Providing Geo-Elasticity in Geographically Distributed Clouds

Tian Guo, Worcester Polytechnic Institute
Prashant Shenoy, University of Massachusetts Amherst

Geographically distributed cloud platforms are well suited for serving a geographically diverse user base. However traditional cloud provisioning mechanisms that make local scaling decisions are not adequate for delivering best possible performance for modern web applications that observe both temporal and spatial workload fluctuations. In this paper, we propose GeoScale, a system that provides geo-elasticity by combining model-driven proactive and agile reactive provisioning approaches. GeoScale can dynamically provision server capacity at any location based on workload dynamics. We conduct a detailed evaluation of GeoScale on Amazon's geo-distributed cloud, and show up to 40% improvement in the 95^{th} percentile response time when compared to traditional elasticity techniques.

CCS Concepts: •**Networks** → **Cloud computing;** •**Mathematics of computing** → *Queueing theory;* •**Social and professional topics** → *Pricing and resource allocation;*

Additional Key Words and Phrases: Geo-distributed clouds, dynamic resource management

ACM Reference Format:
Tian Guo, and Prashant Shenoy, 2017. Providing Geo-Elasticity in Geographically Distributed Clouds. *ACM Trans. Internet Technol.* X, Y, Article Z (January 2016), 25 pages.
DOI: 000000X.000000X

1. INTRODUCTION

Today's cloud platforms provide numerous benefits to hosted applications such as a pay-as-you-go cost model and flexible, on-demand allocation of resources. Since many internet applications see a dynamically varying workload, a key benefit of a cloud platform is its ability to autonomously and dynamically provision server resources to match a time-varying workload—a property that we refer to as *elasticity*. Cloud platforms such as Amazon EC2 support elasticity in the form of "auto-scaling" [Amazon Auto Scaling Service 2013] where an application provider can choose thresholds on any system metric to automatically scale server capacity *when* monitored metrics violate pre-specified thresholds.

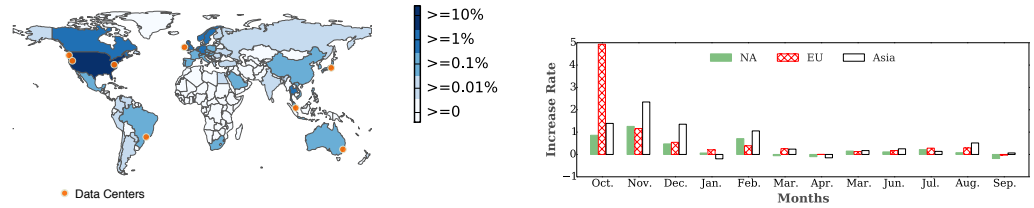
Modern cloud platforms are becoming increasingly distributed and offer a choice of multiple geographic sites and data centers to application providers to host their applications. A geographically distributed cloud that offers a choice of multiple locations to host cloud applications provides two key benefits. First, an application provider can choose a location that is closest to its user base to optimize user-perceived performance. Second, for those applications that have its users spread across multiple geographic regions, such a geo-distributed cloud offers the possibility of hosting application replicas at multiple cloud sites so that users in a region can be serviced from the closest application replica.

An internet application that services a geographically diverse user base is subject to workload dynamics that exhibit both *temporal* and *spatial* variations. Temporal variations include fluctuations

---

This paper is an expanded version of a four-pages WIP paper [Guo et al. 2016] that appeared in IEEE International Conference on Cloud Engineering (IC2E) 2016. The majority of this work was done when Tian Guo was at University of Massachusetts Amherst. This work is supported by the National Science Foundation, under grant 1229059, grant 1345300 and grant 1422245.
Author's addresses: Tian Guo, Department of Computer Science, Worcester Polytechnic Institute; Prashant Shenoy, College of Information and Computer Sciences, University of Massachusetts Amherst.

DOI: 000000X.000000X



(a) Heat map of Gowalla's geographically distributed web traffic comprising of 6.5 million user check-ins from Feb. 2009 to Oct. 2010.

(b) Temporal and spatial variations exhibited by Gowalla check-ins dataset from Oct. 2009 to Oct. 2010.

Fig. 1: Geo-dynamics seen by the Gowalla social media application's check-ins data.

over multiple time-scales such as time-of-day effects, seasonal time-of-year effects as well as sudden load spikes due to flash crowds. Spatial variations are independent workload spikes that are specific to geographic regions. Spatial variations in the workload occur since the application may be more popular in one region (e.g., country) than another, the user growth may differ from one region to another, and regional events (e.g., local festivals or local news stories) may cause spikes in one region without impacting the workload in other regions. These spatial effects are depicted in Figure 1, which shows the client distribution of Gowalla, a popular social application, and illustrates the global client base of such applications [Cho et al. 2011]. Figure 1a shows that the application is more popular in certain regions such as North America and Europe and less popular in regions such as Asia. Figure 1b shows the month-to-month growth of users seen by the application. It shows that growth from users in Asia is higher than users growth in other regions. The figure also shows that the rate of growth fluctuates from month to month; for example, there is a spike of European users in October, while there are some months where the number of users from some region sees a negative growth (i.e., a decline) while other regions see positive growth. While techniques for handling temporal variations in user traffic have been studied [Singh et al. 2010], provisioning techniques for handling both temporal and spatial fluctuations in the workload are needed for applications with a geographically diverse user base.

In this paper, we argue that a geographically distributed cloud platform should support *geo-elasticity* to efficiently meet the need of internet applications that observe *geo-dynamic* workloads, i.e. workloads that exhibit both temporal and spatial variations. Geo-elasticity enables the cloud platform to autonomously vary the cloud locations hosting application replicas as well as vary the number of application servers at each location to handle *both* temporal and spatial variations in the workloads. Much of the prior work on dynamic provisioning focuses on local elasticity mechanisms within a single data center [Lim et al. 2010; Singh et al. 2010; Shen et al. 2011], which is sub-optimal for geo-dynamic workloads *since it does not allow the set of cloud locations that host the application to vary dynamically*. We design a system called GeoScale to addresses these challenges. In doing so, our paper makes four contributions:

First, we present a geographic profiling and forecasting technique that monitors the workload of a cloud application, geographically clusters the workload into the cloud locations, and employs time-series forecasting to predict the future workload across regions.

Second, we present a new geo-elasticity technique that can handle dynamics in both the volume and geographic distribution of application workloads. At the core of our approach is a queuing-theoretic model, seeded with empirical measurements, that determines the server capacity needed at each cloud location.

Third, we design proactive and reactive algorithms for geo-elasticity that incorporate our workload forecasting and capacity modeling techniques. Broadly our proactive approach provisions capacity at longer time scales while our reactive approach is able to handle capacity allocations for unpredictable or unexpected workload changes.

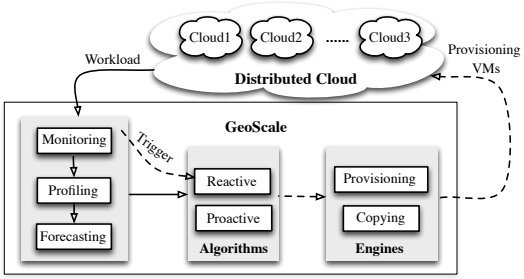


Fig. 2: **GeoScale architecture.** GeoScale is comprised of three key components, 1) workload monitoring, profiling, and forecasting engine, 2) model-driven proactive and reactive geo-elastic provisioning algorithms, and 3) geo-elastic cloud provisioning and copying engine.

Fourth, we present a prototype implementation of GeoScale and conduct a detailed experimental evaluation of our system by using Amazon EC2 distributed cloud to run representative applications and PlanetLab nodes to inject a geographically diverse workload. Our experimental results show 13% to 40% improvement in the 95^{th} percentile response time when compared to traditional elasticity techniques. With pre-copying optimizations, GeoScale achieves fast provisioning of new server capacity in tens of seconds.

2. BACKGROUND AND PROBLEM STATEMENT

In this section, we first present background on geo-distributed clouds and the application model assumed in our work, followed by a description of the geo-elasticity problem addressed in the paper.

Geographically distributed Clouds. Our work assumes a geo-distributed cloud platform that comprises data centers from different locations. The cloud platform considered is an Infrastructure-as-a-Service (IaaS) public cloud that provides virtualized compute and storage resources, in the form of virtual machines (VMs), to its customers. We assume that the cloud exposes application programming interface (API) to customers to request, start, stop, and terminate servers at a specific location; customers do not have direct access to the underlying hypervisor on a physical server and must manage their VMs through the cloud's API. We assume the cloud platform has the capability to monitor and analyze the incoming workload of an application to determine both the geographic distribution and the temporal variations in the workload.

Application Model. Our work targets multi-tier web-based applications that service a geographically distributed client base. The application employs a front-end tier that implements the application logic and a back-end tier that stores application data, often in a database. We assume the application—either with both tiers deployed inside a single VM or separate VMs—is designed to be replicable. That is, we can spawn multiple VMs, each housing a replica of the application (or one of its tiers), to host such applications. We further assume that the task of maintaining consistency among back-end replicas is handled by applications using any method that suits their needs. For example, an application can draw upon techniques [Pujol et al. 2010; Kraska et al. 2013; Patiño Martinez et al. 2005] used by master-slave databases, multi-master databases, or database middleware systems that offer a spectrum of tradeoffs between costs and performance.

(Geo-)Elasticity. In this work, we consider cloud-based elasticity mechanisms that autonomously provision the server resources on behalf of an application. We look at both model-based and reactive provisioning techniques that enable horizontal scaling by dynamically replicating the application VMs. For reactive provisioning, application providers are assumed to specify thresholds on performance metrics such as request rate, response time or server utilization. The cloud platform then monitors any threshold violations to allocate or deallocate server capacities within each data center. We refer to this traditional form of elasticity as *local elasticity*. However, local elasticity is suboptimal for applications that have geographically-diverse user base since it fails to consider spatial

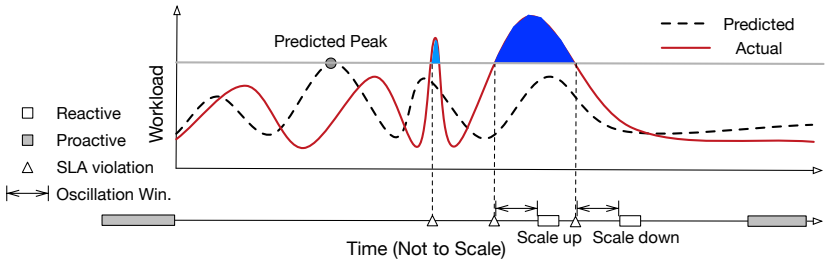


Fig. 3: **GeoScale hybrid provisioning illustration.** GeoScale proactively provisions server resources based on the the predicted workload peak in the coming provision period. Within each proactive provisioning window, reactive engine monitors for SLA violations. When SLA violations are detected (blue shaded areas) and last more than oscillation window (darker blue area), GeoScale uses reactive algorithm to scale up server resources.

workload variations. We propose *geo-elasticity*, a mechanism that dynamically provisions server resources at any geographic location when needed by taking both temporal and spatial workload variations into consideration. Geo-elasticity enables better user-perceived performance by allowing resources to be provisioned *closer* to clients, rather than being constrained by the current set of cloud sites that host an application. The goal of our work is to design and implement such a geo-elasticity technique into a distributed infrastructure cloud platform; while we currently implement our approach as a cloud middleware, our techniques are easily integrated into the cloud platform fabric. We also compare our approach with a manual approach of choosing cloud sites and a Content Delivery Network(CDN) based approach.

