiment lasts 20s. In Fig. 7, each data point represents the mean over five simulation runs with different seeds.

Fig. 7a shows the number of placement changes as the hysteresis ratio increases. For comparison, the figure also includes results for a heuristic application placement from [29] at 0 point on the x-axis. We see that with the increase of the hysteresis ratio, the number of placement updates drops but the total number of instances increases. When hysteresis ratio reaches 3, our approach results in *fewer* placement updates than algorithm from [29], even though the latter computes the new placement by adjusting the current configuration. Admittedly, as Fig. 7b shows, this comes at the expense of a certain increase in the number of instances, especially at higher load factors (a third more instances). We argue that this modest increase is justified by a significantly better performance of our approach, as we will see in the § 6.6. Interestingly, the delay penalty is negligible - less than 2% – and is not shown here. We use hysteresis ratio 3 for the rest of our experiments.

6.6 Policy Evaluation

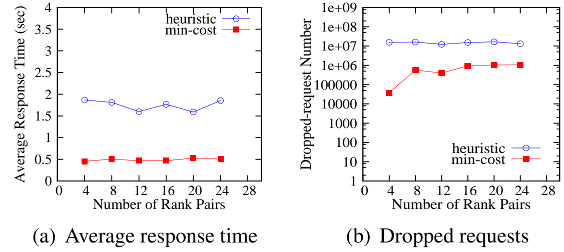
This section compares the quality of the policies produced by our approach and prior work. To our knowledge, the only works that jointly address the problems of demand distribution and application placement are [29] and [25]. Since our approach and [25] are not directly comparable ([25] aims at minimizing the replica load imbalance rather than optimizing the proximity), we compare our approach with [29]. The latter represents a drastically different approach from ours: it heuristically adjusts current placement by replicating or migrating instances and modifies server selection strategy according to the observed demand.

In these experiments, we use the dynamic load patterns in § 6.2 with load factor 0.5 and regional rate 0.9. Experiments start with full deployment and every request is forwarded to the closest data center. We generate the initial demand according to § 6.2. For the first



(a) Average response time (b) Dropped requests

Figure 8: Policy performance (Vary-All-App)



(a) Average response time (b) Dropped requests

Figure 9: Policy performance (Exchange-Rank)

15 logical seconds, the system is in a warm-up stage, where we update the policies every second so that the policies reflect the initial demand pattern after this stage. This is done for fairness to [29] as it adopts to the desired configuration incrementally. Then the system goes into the measurement stage, lasting 900 logical seconds, in which the demand is recomputed every 150 sec. according to the dynamic load pattern used, and the policies are updated every 30 sec. When computing the policies, we collect request rates through the statistics from all data centers as in reality. We use exponential moving average (smoothing factor 0.6) to maintain these statistics.

Fig. 8, 9 show the average response time and number of dropped requests for the two approaches, for the first two workloads (Reshuffle-All is not shown due to the space limit and is included in the extended version [27]). The curves corresponding to our approach and the approach of [29] are labeled, respectively, ”min-cost” and ”heuristic”. The results show dramatic performance advantage of our approach for both metrics. The average response time shows improvements at least by a factor of 2, and dropped requests reduce by orders of magnitude.

6.7 Scalability

We turn to the scalability of our approach. Our baseline setup of the system includes 100,546 client clusters, 20 data centers and 100 applications. We measure the execution time of our algorithms by increasing one of these parameters and keeping the other two constant. For the purpose of simulations, whenever we add a new entity to the setup and need a network delay between it and other entities, we pick the delay at random between 0 and 500ms. We utilize Dell PowerEdge 2950 server with 8 cores and 16G memory in the experiments.

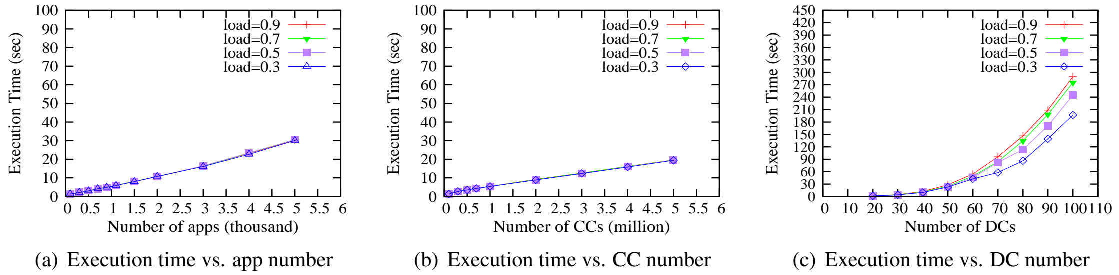
The results are presented in Fig. 10. They show that the execution time grows almost linearly with the number of client clusters and applications, but superlinearly with the number of data centers. The latter makes sense since, for the prefix length 3 we used, the size of the min-cost flow model of Fig. 3 grows as the power of 4 of the number of data centers (§ 4.2). Meanwhile, when the number of client clusters and applications increases, the size of the min-cost flow problem used in the first phase (optimization with full deployment) does not change, but the size of the problems used in the second phase (application placement) and in the flow-splitting phase increase linearly. As shown, the execution time remains within tens of seconds for thousands of applications, millions of client clusters, and tens of data centers.