Formally, the geo-elasticity problem can be stated as: given an application servicing clients $C = \{c_1, c_2 \ldots c_n\}$, we wish to provision a set of servers $S = \{s_1, s_2 \ldots s_p\}$ among a set of cloud locations $L = \{l_1, l_2 \ldots l_k\}$ such that an application-specified SLA metric is satisfied and the average client network latency is minimized.

2.1. GeoScale System Architecture

We design GeoScale, as depicted in Figure 2, to provide geo-elasticity in geographically distributed clouds. GeoScale is implemented as a middleware layer that uses cloud API to programmatically provision servers on behalf of cloud applications to handle workload dynamics. The workload monitoring, profiling and forecasting engine is responsible for monitoring the incoming request, creating a geographic profile of the workload using clustering techniques, and then employing time-series forecasting techniques to predict the future workload based on the recent history. GeoScale supports two provisioning algorithms: the proactive provisioning algorithm handles long-term provisioning based on future forecasts, while the reactive provisioning algorithm reacts to short-term workload dynamics, unexpected workload spikes and even forecast errors, all of which may need additional capacity, beyond that provisioned by the proactive method. The algorithms are model-driven and use queueing models to provision sufficient capacity to meet the application-specified SLA, by capturing the relationship between system resources and application performance. Finally both the copying and provisioning engines use cloud API to perform on-demand data copying or lazy pre-copying, to deploy servers at chosen cloud sites or make adjustments to the number of servers.

We describe the design and implementation of three key components of GeoScale in Section 3 and Section 4, followed by a detailed experimental evaluation in Section 5.

3. PROVIDING GEO-ELASTICITY USING GEOSCALE

GeoScale uses a hybrid provisioning approach—proactive and reactive provisioning—to handling workload fluctuations, as illustrated in Figure 3. At the end of each provisioning period, GeoScale

deactivates its reactive engine and starts its proactive one that provisions enough resources in each data center based on the predicted peak workload. Proactive provisioning can take up to a couple hours if it involves steps such as copying application data across geographically distributed data center locations, or warming up in-memory cache [Guo et al. 2013; Scryer 2013]. Therefore, it is crucial to mask the lengthy provisioning time by proactively allocating resources before the workload spike arrives.

After the proactive engine has successfully setup application servers, GeoScale then re-activates the reactive engine that monitors SLA violations. Upon detecting a violation, the reactive engine will trigger reactive provision only if the violation lasts long enough, i.e., longer than a predefined oscillation window. As a result, only the second violation causes additional resources being provisioned, as shown in Figure 3. The use of an oscillation window is a common technique [Wood et al. 2007] to restrict actions to lasting behaviors. We leverage it to prevent the reactive engine from scaling up and down resources for temporary workload fluctuations.

When workload drops below a predicted peak for longer than the oscillation window, the reactive engine will stop all the servers it started. However, the reactive engine will not scale down resources provisioned by the proactive engine. As a result, over provisioned resources by the proactive engine are wasteful, i.e. incurring unnecessary cloud bills. However, such scenarios can be mitigated by tuning the provision period—a smaller window allows the proactive engine to adjust server allocations more frequently and make better workload predictions.

3.1. Workload Monitoring, Profiling and Forecasting

Workload Monitoring. We assume that GeoScale has access to a log of the application's incoming requests at each data center location that houses a replica of the application. The incoming request stream at each site can be logged either at a load balancing switch that distributes incoming requests to front-end replicas, or can be constructed by periodically aggregating request logs directly from each front-end replica (e.g., apache web server logs). We assume that request logs contain information such as a time-stamp, client IP address, requested URL, service time and response time seen by that request.

Geographic Workload Clustering. Given a request trace from each site, GeoScale first translates each client IP address to the client's geographic location using an IP geolocation technique.[1] The client locations are then mapped to pre-configured *geographic bins* that are specified by the application provider. Each bin represents a certain geographic region and could be configured at different granularity based on the level at which the workload needs to be monitored. For instance, a bin may represent an entire city such as Los Angeles, or a state such as Massachusetts, or larger regions such as countries or even continents. GeoScale uses the specified bins to track the workload from the corresponding geographic regions. Next, GeoScale calculates the geographic distances d, as a proxy for network distance [Ng and Zhang 2002], between each bin—represented by a weighted center of all requests within the bin—and data center pair by using the Heversine formula in Equation (1).

$$\begin{aligned} a &= \sin^2(\frac{\Delta\phi}{2}) + \cos\phi_1 \cdot \cos\phi_2 \cdot \sin^2(\frac{\Delta\varphi}{2}) \\ d &= 2R \cdot \arcsin(\min(1, \sqrt{a})), \end{aligned} \tag{1}$$

where ϕ_i and φ_i represent the latitude and longitude of location i and R is the mean radius of the Earth. This yields a sorted list of data centers, in ascending order of distance, for each geographic bin. Note, unlike network distance, obtaining geographic distances d does not require explicit client participations.

The iterative greedy clustering then proceeds by assigning each geographic bin (and its traffic volume) to the closest cloud data center. Once all bins have been mapped to a data center, the clustering algorithm sums the application traffic assigned to each data center. If a data center does

[1] GeoScale uses Maxmind [Maxmind GeoIP Service 2013] GeoIP location service.

not receive sufficient traffic (less than a threshold τ) to justify deploying an application replica, the data center is removed from the sorted list of all bins. The mappings between geographic bin and data center are updated with the new sorted lists. This greedy iterative clustering process continues until each bin has a mapping to the closest possible data center such that all data center locations have at least a threshold τ amount of traffic mapped onto them.

We also formulate the problem of determining a mapping X_{ij} between a geographic bin i and a cloud location j as an Integer Linear Program(ILP) as follows:

$$\text{Minimize} \sum_{i=1}^{M}\sum_{j=1}^{K} w_i X_{ij} D_{ij}$$

Subject to

$$\sum_{i=1}^{M} X_{ij} T_i \geq \tau \quad \forall j = 1 \ldots K \tag{2}$$

$$\sum_{j=1}^{K} X_{ij} = 1 \quad \forall i = 1 \ldots M \tag{3}$$

$$X_{ij} \in \{0,1\} \quad \forall i = 1 \ldots M, j = 1 \ldots K \tag{4}$$

$$X_{ij} P_j \leq P \quad \forall i = 1 \ldots M, j = 1 \ldots K \tag{5}$$

where X_{ij} is a binary variable that is set to 1 if i^{th} bin is served by j^{th} cloud location and set to 0 otherwise. D_{ij} denotes the distance between i^{th} bin and j^{th} cloud location, T_i represents i^{th} bin's traffic volume, and P_j specifies the maximum amount of time needed to provision server resources in j^{th} cloud location. P_j can take values ranging from minutes to hours [Mao and Humphrey 2012; Sharma et al. 2015] depending on factors such as database availability, and instance types. Further, P represents the upper bound of allowable provisioning duration, and needs to be predefined by application owners. Constraint (2) ensures the chosen cloud location receive a threshold τ amount of traffic; Constraints (3) and (4) together ensure that each bin is mapped to one and only one cloud site. Constraint (5) restricts the set of cloud locations GeoScale can select from and is only used for global reactive provisioning (see Section 3.3). The objective function minimizes the weighted latency among all regions by assigning weights w_i that are proportional to T_i.

Workload Forecasting. The final step is to employ workload forecasting to determine the future (peak) workload that will be seen by each cloud site based on the above mapping. Any workload prediction or forecasting technique can be used for this purpose [Hellerstein et al. 1999]. GeoScale uses a simple approach that aggregates the workload traces from each bin that is mapped onto a cloud location to determine an aggregate workload trace for that location. The trace yields a time series of previous request rates and the time series can be modeled as an ARIMA time series to forecast the future load; the peak of the future predictions is chosen as the maximum load that will be seen by that cloud location for the application. An alternate approach is to combine the request traces from all bins mapped to the cloud location to generate an aggregate request trace and compute a workload distribution from this aggregated trace. The workload distribution is a probability distribution of request rates and the likelihood of seeing that request volume. A high percentile (e.g., the 95^{th} or the 99^{th} percentile) of this distribution can be chosen as the peak workload likely to be seen by this data center based on past observations. In either case, we obtain a prediction λ_j^p of the peak workload that will be seen by j^{th} cloud location and the provisioning algorithm must then provision sufficient capacity to handle this workload and meet the application's SLA.

3.2. Proactive Geo-Elastic Provisioning

GeoScale supports two types of geo-elastic provisioning, i.e., proactive and reactive, to handle workload dynamics at different time scales. In this section, we first describe proactive provisioning that operates periodically at longer time scales of hours or days and provisions server capacity to handle long-term workload trends observed at these time scales. We the reactive provisioning next in Section 3.3.

Varying Cloud Locations. Proactive provisioning uses the workload predictions from the previous section to drive the provisioning algorithm. Note that the previous step maps workloads from each geographic region or bin to the closest possible cloud location. In a scenario where an application rises in popularity in a new geographic region, the newly observed workload may get mapped to a new cloud location, causing proactive provisioning to start up one or more servers at this new location. Similarly a diminishing workload from a region may cause servers at a current location to be shut down, with the residual traffic from that region redirected to the closest active cloud location. Thus, proactive provisioning provides geo-elasticity by using observed changes in the spatial distribution of the workload to vary the number of *cloud locations* that house replicas of the application. This is in addition to handling changes in the temporal distribution in load at existing locations, which is handled by scaling the number of servers at those sites up or down.

Deriving Server Capacity. GeoScale employs a model-driven approach for its proactive provisioning algorithm. To determine the server capacity (i.e., number of servers) required at each location, let λ_j^p denote the peak workload that will be seen by this location j as per the workload forecasting engine. We employ a $G/G/1$ queueing model of an individual server to determine the maximum request rate λ_j^c that can be serviced by a *single cloud server* without violating the application's SLA. Modeling for individual server provides us the flexibility to employ heterogeneous resources. That is, if we were to use $G/G/n$ to model a network of n servers that share a request queue, the corresponding performance model can only help us reason about provisioning using homogeneous servers.

We use Kingman's theorem [Kingman 1961] for G/G/1 queue under heavy traffic. The theorem states that waiting time W is an exponential distribution with mean $E[W] = \frac{\sigma_a^2+\sigma_b^2}{2(\frac{1}{\lambda}-\bar{x})}$; where σ_a^2 and σ_b^2 denote the variance in the requests inter-arrival time and service time, and λ and $\bar{x}$ represent the request arrival rate and mean service time seen by this queueing system. Here, SLA y is defined as the 95^{th} percentile of server response time. That is, if the $y = 2$ seconds, the 95^{th} percentile of response time T, denoted as $\alpha_T(95)$, has to satisfy $\alpha_T(95) < 2$. We then derive the upper bound on the maximum rate λ_j^c under heavy traffic as shown in Equation (6). A sketch of the derivation is provided in Appendix A and additional details can be found in [Guo et al. 2015].

$$\lambda_j^c < \left[\bar{x}_j + \frac{3({\sigma_{ja}}^2 + {\sigma_{jb}}^2)}{2\,(y-\bar{x}_j)}\right]^{-1} \tag{6}$$

Obtaining Server Statistics. GeoScale uses empirical measurements from the workload monitoring component to estimate the variance of inter-arrival times and service times of requests ${\sigma_{ja}}^2$ and ${\sigma_{jb}}^2$. The request service times $\bar{x}_j$ at location j can be computed from server logs or measured by profiling the application on the server; if location j is a new location with no previous history, the observed service time from an existing nearby cloud location can be used as the initial estimate. The SLA y is specified by the application provider as the upper bound of response time that should not be violated, in our case 95^{th} percentile of server response time. Since all terms of Equation (6) are either known or empirically measured, we successfully obtain the maximum request rate λ_j^c that can be handled by a single server.

Calculating Server Numbers. We calculate the number of servers S_j required at location j to handle a peak request rate of λ_j^p in Equation (7). If S_j is greater than the current number of servers

$\hat{S_j}$ provisioned at location j, then the provisioning algorithm needs to scale up capacity by allocating $(S_j - \hat{S_j})$ additional servers. If $S_j < \hat{S_j}$, then capacity is scaled down by deallocating $(\hat{S_j} - S_j)$ servers. If $\hat{S_j} = 0$, this is a newly chosen cloud location for the application, and S_j new servers need to be started up at this location j. At the end of proactive provisioning, $\hat{S_j}$ is set to S_j,

$$S_j = \left\lceil \frac{\lambda_j^p}{\lambda_j^c} \right\rceil . \tag{7}$$

3.3. Reactive Geo-Elastic Provisioning

GeoScale employs reactive provisioning to make agile changes to the long-term capacity provisioned by the proactive algorithm—to handle any unexpected workload surges or to correct for errors in the predictions used by the proactive algorithm. GeoScale supports two forms of reactive provisioning: local and global. *Local reactive* provisioning is the simpler of the two and is used to make *local* adjustments to the capacity at a local cloud site—independently of application replicas at other locations. Local reactive provisioning is useful when there are changes in the volume of workload arriving at a cloud location but the overall geographic distribution of the workload remains (mostly) unchanged. Local reactive provisioning is triggered when response time SLAs are violated at one or more local cloud servers. GeoScale then estimates the new workload λ_j^n arriving at the current location j and determines the new server capacity needed to handle this workload as $S_j^n = \lceil \frac{\lambda_j^n}{\lambda_j^c} \rceil$ and provisions an extra $(S_j^n - \hat{S_j})$ servers for the application.

GeoScale also supports a more sophisticated *global reactive* provisioning approach that takes a holistic view of all the cloud locations upon being triggered, rather than just making capacity changes at a single local site. Global reactive provisioning is useful when the application experiences a sudden change in its geographic workload distribution (possibly in addition to changes in request volumes). GeoScale's global reactive provisioning algorithm requests the newly observed workload from the workload monitoring engine and obtains the current request rates emanating from each geographic region (or bin). It then checks for deviations in the workload observed in each geographic regions and the previously predicted peak workload for that region—to determine which region caused the SLA violations to occur. The algorithm then provisions new server capacity for the extra workload at a cloud site that is closest to each such region. Observe that this may cause the reactive approach to spawn new replicas at new cloud locations that did not previously host the application (yielding reactive geo-elasticity). Since reactive provisioning must be agile (to handle SLA violations that are already occurring), the ability to quickly provision new capacity at a new cloud location depends on whether the VM images (and data) for the applications are already precopied to that region. If images or data are not available, it may be faster to provision capacity at another nearby location that already houses an application replica, rather than at the optimal cloud site (since application images and data are already present at that site, new servers can be quickly provisioned). In essence, when making decisions in the global reactive mode, GeoScale takes into account provisioning time and only selects those data centers that can spawn application replicas fast (refer to constraint (5)). As noted below, GeoScale pre-copies application VM images and data in the background to new locations to reduce the replica startup times for proactive and reactive provisioning—which allows for global reactive provisioning to be more effective in such scenarios.

3.4. Agile Provisioning Using Precopying

The goal of GeoScale's provisioning engine is to actually scale server capacity up or down as dictated by the proactive or reactive algorithms. To do so, the provisioning engine assumes that the virtual machine disk image and copies of all network storage volumes holding application data are available to it. To add a new server to an application, the provisioning engine first uses the cloud API to start a new server from the VM image. Once the server starts up, any additional network stor-

age volumes may be attached to the server (by first making a snapshot and a copy of the additional storage volume). The newly started server is then added to the pool of application replicas at that site—for instance, by adding it to the pool of servers that are served by a front-end load balancing switch.

The latency involved in provisioning a new server for the application depends on whether the VM image and disk data are already available at the desired cloud site. If so, provisioning times depend on the time needed by the cloud platform to spin up a new server, followed by a small configuration overhead. If not, the VM image and application data must first be copied over from another cloud site where it is available—in this case, provisioning latency is dominated by cross-data center data copying overheads, which can be substantial if the application disk state comprises tens or hundreds of gigabytes of data. To reduce such latencies, GeoScale employs a lazy pre-copy optimization where it takes periodic snapshots of application's VM disks and lazily copies them in the background to new sites that are likely to house this application's replicas in the future. The choice of future sites requires manual input from the application provider or the use of historical workload information to pick geographic regions that are seeing steady workload increases.

The pre-copying process involves two steps: (i) GeoScale takes an application-agnostic snapshot of its disk volume and periodically takes incremental snapshots of subsequent changes and lazily transfers these snapshots to each potential future cloud site in the background; (ii) block-level deltas are used, in place of sending entire blocks, when disk blocks see small changes to their data. In either case, if data has already been pre-copied to a location, the VM disk state can be quickly reconstructed by transferring any incremental changes since the most recent pre-copying and a new server is started using this reconstructed disk image. If pre-copied data is not available, GeoScale initiates a full transfer of the VM disk image and any other application data volumes to the new site and starts up the new server once the data has been copied to the new site. The latter method is acceptable for proactive provisioning since it allocates capacity over long time-scales, but may not be desirable for agile reactive provisioning that needs to quickly scale up capacity. Therefore, when data is not available in the closest data center during global reactive provisioning, GeoScale may choose to start up new servers in other nearby locations where VM disk images are already available, rather than copying data. Specifically, the locations are selected using the ILP from Section 3.1. As noted in the ILP constraint (5) and objective function, the subset of eligible data centers are the ones that can finish provisioning servers within a given time threshold; and the specific locations are then chosen to minimize weighted latency among all regions.

3.5. Discussion

Hybrid vs. reactive. GeoScale explores a hybrid approach of using reactive and proactive provisioning in geographically distributed cloud locations. Prior research [Urgaonkar et al. 2005; Shen et al. 2011] has shown the benefits of combining the two approaches for local elasticity. In addition, industry and companies are also accepting such hybrid provisioning approaches. For example, Netflix [Scryer 2013] has argued that combining proactive provisioning with Amazon's auto-scaling (a reactive approach) works better for them. As noted earlier in the paper, provisioning and configuring an application server can take up to a few hours in the geo-elasticity case—such sustained, multiple hours of SLA violations may not be acceptable to many applications. So the benefits of applying a hybrid approach for geo elasticity are even greater when compared to local elasticity.

Next, we would like to explain why a reactive approach alone may not be the best strategy. A pure reactive approach inherently acts after a workload increase is observed. In scenarios where the workload is rising quickly, this may cause a temporary violation of SLA until new capacity is provisioned. In contrast, a proactive approach attempts to predict future workload increases (through workload forecasting) and provisions new capacity *before* the workload begins to increase. This avoids transient SLA violations for the application.

While the reactive approach may be adequate for situations with slow workload growth and quick provisioning, it has limitations for workload spikes. The situation is exacerbated for geo-elasticity since provisioning time can be orders of magnitude longer, e.g., when trying to copy state to a

remote data center and then start servers at the new location. This can increase the duration for which SLA violations are observed.

GeoScale's use case scenarios. GeoScale supports both proactive and reactive provisioning that are designed for different scenarios. Proactive provisioning relies on predicting workload intensity and uses workload prediction as a basis to allocate or de-allocate server resources. The effectiveness of proactive provisioning depends on several key factors, such as accuracy of workload prediction and cloud server capacity variations [Schad et al. 2010]. Therefore, in addition to periodically invoking proactive provisioning that makes adjustment plans using holistic information, GeoScale also performs reactive provisioning when SLAs are violated. Local reactive provisioning behaves similarly to Amazon EC2 "auto-scaling" while global reactive provisioning makes trade-offs between provisioning time [Mao and Humphrey 2012] and network latency reduction. Overall, GeoScale is configured to perform proactive provisioning periodically and only uses reactive provisioning as a complementary method to offset provision inefficiencies due to prediction errors.

Provisioning agility vs. DNS overhead. Unlike traditional provisioning that uses a static set of cloud location, GeoScale might introduce performance overhead from DNS lookups. Here, we discuss when such overhead can happen and the impact on client response time. In particular, setting appropriate time-to-live (TTL) values for DNS `A` records can impact how fast end users can benefit from servers in newly provisioned data centers, and how quickly GeoScale can scale down server resources. For example, when we have type `A` resource records with $TTL = 300$ seconds, if GeoScale provisions servers in a new data center, some end clients won't have access to new IP addresses for up to 5 minutes. This is because clients might still have cached DNS lookup responses from before the new type `A` record containing the updated IP addresses is configured. Similarly, GeoScale needs to wait for at least 5 minutes before deactivating a data center in order to avoid end user request errors. Therefore, to obtain the maximum provisioning agility, we need to set the DNS records TTL to a value as low as possible.

However, a shorter TTL value might increase DNS lookup time, e.g., taking 1 second instead of 100 ms [Ager et al. 2010]. This is because a shorter TTL value causes downstream DNS servers and caches to expire records faster. When cached DNS requests expire, lookup requests have to travel all the way to an authoritative name server. Note that, this type of DNS lookups will only happen once every TTL seconds, and once DNS records are cached on an intermediary transit path, subsequent lookups within TTL seconds can still benefit from caching. In addition, each client will only need to perform at most $\lceil \frac{TTL}{T_{session}} \rceil$ DNS lookups, where $T_{session}$ is the duration of each HTTP session. Because sessions can have many requests, overhead associated with DNS therefore can be amortized across all requests. Once a domain name is resolved to a public IP address of newly provisioned servers, client requests can then be routed and serviced at the closest cloud location, avoiding unnecessary network latency. Overall, the performance overhead caused by slower DNS lookups is minimal and can be offset by hundreds of milliseconds network reductions for *every* client request.

Considering the above, GeoScale employs an adaptive TTL policy for `A` records. Specifically, GeoScale uses an initial TTL value of 3600 seconds (maximum) and decreases it by 300 seconds whenever GeoScale makes provisioning plans that involve changing the set of active data centers, i.e., data centers that host application servers. Intuitively, the more frequent application cloud configuration changes are, the lower the TTL. The minimal TTL is restricted to be 300 seconds. In contrast, TTL is increased by 300 seconds when the provisioned data centers remain the same. By using an adaptive TTL policy, GeoScale automatically makes tradeoffs between provisioning agility and lookup overheads.