We argue this reflects realistic platform sizes and acceptable execution time. Indeed, Krishnamurthy and Wang found roughly 400K client clusters on the Internet [19], and most infrastructure providers, such as Limelight and AT&T, operate up to 20-30 data centers. Execution time in the order of tens of seconds also seems acceptable: Oppenheimer et al., considering three real workloads, recommend application placement be done in the order of every 30 min. [24]; Wendell et al. found demand to be fairly stable on the 10-min. time scale [36].

7 Prototype

To demonstrate the operation of our system, we implemented our approach and deployed a testbed that mimics a global platform. We use five machines to emulate five data centers: one in Japan, one in UK, one in Australia, one in California and one in New York. We use another five machines to mimic the clients at these locations. To emulate global deployment, we hard-code the distances between the machines representing clients and data centers using measured ping RTTs between PlanetLab nodes in the mimicked locations. We used MyXDNS [10], a DNS server configurable with external server selection policies, as a request router.

In the prototype, a decision component collects utilization and demand distribution from data centers, periodically computes placement and server selection policies using our approach, and uploads the new server selection policy into MyXDNS. MyXDNS and our decision component both run on a separate machine, updating the policy every 30 seconds. On machines that mimic data centers, we install the WebSphere application server running the TPC-W benchmark (with the browsing mix workload) as the application. We set server capacity to 100 req/s and capacity watermark at 70%. Yet we report results in terms of actual server utilization, thus justifying (at least for this application) our assumption about



(a) Execution time vs. app number (b) Execution time vs. CC number (c) Execution time vs. DC number

Figure 10: Scalability of Policy Computation

feasibility of using request rates as measure of demand and utilization. We use the following two scenarios to demonstrate how our system responds to the dynamically changing demand.

7.1 Demand Shift

Our first scenario shows the ability of the system to handle demand shifts from one region to another. In this scenario, we generate requests from only one location at a time and at a level that a single data center can cope with, but we change the location every 120s.

Fig. 11 shows the CPU utilization of the five machines imitating data centers. It indicates that the system handles this scenario successfully. Indeed, the application placement follows the demand after the delay induced by the periodicity of policy updates. Only one instance of the application is deployed at a time except during transitions, since our prototype is careful not to enact instance deletion until it completes pending requests - this is seen from an overlap in utilization curves.

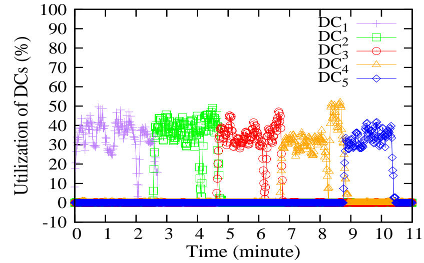


Figure 11: DC utilization with demand shift

7.2 Flash Crowd

Our second scenario imitates a flash crowd coming from one region. In this experiment, we generate requests from a single fixed location throughout the experiment but the amount of requests increases in the first 220s, then stays constant for 120s, and then drops in the final 220s. The application is initially placed in the data center in the region that generates the demand.

Fig. 12 shows the CPU utilization of the data centers in this scenario, again demonstrating successful operation of the system. Initially, data center DC_1, the nearest to client demand, is sufficient to handle the workload. As its utilization exceeds the watermark, the application is deployed at two more data centers - first at the second closest data center DC_2 and then at DC_3, the third closest. Once the flash crowd subsides, the system removes the application from the two distant data centers, first from DC_3 and then from DC_2. Note that during the flash crowd, the two closest data centers are utilized equally (up to their capacity watermark) and the more distant data center DC_3 receives only the overflow demand. Also note transient effects around 60 and 150 seconds, due to periodicity in policy recomputation (hence an inherent lag in reaction to changing demand) and an occasional unpredictable change in demand. E.g., at around 120 sec, the request rate produced by the demand generator unexpectedly dropped (not following the workload pattern), causing the system to lower selection probability of DC2. But right after that, the workload increased back to normal, leading to spike in utilization of DC1, while leaving DC2 only modestly utilized.