Other considerations in geographic provisioning. GeoScale uses a centralized approach that collects and aggregates workload information from distributed locations. Because GeoScale has a holistic view of workload distribution, it is well suited to make trade-offs between provisioning cost and performance gain, as demonstrated by our experiment and simulation results in Section 5.4. That is, if a data center location is only going to serve sparse client workloads, GeoScale can decide to inactivate this location and redirect future workload to a closer-by active cloud. Similarly, if client

workloads emerge near inactive cloud locations, GeoScale can provision an appropriate amount of servers in these new locations. However, approaches such as single-site elasticity or multi-site elasticity, only scale up and down servers in existing cloud locations. Therefore, such static provisioning approaches, failing to take advantage of low network latency, are sub-optimal for applications with growing and dynamic workload from different geographic locations. In contrast, GeoScale is most useful for such applications. But for applications with more centralized workloads or evenly distributed workloads, approaches such as single-site elasticity or multi-site elasticity that provision server resources in a static subset of all available cloud locations, are equivalently efficient.

4. GEOSCALE IMPLEMENTATION

We have implemented a prototype of GeoScale on top of Amazon EC2 cloud. GeoScale is written in Python and Java, and is implemented as a middleware that uses EC2 API to provide *geo-elasticity* to applications across Amazon's global network of cloud data centers. When deploying applications using GeoScale, application owners need to provide GeoScale access to application VM images. These images contain both application code and data, and are preconfigured with necessary components such as Apache web server, or MySQL database. In addition, application owners need to provide either previous workload information together with server statistics (see Section 3.2) or initial server configurations, including number of servers, server type and cloud locations. GeoScale then deploys application by renting servers from EC2, and proactively adjusts resources periodically. GeoScale consists of a central provisioning manager that runs in one of Amazon's EC2 data centers and a distributed daemon that is deployed to all managed application servers. This central manager is chosen to run in a data center location that offers cheaper server resources and has the best network connectivity to the active data centers for fast aggregation of monitoring data. Because the set of active data centers vary over time due to workload fluctuation, the location of the centralized manager is updated to reflect such dynamics. However, for all our experiments, we host the centralized manager in Amazon's Virginia data center for simplicity. Next, we describe the components of GeoScale's central manager in detail, as depicted in Figure 2.

- **Workload Monitoring, Profiling and Forecasting.** GeoScale collects workload statistics by running a small daemon on each of the application's servers; the component collects system statistics using standard Linux utilities, e.g., `sar`, and gathers application logs (such as Apache Tomcat logs). We modified the Apache Tomcat server to log request service times, in addition to standard information such as response times and client information. Request service time information is essential in our queueing-based approach for deriving server capacity, discussed in Section 3.2. If the application uses a load balancing software component, request logs can be gathered from this component, rather than from application logs at each server replica. All monitored data is sent to a central GeoScale server that runs on a Virginia-based medium-sized server for archiving and further analysis. The central manager stores historical workload information in SQLite database; it uses the GeoIP-location service to map IP addresses to their geographic locations. Workload forecasting is done using ARIMA time-series model from Python's `StatsModels` and `Pandas` Library, to predict the future workload for the application.
- **Provisioning algorithms.** Our model-driven provisioning algorithms is the core component of the central manager. GeoScale supports both proactive and reactive provisioning and by default, GeoScale periodically performs proactive provisioning that determines a new set of cloud locations and the number of servers needed for each location. The proactive provisioning period has a default configuration of 24 hours. This time window can be tuned to reduce unnecessary cloud cost due to over provisioning—proactive engine provisions for the peak workload predicted for the next time window, and reactive engine is only triggered by workload spikes. Currently, we use a heuristic method to adjust it based on past provisioning accuracy and provisioning time. The provisioning accuracy a_p is roughly indicated by the number of times reactive provisioning is successfully triggered. The provisioning period window is increased by a_p hours and is decreased by 1 hour when $a_p = 0$. Intuitively, the proactive engine is invoked less frequently when the workload

Table I: Amazon's Distributed EC2 Cloud

Name	Location	Latitude	Longitude	Time Zone	built(yrs)
us-east-1	Virginia	38.13	-78.45	UTC-05:00	2006
eu-west-1	Ireland	53.00	-8.00	UTC+00:00	2007
us-west-1	California	41.48	-120.53	UTC-08:00	2009
ap-southeast-1	Singapore	1.37	103.80	UTC+08:00	2010
us-west-2	Oregon	46.15	-123.88	UTC-08:00	2011
ap-northeast-1	Japan	35.41	139.42	UTC+09:00	2011
sa-east-1	Brazil	-23.34	-46.38	UTC-03:00	2011
ap-southeast-2	Australia	-33.86	51.20	UTC+10:00	2012

pattern is less predictable. The minimal provisioning window is set to be the average provisioning time taken.

GeoScale runs its proactive engine in the same central server that stores historical workload and server resource utilization information. The proactive engine takes these data and predicted workload for the next period as input, and produces a provisioning plan using our queueing model-based algorithms. Within each provisioning window, our reactive engine continuously monitors the actual incoming workload and compares it with the predicted peak, as an indication of SLA violations. If the actual workload is consistently higher for more than the oscillation window (configured as 5 minutes), the reactive engine will first decide whether to use the global reactive or the local reactive algorithms based on the availability of up-to-date workload data from current set of active data centers, and the locations of where VM images are archived. When using the global reactive algorithm, the reactive engine will provision servers in data centers—possibly in one of the archived data centers—that are closest to the actual workload spike; while when using the local reactive algorithm, the reactive engine will provision servers in data centers where workload spikes are observed and reported from.

- **Provisioning engine.** This component uses `boto`, a python interface to Amazon web services API, to manage the application's VMs. This provisioning engine actuates by starting, stopping or terminating cloud servers in appropriate cloud locations, based on provisioning plans derived by provisioning algorithms. To relinquish running servers during proactive provisioning, GeoScale uses the `terminate_instances` command. Doing so stops all billings associated with the servers, including computation and EBS storage. However, in order to retain the provisioning agility, GeoScale only de-allocates computation resources but holds on to EBS storage by using the `stop_instances` command. Although stopped instances incur an EBS storage fee proportional to the entire stopped duration, they provide GeoScale the ability to provision more quickly when compared to provisioning a new instance from scratch [stopped vs terminated instances 2017; Amazon EBS Pricing 2017]. In addition, it works closely with the copying engine to manage the application's VM disk images and data, and copy them to new data centers as needed.
- **Copying engine.** This component is responsible for managing an application's VM images and data volumes. GeoScale supports both on-demand copying and lazy pre-copying. It employs EC2 snapshot techniques, e.g., `ec2-create-snapshot` and `ec2-copy-snapshot`, to create and transfer application-agnostic snapshots within and across data centers; for incremental snapshot data, we use *rsync* to transfer incremental snapshot data efficiently, e.g., in compressed form.

5. EXPERIMENTAL EVALUATION

We have conducted a detailed experimental evaluation of GeoScale on Amazon's EC2 cloud. Our evaluation focuses on (i) benefits of exploiting workload dynamics, (ii) the efficacy of our capacity model, (iii) a comparison of GeoScale with current approaches such as using a CDN or the manual choice of multiple cloud locations, (iv) efficacy of proactive and reactive provisioning, and (v) benefits of GeoScale's pre-copying optimizations. Next we describe our experimental setup and then our results.

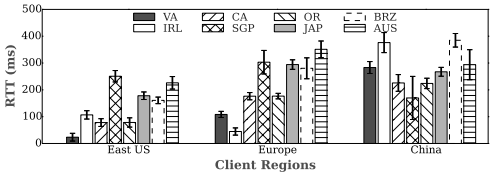


Fig. 4: Clients from different regions observe minimal mean round-trip time at different cloud locations.

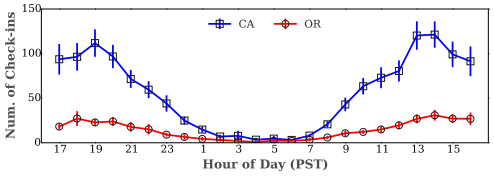


Fig. 5: Gowalla workload's *temporal* time of day effects.

Experimental Setup. Our experiments use Amazon EC2 as our distributed cloud. As shown in Table I, Amazon offers a choice of multiple cloud locations across several continents for hosting applications; our experiments exploit this flexibility and allow GeoScale to provision servers for applications at any location. To inject a geographically diverse workload to the cloud-based application, we employ 101 PlanetLab nodes that are distributed across South and North America, Europe, Asia and Australia. PlanetLab nodes are chosen primarily based on their availability at the time of experiments and their proximity to data centers in Table I. We run client workload generators on these nodes and use a home-grown custom DNS service that resolves the server IP addresses, requested by these client machines, to the nearest cloud site that hosts an application replica.[2] In addition to using geographic distance between clients and cloud servers, calculated using IP-geolocation and the Haversine formula, we also collect ground-truth network distance of each PlanetLab node from the various EC2 data centers using empirical round-trip times (RTT) measurements. We use `fabric` [Fabric Python Module 2013] to automate the running of our experiments.

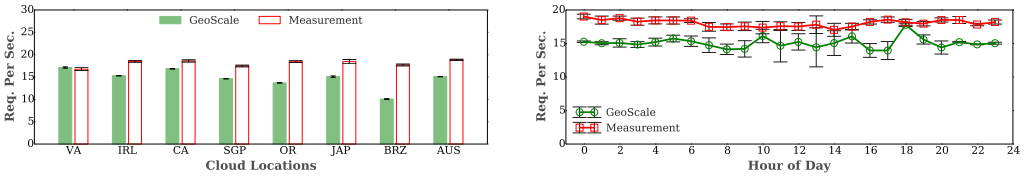
Application Workloads. We use a Java implementation of TPC-W [The ObjectWeb TPC-W implementation 2005] as a representative multi-tier web application for our experiments. The TPC-W benchmark emulates an online bookstore and employs a two-tier architecture, a web server tier based on Apache Tomcat and a database tier based on MySQL. The system provides a client workload generator that we run on PlanetLab nodes to generate and inject a mix of browsing and shopping requests. We assume that the TPC-W application uses an eventual consistency model across replicas where updates to the product catalog of the TPC-W web-store are made in batches and propagated using eventual consistency. In addition to using the TPC-W benchmark, we also employ user traces (i.e., check-in request logs) from Gowalla, a location-based social networking application [Cho et al. 2011]. The dataset comprises 6.5 million requests over a 20-month period starting February 2009 and each request includes time-stamp and the GPS coordinates (latitude and longitude) of user check-ins.

5.1. Exploiting Workload Dynamics

In Figure 1, we showed that application workload exhibits spatial and temporal variations, and here we demonstrate the benefits of considering both spatial and temporal fluctuations when provisioning capacity.

We first group the PlanetLab clients into three regions and calculate the mean RTT seen by clients in each regions to the various Amazon EC2 data centers. Figure 4 shows that network latency increases with increasing geographic distance across all three regions, which validates our assumption that geographic distance can be used as a rough proxy for network distance. Moreover, different client regions see the smallest network latency at different data center locations, e.g. eastern US has the best latency when serviced from Virginia (VA) data center while Europe enjoys the best latency from the Ireland (IRL) data center. In fact, the latency benefits for placing replicas closest to clients could be as high as 70% compared to the second best choice. This demonstrates that it is more

[2]Any latency-based routing DNS service, such as Amazon Route 53, could be plugged-in for domain name resolution.



(a) Maximum capacity of small server instances from different EC2 locations.

(b) Performance interference causes the server capacity to vary at different times of the day.

Fig. 6: **Comparisons of GeoScale's capacity Modeling with empirical measurement.** We observe cloud location-based and interference-based capacity variations even for the same type of server instance.

beneficial to provision replicas at cloud locations that are close to the users, rather than centralizing all replicas at a single cloud location, which yields worse performance to distant clients.

Next we simulate the Gowalla workload by iteratively assigning its clients to the closest available cloud locations. Figure 5 depicts the temporal variations seen by the Gowalla workload at two simulated servers at Amazon's California and Oregon cloud locations. The servers observe time of day effects where the workload peaks during the day and ebbs at night—in line with the expected temporal variations in the load. These visible time of day effects indicate that we can dynamically scale up and down the server capacity during the day and night. Together these results indicate that exploiting both spatial and temporal workload effects can yield significant benefits.

5.2. Capacity Model Analysis

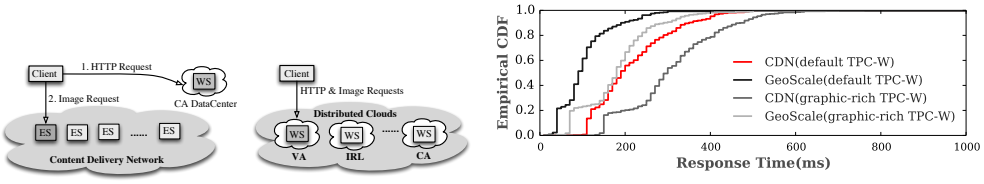
To evaluate our queueing-based model, we empirically measure the capacity of Amazon's small server instances running in different cloud locations at different times of day and compare our measurement to the model's predictions. For each run, we host the TPC-W multi-tier application on a small server instance in one of the cloud locations during a particular hour and ran the clients on a nearby PlanetLab node.[3] We warm-up the application for five minutes and then steadily increase the workload until GeoScale detects SLA violations (i.e. 95^{th} percentile response time is greater than 1 second). We record the corresponding request rate and use it to represent the server capacity, i.e., the number of requests a server can service without violating SLA. At the same time, we collect the application service time and request inter-arrival rate statistics and use the queueing model described in Section 3.2 to estimate the server capacity. We ran this experiment with different cloud locations and hour of day combinations and repeated the runs for 5 times for each combination.

Figure 6a compares the server capacity predicted by GeoScale with the empirically measured ones for each cloud location. The measured server capacity varies across cloud locations, with up to a 12% difference in capacity across locations for the same type of server. The capacity variances across different cloud locations emphasize the need for location-specific performance models for the same type of server.

Figure 6b depicts the variations in the maximum server capacity of a small server instance at different times of the day. As shown, the measured server capacity varies over time and we see up to 11.6% difference in the empirically observed peak capacity over time—with the highest observed capacity at midnight and minimum capacity around noon. Again GeoScale's models make conservative predictions with accuracies ranging from 75.2% to 97.9%.

These observed capacity variations can be attributed to several aspects [Tickoo et al. 2010], such as heterogeneous server hardware deployed at different locations (see Table I) or interference from other co-located VMs on each physical server. Regardless of the underlying causes for variations,

[3]The mean round-trip time between client and server is 53.63 ms.



(a) Setup for CDN. (b) Setup for GeoScale. (c) CDF of client-perceived response time.

Fig. 7: **A comparison of GeoScale's geo-elasticity to a CDN-based approach.** GeoScale outperforms the CDN-based approach for both default and graphic-rich browsing workload.

Table II: Different percentiles of the client-perceived latency for the CDN and GeoScale approaches.

Experiment Type	25% (ms)	50% (ms)	75%(ms)	95% (ms)
CDN (default TPC-W)	150.0	198.0	275.3	410.3
GeoScale (default TPC-W)	70.0	98.0	130.0	244.4
CDN (graphic-rich TPC-W)	254.0	304.0	389.0	521.0
GeoScale (graphic-rich TPC-W)	142.8	186.5	228.0	343.2

our empirical measurements provide a guideline for parameterizing our models. That is, in the absence of strict performance isolation between unrelated VMs on a cloud server, the parameters for our queueing-based capacity model must be chosen conservatively—for instance, by carefully choosing parameters when there is high interference. Doing so ensures that application SLAs will not be violated even in the worst case scenario of cross-VM interference.

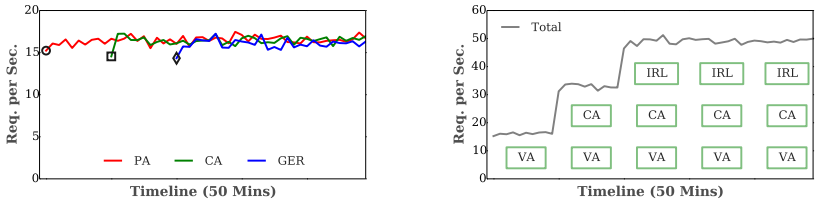
5.3. Comparison with a CDN

This section and the next compare our geo-elasticity approach with two current approaches: use of a CDN to service a distributed user base and use of manually chosen cloud locations to do so.

In the CDN case[4], depicted in Figure 7a, we deploy the TPC-W application at Amazon's California data center and host all static content such as images using in edge servers that are deployed in all cloud regions. The client, which runs on PlanetLab node in Pennsylvania, makes initial requests to the TPC-W California servers to obtain the dynamically generated HTML page and then loads all embedded images within the HTML page from nearest CDN server (see Figure 7a). In case of GeoScale, the TPC-W application is itself replicated across cloud locations as shown in Figure 7b and the client loads both dynamically-generated page and all of its embedded static content from the closest server (which happens to be the Virginia server for the PlanetLab client).

Figure 7c depicts the CDF of the response times seen by the client to load a page and all of its content for the TPC-W browsing workload mix. Table II depicts various percentiles of the response times for the two approaches. As shown, GeoScale outperforms the CDN approach, with a 40.43% reduction in the 95^{th} percentile of the response time from 410.3 ms to 244.4 ms. Since TPC-W performs more request processing to create dynamic content and has relatively less static content, GeoScale is able to outperform the CDN approach. We then increase the size of static images served by the TPC-W application and repeat this experiment. As shown in Figure 7c and Table II, graphic-intensive requests increase the response times, but the gap between GeoScale and the CDN approach decreases for more graphic-rich version of TPC-W—the benefit in the 95^{th} percentile of response time drops from 40.43% to 34.13%. Thus, the more graphic-intensive the application and less dynamic content it serves, the smaller the difference between these two methods. We also note that the two approaches are not mutually exclusive, since GeoScale can replicate the application across

[4]We do not use any commercial CDNs due to the lack of control of edge server locations.



(a) Client workload. (b) Proactive Provisioning.

Fig. 8: Illustration of proactive provisioning set up and result.

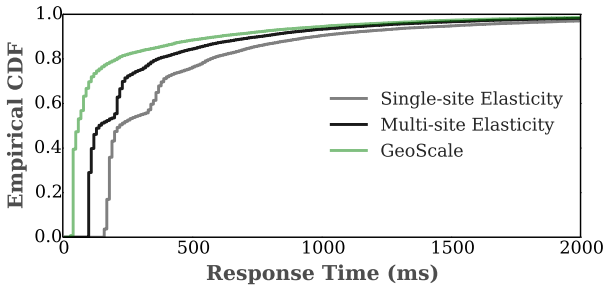


Fig. 9: ECDF Comparison of client-perceived response time to single-site and multi-site elasticity. By employing geo-elastic proactive provisioning, GeoScale yields the best response times, with 95% of the requests finishing in less than 1060ms.

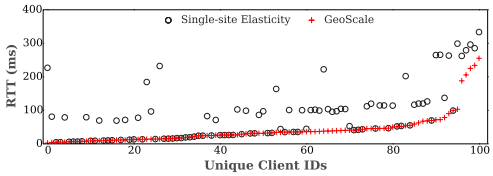


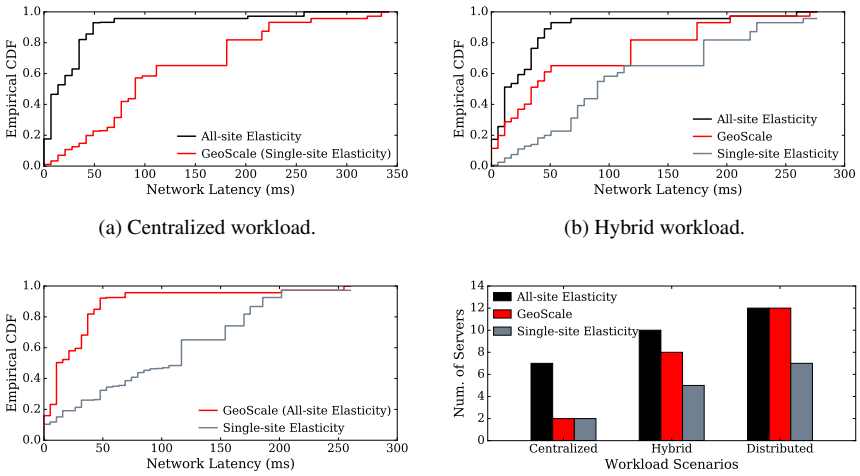
Fig. 10: GeoScale improves the latency seen by more than half the clients over Single-site Elasticity.

Fig. 11: GeoScale's greedy workload clustering has comparable performance to the more expensive ILP approach.

cloud locations and offload its static content to a CDN, allowing for the hybrid approach to take advantage of the larger number of locations supported by a global CDN.

5.4. Benefits of Geo-elastic Proactive Provisioning

Provisioning Techniques. To demonstrate the benefits of GeoScale's proactive provisioning approach, we compare it with two variants of local elasticity where the choice of cloud locations is made manually: (i) single-site elasticity (SSE) and (ii) multi-site elasticity (MSE). SSE is a centralized approach that hosts all replicas of the application at a single cloud location, while MSE houses the application replicas at a pre-determined static set of locations. In contrast, GeoScale has the flexibility to vary the locations that host the application as well as the number of servers at each location. We assume all three provisioning approaches employ the queuing model described in Section 3.2 to handle temporal workload fluctuations.



(a) Centralized workload.

(b) Hybrid workload.

(c) Geo-distributed workload.

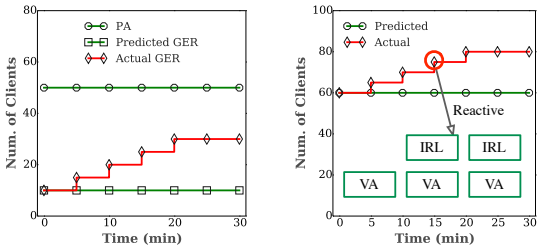
(d) Provisioning cost.

Fig. 12: ECDF comparison of network latency. GeoScale can adapt its behaviors under different workload scenarios. As client workload becomes increasingly geo-distributed, the gap of network latency between ASE and GeoScale shrinks.

Our experiment involves running TPC-W clients on PlanetLab nodes at three locations: Pennsylvania and California in the USA and Germany in Europe. The workload generated by these client sites is depicted in Figure 8a. Initially only the Pennsylvania clients send requests to the application; at t=10 minute, the California clients start sending requests, followed by requests from the Germany clients at t=20 minute. All three proactive approaches are able to scale up server capacity in response to the workload increase. The main difference is *where* the servers are provisioned. The SSE technique centralizes all replicas at Amazon's Virginia (US-East) cloud location and scales server capacity from one server to three servers at this site to handle the workload increase. The MSE approach is configured with replicas at Amazon's Virginia and California (US-East and US-West) data centers and it provisions one server in California and two servers in Virginia to handle the incoming workload. GeoScale's proactive elasticity technique allocates one server in Amazon's Virginia data center to handle the Pennsylvania clients, followed by another server in Amazon's California location to handle California traffic, and a third server in Amazon's Ireland data center to handle the traffic from Germany as shown in Figure 8b.