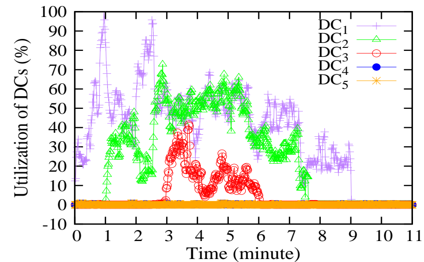


Figure 12: DC utilization with flash crowd

8 Related Work

While many efforts have addressed application placement and server selection, they mostly consider only one of these two problems. Schemes in [28, 22, 14, 15, 17, 18, 20] address the placement problem assuming requests are always forwarded to the closest replica. This makes these approaches suboptimal in practice as servers have limited capacity. Some works formulate global optimization problems [28, 14] but use them only as the ba-

sis for comparison since they are impractical due to computational cost. Our prefix clustering approach makes global optimization practical in many cases.

Other approaches focus on the server selection assuming a given set of replicas [11, 36, 30, 13, 31]. None of them take into account the distance between server and back-end database, partly because they mostly consider server selection for CDNs, which do not have this issue. In particular, [36] proposes an optimal decentralized server selection algorithm done by a set of mapping nodes. We address a joint placement and selection problem in a centralized manner but handle the scale issue through a novel prefix clustering technique. In [11], the authors use a min-cost flow model to generate the server selection strategy. However, they assume that the placement of applications is fixed while our approach includes the placement aspect. Furthermore, unlike our clustering technique, their approach to scalability depends on a fortuitous placement configuration.

Among the few works that tackle both placement and server selection, [25] proposes distributed placement and server selection algorithms. However, their server selection aims to balance load without considering proximity. In [29], the authors propose decentralized placement and centralized server selection algorithms that take into account both server load and proximity but compute both policies in isolation. Our unified approach showed performance advantages over it. None of them considers the distance between server and back-end database.

9 Conclusion and Future Work

This paper addresses a problem of efficient hosting of multiple applications in a globally distributed cloud computing platform and makes two main contributions. First, we design a unified approach for application placement and demand distribution policies and show its promise through both simulation experiments and a prototype testbed demonstration. Second, we propose a novel demand clustering technique and show that it makes policies based on global optimization models practical for realistic-size environments. We hope our clustering technique will be found useful beyond its application to the particular algorithms discussed here.

Important issues for future work include extending out approach to account for energy consumption, consider inter-dependencies among hosted applications, allow applications to have different priorities, and ensure pre-defined quality of service levels.

References

[1] http://download.oracle.com/docs/cd/e13222_01/wls/docs60/cluster/servlet.html.

[2] http://en.wikipedia.org/wiki/minimum_cost_flow_problem.

[3] http://mirage.cs.uoregon.edu/p2p/snapshots.html.

[4] http://www.igsystems.com/cs2/index.html.

[5] http://www.maxmind.com.

[6] http://www.mesquite.com/documentation.

[7] AHUJA, R., MAGNANTI, T., AND ORLIN, J. *Network flows: theory, algorithms, and applications*. Prentice hall, 1993.

[8] AKINC, U. Multi-activity facility design and location problems. *Management Science 31*, 3 (MAR 1985), pp. 275–283.

[9] AL-QUDAH, Z., ALZOUBI, H., ALLMAN, M., RABINOVICH, M., AND LIBERATORE, V. Efficient application placement in a dynamic hosting platform. In *WWW* (2009), pp. 281–290.

[10] ALZOUBI, H., RABINOVICH, M., AND SPATSCHECK, O. MyXDNS: A request routing DNS server with decoupled server selection. In *WWW* (2007), pp. 351–360.

[11] ANDREWS, M., SHEPHERD, B., SRINIVASAN, A., WINKLER, P., AND ZANE, F. Clustering and server selection using passive monitoring. In *IEEE INFOCOM* (2002), pp. 1717–1725.

[12] ARMBRUST, M., FOX, A., GRIFFITH, R., JOSEPH, A., KATZ, R., KONWINSKI, A., LEE, G., PATTERSON, D., RABKIN, A., STOICA, I., ET AL. Above the clouds: A Berkeley view of cloud computing. *UC Berkeley, Tech. Rep. UCB/EECS-2009-28* (2009).

[13] BAKIRAS, S. Approximate server selection algorithms in content distribution networks. In *IEEE ICC* (2005), pp. 1490–1494.

[14] BARTOLINI, N., PRESTI, F., AND PETRIOLI, C. Optimal dynamic replica placement in Content Delivery Networks. In *IEEE ICON* (2003), pp. 125–130.