Figure 9 shows the CDF of the client response times across all clients for the three provisioning approaches. SSE has the highest response times since it uses a single cloud location to serve global traffic, causing distant clients to see worse response times. MSE approach uses a couple of fixed locations to host the application and is able to direct clients to the closer of the two fixed locations, yielding better response times than SSE. GeoScale yields the best response times since it is able to provision servers that are closest to the clients. The 95^{th} percentile response time provided by GeoScale is around 1060 ms, a 31.17% improvement over SSE and a 13.11% improvement when compared to MSE.

Aside from the above experiment, we also use the network latency data collected between PlanetLab nodes and servers from geographically distributed clouds to simulate the behaviors of three provisioning techniques. We create three workload scenarios that represent workload characteristics of growing geo-distributed applications at different stages. Specifically, each scenario differs in the level of spatial distribution, from fully centralized to evenly distributed among all regions. For example, in the first scenario (centralized workload), client workloads are mostly centralized around



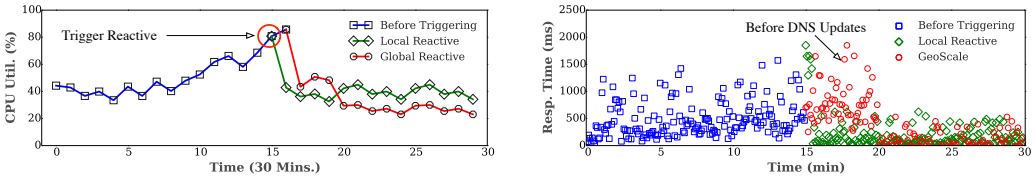
(a) Client workload. (b) Global reactive.

Fig. 13: Illustration of reactive provisioning set up and result.

one region with light volumes from all the other regions. We control the aggregate amount of client workloads to each popular cloud location to be slightly more than threshold τ, and for this simulation we fix this threshold to be the capacity of a single server. In Figure 12, we plot the network latency distribution and provisioning cost achieved by three different techniques. In the first scenario (Figure 12a), e.g., applications that have regional popularity, GeoScale (equivalent to SSE) yields network latencies with a 95^{th} percentile that is about twice as much as that of All-site Elasticity (ASE), where replicas are managed autonomously within all available data center locations. This is because GeoScale also takes provisioning costs into account, and avoids placing servers in cloud locations that will have less than the threshold τ traffic. In this particular scenario, GeoScale yields a 71% saving on server costs when compared to ASE (Figure 12d). In the last workload scenario (Figure 12c), e.g. applications that are popular among all geographic regions, GeoScale (equivalent to ASE) decreases 95^{th} percentile network latency from 204 ms to 72 ms, a 65% improvement. However GeoScale will provision five more servers than SSE because GeoScale chooses to provision in six different cloud locations. Note, these additional five servers are only lightly loaded and have the ability to scale to larger workload volume at each cloud location. In summary, GeoScale can adaptively provision servers in cloud locations to make trade-offs between performance, e.g. network latency or response time, and provisioning cost, under different workload scenarios. Such trade-offs are achieved by varying the value of threshold τ.

Cloud Location Flexibility. To further demonstrate the benefits of flexibly using all accessible cloud locations to host application replicas for geographically distributed users, we compare the average network latencies seen by clients between SSE and GeoScale. In Figure 10, we plot the average network latencies of all 101 PlanetLab clients to the closest Amazon data center in ascending order as chosen by GeoScale. We also plot the network latency of each client to the Amazon's Virginia data center as in SSE approach. As shown, when provisioning capacity using SSE in Amazon's Virginia data center, clients experience up to 333.47 ms network latency. GeoScale is able to utilize all the available data centers and to choose the nearest for each client, yielding up to 224.43 ms reduction in network latency for clients.

Workload Clustering. We compare the mean network latency achieved by GeoScale's greedy workload clustering algorithm to that of the ILP algorithm. Both algorithms use a same list of predefined geographic bins as well as the network latency between each PlanetLab client and Amazon data center, to produce a mapping between each bin and the best cloud location. We repeat each experiment ten times for different numbers of available cloud locations. Figure 11 depicts the mean network latency among all clients with increasing number of cloud locations. As shown in the figure, the greedy approach exhibits performance that is close to the more expensive ILP approach, indicating that the greedy workload clustering algorithm will yield good results in practice.



(a) Average CPU utilization of replicas.

(b) Response times of clients.

Fig. 14: **Comparison of local and global reactive provisioning.** Global reactive provisioning leads to lower CPU utilization and better client response time compared to local reactive one.

5.5. Geo-elastic Reactive Provisioning Performance

Next, we evaluate GeoScale's reactive provisioning and compare CPU utilizations and average response times differences between global and local reactive provisioning. At the beginning of the experiment, GeoScale only provisions a single server in Virginia data center based on the workload prediction of client traffic from Pennsylvania and Germany, as shown in Figure 13a. Since the predicted traffic from Germany is too low to justify deploying a server in Europe, no server is provisioned in the Ireland data center. On the other hand, the actual workload from Germany shows a steady, unexpected increase and eventually saturates the only server in Virginia. GeoScale detects the SLA violation at t=15 minute and therefore triggers the reactive provisioning. Local reactive approach reacts to SLA violations by provisioning an additional server in the same data center while global reactive approach obtains workload data from the workload monitoring engine, and upon seeing the workload increase from Germany, reactively provisions a second server in the Ireland data center that is closest to Germany, as shown in Figure 13b.

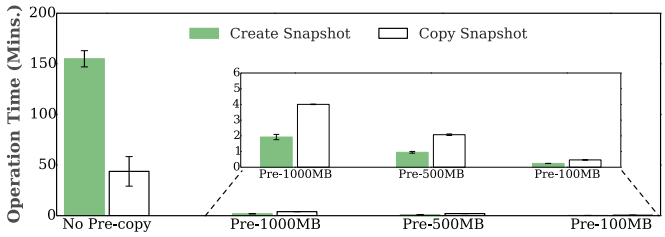
As shown in Figure 14a, the workload increase causes the server utilization to rise and SLA violations to increase steadily, until it reaches a point (orange circle) where reactive provisioning is invoked. It takes longer for server utilization to drop when using global reactive provisioning since Germany clients progressively switch to the new server due to DNS propagation delay of around 5 minutes. Similarly, in Figure 14b, the response time SLA violation also subsides faster when using local reactive provisioning at t=16 minutes. But global reactive provisioning achieves lower mean response time since it is able to provision the server closer to where the workload increase is seen.

5.6. Pre-copying Optimization Analysis

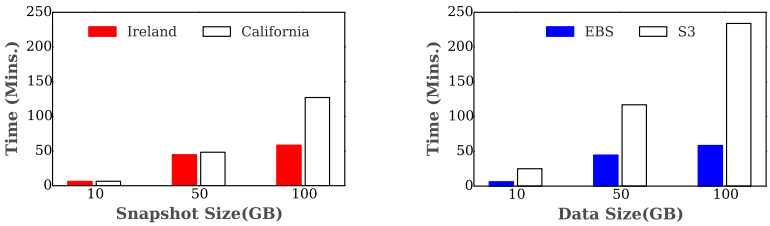
GeoScale employs three optimizations for fast geo-elastic provisioning (i) pre-copying large VM image (ii) dynamically choosing data center pairs for pre-copying based on available bandwidth (iii) using Amazon EC2 storage volumes for transferring incremental data. In our final experiment, we justify our choice of optimizations based on the huge reduction in provisioning time. We use TPC-W applications configured with different database sizes(i.e. 10GB, 50GB and 100GB) We measure the operation time of GeoScale equipped with three optimizations and compare the results with those of the alternatives.

Pre-copying Large VM Image. In order for GeoScale to provision an application replica at a new cloud location, it must have access to VM disk image as well as any additional data volumes, such as database, at that location. For applications with large amounts of data, this can result in a long delay in finishing geo-elastic provisioning in the absence of the data. Our first optimization aims at pre-copying VM with large amount of data to cloud locations that are likely to host future replicas. Precopying a VM image to a new location involves two main steps: (a) create a snapshot, e.g. using Amazon's `ec2-create-snapshot`, and (b) copy snapshot from one cloud location to the new location, e.g., using Amazon's `ec2-copy-snapshot` or `rsync` utility.

Figure 15a plots the preparation time to provision an application replica at a new location, i.e., Ireland data center, with and without GeoScale's pre-copying optimization. Without pre-copying



(a) By pre-copying a VM image that includes a 100GB database, GeoScale reduces the provisioning delay from 198.8 minutes to 355.6 seconds (for 1 GB delta), 181.4 seconds (for 0.5 GB delta), or 42.5 seconds (for 0.1 GB delta).



(b) Different location pairs.

(c) S3 v.s. EBS.

Fig. 15: Benefits of pre-copy optimizations.

data into Ireland, the process takes around 200 minutes to finish as shown in the leftmost bar group. By pre-copying the VM data periodically, only the incremental changes since the most recent pre-copying operation need to be transferred, therefore could significantly reduce the preparing time for provisioning. The zoomed-in figure in Figure 15a depicts three scenarios where a delta of 1GB, 0.5GB and 100MB need to be transferred prior to starting up the VM; the observed latency ranges from 355.6 seconds to 42.5 seconds, allowing the replica to be provisioned in minutes, rather than hours.

Dynamic Source Site Selection. While Amazon's data centers have well-provisioned network links between them, we observe significant differences in the amount of bandwidth available (and hence the latency to pre-copy a certain amount of data) between different pairs of Amazon data centers. Figure 15b compares the time to pre-copy application-agnostic snapshots of various sizes from Amazon's Virginia data center to its Ireland or California locations using `ec2-copy-snapshot`. It only takes 3510 seconds to copy a 100GB disk image to Ireland, while it takes 7629 seconds to copy the same disk image to the California location. This observation justifies the need to dynamically choosing pre-copy *source site* based on available bandwidth to further optimize the provisioning time.

Using EBS Volumes. Amazon provides two different ways to store application data, i.e. EBS storage volumes and S3 storage service. We explore the operation time differences in pre-copying data using *ec2-copy-snapshot* operation through EBS and S3 *cp* operation from Amazon's Virginia data center to its Ireland location. Figure 15c shows the overhead of these two different methods to prepare VM data for provisioning. The result shows that the use of EBS to copy incremental data is, on average, 70.53% faster than using S3 for all three data sizes. This is not surprising, since EBS volumes are more expensive, and also offer better performance. Finally, this supports our optimization of using EBS volume to precopy VM data.

6. RELATED WORK

Virtualization and Application Elasticity. Early work in elasticity focused on using virtualization platforms to support elasticity by dynamically adjusting the resources allocated to virtual machines. VM-based elasticity mechanisms include "scale up" [Knauth and Fetzer 2011; Lakew et al. 2014] techniques using VM live migration [Clark et al. 2005] or "scale out" [Lagar-Cavilla et al. 2009] techniques by spawning replicas locally or in the public cloud [Marshall et al. 2010]. Our work focuses on a scale-out scenario and enhances prior work to handle cross-data center elasticity, where VMs can be spawned in multiple data centers. In general, VM-based elasticity is a desirable property that helps to efficiently manage resources of applications with dynamic workload.

However, application-specific optimizations are often needed, e.g., to avoid high operation costs [Mickulicz et al. 2013]. Application-level elasticity has been studied extensively in the context of databases; proposed techniques include the use of live database migration techniques [Das et al. 2011; Elmore et al. 2011] for relational database and stop and migrate techniques [Cooper et al. 2008] for key-value stores. In addition, [Tsoumakos et al. 2013] targets NoSQL database and proposes a framework for providing elasticity of NoSQL clusters using Markov Decision Process. Performance-aware replication of data in geo-distributed cloud stores has also been studied in this context [P N et al. 2014]. Other related efforts target specific scenarios for elasticity that include transparent load balancing [Lim et al. 2010; Rajagopalan et al. 2013] by moving per-flow state in the middle-box or rebalancing data among cloud storage systems.

Leveraging Geo-distributed Clouds. Geographically distributed clouds have become popular and provide resources to run services in multiple locations. Prior work has looked at how to select data center locations to optimize cost and network latency. Specifically, [Zhang et al. 2013] proposes the use of control and game theoretic models to optimize operation cost given both cloud workload and price fluctuations. Greening Load Balancing [Liu et al. 2011] focuses on determining workload for geographically distributed data centers by considering energy costs. Network aware resource allocation [Alicherry and Lakshman 2012] that places user requested VMs into data centers so that the distance between selected data centers are minimized. GeoScale also takes advantage of geo-distributed cloud locations and selects cloud locations by iteratively assigning and grouping client workload to their closest locations, avoiding unnecessary resource provisioning.

Model-driven Approaches. Past work on model-driven techniques have focused on clustering techniques such as k-means [Gmach et al. 2007; Singh et al. 2010] or independent component analysis [Sharma et al. 2008] to characterize dynamic workload or workload spikes [Bodik et al. 2010], which is then used for providing elasticity. In addition, prior work [Nguyen et al. 2013; Morais et al. 2013; Tolosana-Calasanz et al. 2014] also focuses on predicting service performance and leverages such predictions to provision resources. For example, in AGILE [Nguyen et al. 2013], the authors use wavelets that provide medium-term performance predictions; and a black-box performance model that captures the relationship of SLO violations and resource utilizations by fitting profiling data.

Other model-driven approaches [Gandhi et al. 2014; Malkowski et al. 2011; Padala et al. 2009] have relied on queueing theory with specific optimizations, such as using multi-model [Malkowski et al. 2011] or Kalman filtering [Gandhi et al. 2014] to provision virtualized resources in the same data center. Regardless of specific performance models, it is also important to account for potential performance variations due to shared cloud environments [Tickoo et al. 2010; O'Loughlin and Gillam 2014]. GeoScale leverages this past work on model-driven approaches and employs per-site and per-server queueing theoretic models to capture differences in server capacities across cloud locations.

Reducing Latency. Intelligent service placement has been studied in the networking context to reduce user latency. Most of those approaches place services closer to the end-users to reduce network latency [Satyanarayanan et al. 2009; P N et al. 2014; Content Delivery Network 2013]. Our focus is on dynamic capacity provisioning to handle spatial workload dynamics, rather than an intelligent one-time placement of services. Other complementary approaches to reduce web latency include

protocol level approaches [Flach et al. 2013], and application-layer optimizations [Stream Control Transmission Protocol 2013].

7. CONCLUDING REMARKS

In this paper, we presented GeoScale, a system that provides geo-elasticity to replicable multi-tier web application in a geographically distributed cloud platform. Geo-elasticity provides an automatic resource management for applications that experience not only temporal but also spatial workload dynamics. GeoScale achieves geo-elasticity by tracking an application's spatial and temporal workload dynamics, employing a combination of queueing model-driven proactive provisioning and agile reactive provisioning, along with pre-copying optimizations to provision server capacity at the closest possible cloud locations to the distributed user base. Our experimental evaluation of the GeoScale prototype on Amazon EC2 yielded up to a 40% reduction in the 95^{th} percentile response times and up to a 58% reduction in SLA violations for representative web applications, when compared to traditional local elasticity mechanisms.

GeoScale's hybrid provisioning approach allows more agile provisioning experience for cloud-based applications. This is because by proactively provisioning resources, GeoScale can mask potentially long provisioning time; and by reacting to SLA violations in real time, GeoScale can make up for under-provisioning caused by prediction errors or flash crowds. As cloud providers are building more data centers across the globe, it provides GeoScale more opportunities to further improve SLA compliance. It also brings additional scalability challenges when designing systems like GeoScale. For example, our current ILP-based geographically provisioning might take hours to reach an optimized decision with increased number of data center locations. As part of future work, we plan to explore alternative approaches that present tuneable tradeoffs between provision quality and decision , as well as to extend GeoScale to handle heterogeneous cloud servers and integrate horizontal and vertical scaling.

A. SLA-CONSTRAINED SERVER CAPACITY

We model a single cloud server as a G/G/1 queue and use T, W and X to denote the response time, waiting time and service time distributions respectively. The application-level SLA y is defined as a strict upper bound of a high percentile response time. For the ease of illustration, we choose 95^{th} percentile response time. That is, if the SLA $y = 2$ seconds, the 95^{th} percentile of response time T, denoted as $\alpha_T(95)$, has to satisfy $\alpha_T(95) < 2$. Our goal is to derive the amount of requests λ a server could process without violating the SLA. More specifically, we are interested in finding the upper bound of λ when the server is experiencing heavy traffic. These Heavy traffic scenario can be represented as $\rho \to 1$, where ρ denotes server utilization.

First, based on Little's Law, we can represent $\lambda = \frac{\rho}{\bar{x}}$. We then apply Kingman's theorem in order to represent $\alpha_T(95)$ for heavy-traffic servers modeled as G/G/1 queue. Kingman's theorem states that the waiting time W is an exponential distribution with mean $E[W]$ as shown in Equation 8 and 9.

$$E[W] = \frac{\sigma_a^2 + \sigma_b^2}{2(\frac{1}{\lambda} - \bar{x})} \tag{8}$$

$$\begin{aligned} F_W(w) &= P(W \leq w) \\ &= 1 - \exp^{-\frac{1}{E[W]}w} \end{aligned} \tag{9}$$

where σ_a^2 and σ_b^2 represent the variances in requests' inter-arrival times and service times; and $\bar{x}$ denotes the mean service time. By definition, we have:

$$P(T \leq \alpha_T(95)) = 0.95 \tag{10}$$

$$T = W + X \tag{11}$$

Note that all x from service time distribution X has to satisfy Equation 11, we then have $T \approx W + \bar{x}$. Combining Equation 9 to 11, we have an equation of $\alpha_T(95)$ in terms of $E[W]$ and $\bar{x}$ in Equation 12.

$$1 - \exp^{-\frac{1}{E[W]}(\alpha_T(95)-\bar{x})} = 0.95 \tag{12}$$

Applying ln operation to both side, we can express $\alpha_T(95)$ with all known parameters as shown in Equation 13.

$$\alpha_T(95) = \frac{3\lambda(\sigma_a^2 + \sigma_b^2) + 2(1-\rho)\bar{x}}{2(1-\rho)} \tag{13}$$

To satisfy SLA y, we just need to have $\alpha_T(95) < y$. By substituting Equation 13 to the SLA constrain, we obtain the upper bound of request rate λ as:

$$\lambda < \left[\bar{x} + \frac{3({\sigma_a}^2 + {\sigma_b}^2)}{2\,(y-\bar{x})}\right]^{-1} \tag{14}$$

REFERENCES

Bernhard Ager, Wolfgang Mhlbauer, Georgios Smaragdakis, and Steve Uhlig. 2010. Comparing DNS Resolvers in the Wild. In *IMC '10: Proceedings of the 2010 Internet measurement conference*. Melbourne, Australia.

M. Alicherry and T. V. Lakshman. 2012. Network aware resource allocation in distributed clouds. In *INFOCOM, 2012 Proceedings IEEE*. 963–971. DOI:http://dx.doi.org/10.1109/INFCOM.2012.6195847

Amazon Auto Scaling Service 2013. AWS Auto Scaling. (2013). https://aws.amazon.com/autoscaling/

Amazon EBS Pricing 2017. Amazon EBS Pricing. (2017). https://aws.amazon.com/ebs/pricing/

P Bodik, A Fox, M J Franklin, and M I Jordan. 2010. Characterizing, modeling, and generating workload spikes for stateful services. In *Proceedings of the 1st ACM symposium on Cloud computing (SoCC)*. ACM, New York, NY, USA, 241–252.

Eunjoon Cho, Seth A Myers, and Jure Leskovec. 2011. Friendship and mobility. In *the 17th ACM SIGKDD international conference*. ACM Press, New York, New York, USA, 1082–1090.

C Clark, K Fraser, S Hand, J G Hansen, and E Jul. 2005. Live migration of virtual machines. In *Proceedings of the 2nd conference on Symposium on Networked Systems Design and Implementation (NSDI)*, Vol. 2. USENIX Association, Berkeley, CA, USA, 273–286.

Content Delivery Network 2013. Content Delivery Network. http://www.akamai.com/html/resources/content-distribution-network.html. (2013).

Brian F Cooper, Raghu Ramakrishnan, Utkarsh Srivastava, Adam Silberstein, Philip Bohannon, Hans-Arno Jacobsen, Nick Puz, Daniel Weaver, and Ramana Yerneni. 2008. PNUTS. *Proceedings of the VLDB Endowment* 1, 2 (Aug. 2008), 1277–1288.

Sudipto Das, Shoji Nishimura, Divyakant Agrawal, and Amr El Abbadi. 2011. Albatross: lightweight elasticity in shared storage databases for the cloud using live data migration. *Proceedings of the VLDB Endowment* 4, 8 (May 2011), 494–505.

Aaron J. Elmore, Sudipto Das, Divyakant Agrawal, and Amr El Abbadi. 2011. Zephyr: Live Migration in Shared Nothing Databases for Elastic Cloud Platforms. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*. ACM, New York, NY, USA, 301–312.

Fabric Python Module 2013. Fabric documentation. (2013). http://www.fabfile.org/

Tobias Flach, Nandita Dukkipati, Andreas Terzis, Barath Raghavan, Neal Cardwell, Yuchung Cheng, Ankur Jain, Shuai Hao, Ethan Katz-Bassett, and Ramesh Govindan. 2013. Reducing web latency: the virtue of gentle aggression. In *Proceedings of the ACM SIGCOMM 2013 conference on SIGCOMM*. ACM, New York, NY, USA, 159–170.

A Gandhi, P Dube, and A Karve. 2014. Adaptive, model-driven autoscaling for cloud applications. In *11th International Conference on Autonomic Computing (ICAC 14)*. USENIX Association, Philadelphia, PA, 57–64.

D Gmach, J Rolia, L Cherkasova, and A Kemper. 2007. Workload Analysis and Demand Prediction of Enterprise Data Center Applications. In *Workload Characterization, 2007. IISWC 2007. IEEE 10th International Symposium on*. IEEE, Washington, DC, USA, 171–180.

Tian Guo, Upendra Sharma, Prashant Shenoy, Timothy Wood, and Sambit Sahu. 2013. Cost-aware Cloud Bursting for Enterprise Applications. *Transactions on Internet Technology* (2013).

T. Guo, P. Shenoy, and H. Hakan. 2015. *GeoScale: Providing Geo-Elasticity in Distributed Clouds*. Technical Report UM-CS-2015-009. School of Computer Science, Univ. of Massachusetts at Amherst.