[15] CIDON, I., KUTTEN, S., AND SOFFER, R. Optimal allocation of electronic content. In *IEEE INFOCOM* (2001), pp. 205–218.

[16] GOLDBERG, A. V. An efficient implementation of a scaling minimum-cost flow algorithm. In *J. Algorithms* (1997), Academic Press, Inc.

[17] JAMIN, S., JIN, C., KURC, A., RAZ, D., AND SHAVITT, Y. Constrained mirror placement on the Internet. In *IEEE INFOCOM* (2001), pp. 31–40.

[18] JIA, X., LI, D., HU, X., AND DU, D. Placement of read-write web proxies in the internet. In *IEEE ICDCS* (2001), pp. 687–690.

[19] KRISHNAMURTHY, B., AND WANG, J. On network-aware clustering of web clients. In *ACM SIGCOMM* (2000), pp. 97–110.

[20] KRISHNAN, P., RAZ, D., AND SHAVITT, Y. The cache location problem. *IEEE/ACM ToN 8*, 5 (2000), 568–582.

[21] LAGAR-CAVILLA, H., WHITNEY, J., BRYANT, R., PATCHIN, P., BRUDNO, M., DE LARA, E., RUMBLE, S., SATYANARAYANAN, M., AND SCANNELL, A. SnowFlock: Virtual Machine Cloning as a First-Class Cloud Primitive. *ACM TOCS 29* (2011), 2:1–2:45.

[22] LI, B., GOLIN, M., ITALIANO, G., DENG, X., AND SOHRABY, K. On the optimal placement of web proxies in the internet. In *IEEE INFOCOM* (1999), pp. 1282–1290.

[23] MARLER, R., AND ARORA, J. Survey of multi-objective optimization methods for engineering. *Structural and Multidisciplinary Optimization 26*, 6 (2004), 369–395.

[24] OPPENHEIMER, D., CHUN, B., PATTERSON, D., SNOEREN, A., AND VAHDAT, A. Service placement in a shared wide-area platform. In *USENIX ATC* (2006), pp. 26–26.

[25] PRESTI, F., PETRIOLI, C., AND VICARI, C. Distributed dynamic replica placement and request redirection in Content Delivery Networks. In *ACM/IEEE MASCOTS* (2007), pp. 366–373.

[26] QIAN, H., MILLER, E., ZHANG, W., RABINOVICH, M., AND WILLS, C. E. Agility in virtualized utility computing. In *VTDC*, pp. 9:1–9:8.

[27] QIAN, H., AND RABINOVICH, M. Application placement and demand distribution in a global cloud platform: A unified approach. Available at http://engr.case.edu/qian_hangwei/files/tech_report_global.pdf.

[28] QIU, L., PADMANABHAN, V., AND VOELKER, G. On the placement of web server replicas. In *IEEE INFOCOM* (2001), pp. 1587–1596.

[29] RABINOVICH, M., XIAO, Z., AND AGGARWAL, A. Computing on the edge: A platform for replicating internet applications. In *WCW* (2003), pp. 57–77.

[30] RANJAN, S., KARRER, R., AND KNIGHTLY, E. Wide area redirection of dynamic content by Internet data centers. In *IEEE INFOCOM* (2004), pp. 816–826.

[31] SAYAL, M., BREITBART, Y., SCHEUERMANN, P., AND VINGRALEK, R. Selection algorithms for replicated web servers. *ACM SIGMETRICS Performance Evaluation Review 26* (1998), 44–50.

[32] STEWART, C., AND SHEN, K. Performance modeling and system management for multi-component online services. In *USENIX NSDI* (2005), pp. 71–84.

[33] TANG, C., STEINDER, M., SPREITZER, M., AND PACIFICI, G. A scalable application placement controller for enterprise data centers. In *WWW* (2007), pp. 331–340.

[34] TESAURO, G., JONG, N., DAS, R., AND BENNANI, M. A hybrid reinforcement learning approach to autonomic resource allocation. In *USENIX ICAC* (2006), pp. 65–73.

[35] URGAONKAR, B., SHENOY, P., AND ROSCOE, T. Resource overbooking and application profiling in shared hosting platforms. In *USENIX OSDI* (2002), pp. 239–254.

[36] WENDELL, P., JIANG, J., FREEDMAN, M., AND REXFORD, J. DONAR: decentralized server selection for cloud services. In *ACM SIGCOMM* (2010), pp. 231–242.

[37] ZHANG, W., QIAN, H., WILLS, C., AND RABINOVICH, M. Agile resource management in a virtualized data center. In *ACM WOSP/SIPEW* (2010), pp. 129–140.