T. Guo, P. Shenoy, and H. Hakan. 2016. GeoScale: Providing Geo-Elasticity in Distributed Clouds. In *International Conference on Cloud Engineering (IC2E)*. IEEE.

J.L. Hellerstein, Fan Zhang, and P. Shahabuddin. 1999. An approach to predictive detection for service management. In *Proceedings of the Sixth IFIP/IEEE International Symposium on Integrated Network Management (INM)*. IEEE, Boston, MA, 309–322.

J. F. C. Kingman. 1961. The single server queue in heavy traffic. *Mathematical Proceedings of the Cambridge Philosophical Society* 57 (1961), 902–904.

Thomas Knauth and Christof Fetzer. 2011. Scaling Non-elastic Applications Using Virtual Machines. In *2011 IEEE 4th International Conference on Cloud Computing (CLOUD)*. IEEE, Washington, DC, 468–475.

Tim Kraska, Gene Pang, Michael J. Franklin, Samuel Madden, and Alan Fekete. 2013. MDCC: Multi-data Center Consistency. In *Proceedings of the 8th ACM European conference on Computer systems (EuroSys)*. ACM, New York, NY, USA, 113–126.

H A Lagar-Cavilla, J A Whitney, and A M Scannell. 2009. SnowFlock: rapid virtual machine cloning for cloud computing. In *Proceedings of the 4th ACM European conference on Computer systems (EuroSys)*. ACM, New York, NY, USA, 1–12.

Ewnetu Bayuh Lakew, Cristian Klein, Francisco Hernandez-Rodriguez, and Erik Elmroth. 2014. Towards Faster Response Time Models for Vertical Elasticity. In *Proceedings of the 2014 IEEE/ACM 7th International Conference on Utility and Cloud Computing (UCC '14)*. IEEE Computer Society, Washington, DC, USA, 560–565. DOI:http://dx.doi.org/10.1109/UCC.2014.86

H C Lim, S Babu, and J S Chase. 2010. Automated control for elastic storage. In *Proceedings of the 7th International Conference on Autonomic Computing (ICAC)*. ACM, New York, NY, USA, 1–10.

Zhenhua Liu, Minghong Lin, Adam Wierman, Steven H. Low, and Lachlan L.H. Andrew. 2011. Greening Geographical Load Balancing. In *Proceedings of the ACM SIGMETRICS Joint International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS '11)*. ACM, New York, NY, USA, 233–244. DOI:http://dx.doi.org/10.1145/1993744.1993767

Simon J Malkowski, Markus Hedwig, Jack Li, Calton Pu, and Dirk Neumann. 2011. Automated control for elastic n-tier workloads based on empirical modeling. In *Proceedings of the 8th ACM international conference on Autonomic computing (ICAC)*. ACM, New York, NY, USA, 131–140.

M. Mao and M. Humphrey. 2012. A Performance Study on the VM Startup Time in the Cloud. In *Cloud Computing (CLOUD), 2012 IEEE 5th International Conference on*. 423–430. DOI:http://dx.doi.org/10.1109/CLOUD.2012.103

P Marshall, K Keahey, and T Freeman. 2010. Elastic Site: Using Clouds to Elastically Extend Site Resources. In *Cluster, Cloud and Grid Computing (CCGrid), 2010 10th IEEE/ACM International Conference on*. IEEE Computer Society, Washington, DC, USA, 43–52.

Maxmind GeoIP Service 2013. IP Geolocation and Online Fraud Prevention: MaxMind. (2013). https://www.maxmind.com/en/home

Nathan D Mickulicz, Priya Narasimhan, and Rajeev Gandhi. 2013. To auto scale or not to auto scale. In *Proceedings of the 10th International Conference on Autonomic Computing (ICAC 13)*. 145–151.

F. J. A. Morais, F. V. Brasileiro, R. V. Lopes, R. A. Santos, W. Satterfield, and L. Rosa. 2013. Autoflex: Service Agnostic Auto-scaling Framework for IaaS Deployment Models. In *Cluster, Cloud and Grid Computing (CCGrid), 2013 13th IEEE/ACM International Symposium on*. 42–49. DOI:http://dx.doi.org/10.1109/CCGrid.2013.74

T.S.E. Ng and Hui Zhang. 2002. Predicting Internet network distance with coordinates-based approaches. In *INFOCOM*. 170–179.

Hiep Nguyen, Zhiming Shen, Xiaohui Gu, Sethuraman Subbiah, and John Wilkes. 2013. Agile: Elastic distributed resource scaling for infrastructure-as-a-service. In *Proceedings of the 10th International Conference on Autonomic Computing (ICAC 13)*. 69–82.

John O'Loughlin and Lee Gillam. 2014. Performance Evaluation for Cost-Efficient Public Infrastructure Cloud Use. In *Economics of Grids, Clouds, Systems, and Services - 11th International Conference, GECON 2014, Cardiff, UK, September 16-18, 2014. Revised Selected Papers.* 133–145. DOI:http://dx.doi.org/10.1007/978-3-319-14609-6_9

Shankaranarayanan P N, Ashiwan Sivakumar, Sanjay Rao, and Mohit Tawarmalani. 2014. Performance sensitive replication in geo-distributed cloud datastores. In *44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 240–251.

Pradeep Padala, Kai-Yuan Hou, Kang G. Shin, Xiaoyun Zhu, Mustafa Uysal, Zhikui Wang, Sharad Singhal, and Arif Merchant. 2009. Automated Control of Multiple Virtualized Resources. In *Proceedings of the 4th ACM European conference on Computer systems (EuroSys)*. ACM, New York, NY, USA, 13–26.

Marta Patiño Martinez, Ricardo Jiménez-Peris, Bettina Kemme, and Gustavo Alonso. 2005. MIDDLE-R: Consistent Database Replication at the Middleware Level. *ACM Trans. Comput. Syst.* 23, 4 (2005), 375–423.

Josep M. Pujol, Vijay Erramilli, Georgos Siganos, Xiaoyuan Yang, Nikos Laoutaris, Parminder Chhabra, and Pablo Rodriguez. 2010. The Little Engine(s) That Could: Scaling Online Social Networks. In *Proceedings of the ACM SIGCOMM 2010 conference on SIGCOMM*. ACM, New York, NY, USA, 375–386.

Shriram Rajagopalan, Dan Williams, Hani Jamjoom, and Andrew Warfield. 2013. Split/Merge: System Support for Elastic Execution in Virtual Middleboxes. In *Proceedings of the 10th USENIX conference on Networked Systems Design and Implementation (NSDI)*. USENIX Association, Berkeley, CA, USA, 227–240.

Mahadev Satyanarayanan, P. Bahl, R. Caceres, and N. Davies. 2009. The Case for VM-Based Cloudlets in Mobile Computing. *Pervasive Computing, IEEE* 8, 4 (2009), 14–23.

Jörg Schad, Jens Dittrich, and Jorge-Arnulfo Quiané-Ruiz. 2010. Runtime Measurements in the Cloud: Observing, Analyzing, and Reducing Variance. *Proc. VLDB Endow.* 3, 1-2 (Sept. 2010), 460–471. DOI:http://dx.doi.org/10.14778/1920841.1920902

Scryer 2013. Scryer: Netflix's Predictive Auto Scaling Engine. (2013). https://medium.com/netflix-techblog/scryer-netflixs-predictive-auto-scaling-engine-a3f8fc922270

Abhishek B. Sharma, Ranjita Bhagwan, Monojit Choudhury, Leana Golubchik, Ramesh Govindan, and Geoffrey M. Voelker. 2008. Automatic Request Categorization in Internet Services. *SIGMETRICS Perform. Eval. Rev.* 36, 2 (2008), 16–25.

Prateek Sharma, Stephen Lee, Tian Guo, David Irwin, and Prashant Shenoy. 2015. SpotCheck: Designing a Derivative IaaS Cloud on the Spot Market. In *Proceedings of the Tenth European Conference on Computer Systems (EuroSys '15)*. ACM, New York, NY, USA, 16:1–16:15. http://doi.acm.org/10.1145/2741948.2741953

Zhiming Shen, Sethuraman Subbiah, Xiaohui Gu, and John Wilkes. 2011. CloudScale: Elastic Resource Scaling for Multi-tenant Cloud Systems. In *Proceedings of the 2nd ACM symposium on Cloud computing (SoCC)*. ACM, New York, NY, USA, 5:1–5:14.

Rahul Singh, Upendra Sharma, Emmanuel Cecchet, and Prashant Shenoy. 2010. Autonomic Mix-aware Provisioning for Non-stationary Data Center Workloads. In *Proceedings of the 7th international conference on Autonomic computing (ICAC)*. ACM, New York, NY, USA, 21–30.

stopped vs terminated instances 2017. What is the difference between terminating and stopping an EC2 instance? (2017). http://docs.rightscale.com/faq/clouds/aws/Whats_the_difference_between_Terminating_and_Stopping_an_EC2_Instance.html

Stream Control Transmission Protocol 2013. Stream Control Transmission Protocol. http://tools.ietf.org/html/draft-natarajan-http-over-sctp-00. (2013).

The ObjectWeb TPC-W implementation 2005. The ObjectWeb TPC-W implementation. http://jmob.ow2.org/tpcw.html. (2005).

Omesh Tickoo, Ravi Iyer, Ramesh Illikkal, and Don Newell. 2010. Modeling Virtual Machine Performance: Challenges and Approaches. *SIGMETRICS Perform. Eval. Rev.* 37, 3 (Jan. 2010), 55–60. DOI:http://dx.doi.org/10.1145/1710115.1710126

R. Tolosana-Calasanz, J. Diaz-Montes, O. Rana, and M. Parashar. 2014. Extending CometCloud to Process Dynamic Data Streams on Heterogeneous Infrastructures. In *Cloud and Autonomic Computing (ICCAC), 2014 International Conference on*. 196–205. DOI:http://dx.doi.org/10.1109/ICCAC.2014.22

D. Tsoumakos, I. Konstantinou, C. Boumpouka, S. Sioutas, and N. Koziris. 2013. Automated, Elastic Resource Provisioning for NoSQL Clusters Using TIRAMOLA. In *Cluster, Cloud and Grid Computing (CCGrid), 2013 13th IEEE/ACM International Symposium on*. 34–41. DOI:http://dx.doi.org/10.1109/CCGrid.2013.45

B. Urgaonkar, P. Shenoy, A. Chandra, and P. Goyal. 2005. Dynamic Provisioning of Multi-tier Internet Applications. In *Second International Conference on Autonomic Computing (ICAC'05)*. 217–228. DOI:http://dx.doi.org/10.1109/ICAC.2005.27

Timothy Wood, Prashant Shenoy, Arun Venkataramani, and Mazin Yousif. 2007. Black-box and Gray-box Strategies for Virtual Machine Migration. In *Proceedings of the 4th USENIX Conference on Networked Systems Design and Implementation (NSDI'07)*. USENIX Association, Berkeley, CA, USA, 17–17. http://dl.acm.org/citation.cfm?id=1973430.1973447

Q. Zhang, Q. Zhu, M. F. Zhani, R. Boutaba, and J. L. Hellerstein. 2013. Dynamic Service Placement in Geographically Distributed Clouds. *IEEE Journal on Selected Areas in Communications* 31, 12 (December 2013), 762–772. DOI:http://dx.doi.org/10.1109/JSAC.2013.SUP2.1213008