ELASTIC RESOURCE MANAGEMENT IN DISTRIBUTED CLOUDS

A Dissertation Presented

by

TIAN GUO

Submitted to the Graduate School of the
University of Massachusetts Amherst in partial fulfillment
of the requirements for the degree of

DOCTOR OF PHILOSOPHY

September 2016

College of Information and Computer Sciences

ELASTIC RESOURCE MANAGEMENT IN DISTRIBUTED CLOUDS

A Dissertation Presented

by

TIAN GUO

Approved as to style and content by:

Prashant Shenoy, Chair

David Irwin, Member

Don Towsley, Member

Tilman Wolf, Member

James Allan, Department Chair
College of Information and Computer Sciences

DEDICATION

To my family, a pumpkeen, and a muffin.

ACKNOWLEDGMENTS

I would like to thank everyone who has encouraged and supported me during my journey for Ph.D. First and foremost, I want to express my gratitude towards my mom and dad who provide the best learning environment they possibly could from the very beginning. Their wholehearted support for my decision to continue graduate school in the USA is one of the few things that has kept me going.

During my time at UMass, I am fortunate to have Professor Prashant Shenoy as my research advisor. He had given me a tremendous amount of support ever since I joined LASS lab. In particular, I want to thank him for showing me the way to be a patient and effective researcher, as well as his unconditional help towards my job search. I am also very thankful to have Professor Tilman Wolf as my mentor and committee member. He gave me lots of very good advice that helped me overcome obstacles. In addition, I would like to thank the rest of my committee members, Professor David Irwin and Professor Don Towsley, for providing valuable feedbacks.

I would also like to thank K. K. Ramakrishnan, Vijay Gopalakrishnan, Seungjoon Lee, Hakan Hacigümüş, Jun Tatemura for mentoring and helping me during my internships at AT&T and NEC. With Prashant's support, both collaborations have continued after my internships and led to meaningful outcomes. Moreover, I would like to acknowledge all other research collaborators: Tim Wood, Upendra Sharma, Prateek Sharma, Stephen Lee, Xin He and Patrick Pegus II.

I have truly enjoyed my time at UMass. It would not be possible if it were not for an excellent CS community and all the volunteers for organizing various social events. Nevertheless, I am happy to be part of a collaborative LASS and a supportive CS Women

Group. Specifically, I would like to thank Leeanne Leclerc and Karren Sacco for their timely help to make sure my paperwork is organized and on track. In addition, I want to acknowledge Emmanuel Cecchet and Tyler Trafford for their help for managing both EC2 and server clusters.

I also have had an incredibly wonderful time in Amherst with an amazing group of friends. My lovely roommate Chang Liu, Yanjie Li, and XiaoLan Wang have proven to be excellent company for movies, food or everything. I went through the newcomer's experience with Wenzhe Wu, Pengyu Zhang, Weize Kong and Xiang Zhao when I first arrived in Amherst. And Bo Jiang, Xiaobing Xue, Wentian Lu, Tingxin Yan, Xiaozheng Tie have truly helped me so many times for so many things. Together with my foosball partner Stephen Lee, we had played an endless number of games with other foosball mates Mostafa Dehghan, Srinivasan Iyengar, Prateek, Sharma and Chang Liu. And that is something I will not forget. I probably had forced my company upon Jimmy Wright, Anton Bakalov, and Jason Beers so many times yet they never refuse to hang out. I also played quite a few board games with Tom and Fiona Helmuth and later their cute son Ben. Special thanks also go to Emma Strubell and Shaofang Wang who prove to be more than good workout buddies.

Last, I would like to acknowledge my dear pumpkeen and our naughty cat muffin for their supports and sacrifices throughout graduate school. My motivation for getting a Ph.D. peaks when pumpkeen got his and asked me to call him Doctor all the time. But yes. If I were to choose again, I will still work for a Ph.D. Not only because I have learned and experienced so much, but also because it is once in a lifetime opportunity.

ABSTRACT

ELASTIC RESOURCE MANAGEMENT IN DISTRIBUTED CLOUDS

SEPTEMBER 2016

TIAN GUO

B.S., NANJING UNIVERSITY

M.S., UNIVERSITY OF MASSACHUSETTS AMHERST

Ph.D., UNIVERSITY OF MASSACHUSETTS AMHERST

Directed by: Professor Prashant Shenoy

The ubiquitous nature of computing devices and their increasing reliance on remote resources have driven and shaped public cloud platforms into unprecedented large-scale, distributed data centers. Concurrently, a plethora of cloud-based applications are experiencing multi-dimensional workload dynamics—workload volumes that vary along both time and space axes and with higher frequency.

The interplay of diverse workload characteristics and distributed clouds raises several key challenges for efficiently and dynamically managing server resources. First, current cloud platforms impose certain restrictions that might hinder some resource management tasks. Second, an application-agnostic approach might not entail appropriate performance goals, therefore, requires numerous specific methods. Third, provisioning resources outside LAN boundary might incur huge delay which would impact the desired agility.

In this dissertation, I investigate the above challenges and present the design of automated systems that manage resources for various applications in distributed clouds. The intermediate goal of these automated systems is to fully exploit potential benefits such as reduced network latency offered by increasingly distributed server resources. The ultimate goal is to improve end-to-end user response time with novel resource management approaches, within a certain cost budget.

Centered around these two goals, I first investigate how to optimize the location and performance of virtual machines in distributed clouds. I use virtual desktops, mostly serving a single user, as an example use case for developing a black-box approach that ranks virtual machines based on their dynamic latency requirements. Those with high latency sensitivities have a higher priority of being placed or migrated to a cloud location closest to their users. Next, I relax the assumption of well-provisioned virtual machines and look at how to provision enough resources for applications that exhibit both temporal and spatial workload fluctuations. I propose an application-agnostic queueing model that captures the resource utilization and server response time. Building upon this model, I present a geo-elastic provisioning approach—referred as *geo-elasticity*—for replicable multi-tier applications that can spin up an appropriate amount of server resources in any cloud locations. Last, I explore the benefits of providing geo-elasticity for database clouds, a popular platform for hosting application backends. Performing geo-elastic provisioning for backend database servers entails several challenges that are specific to database workload, and therefore requires tailored solutions. In addition, cloud platforms offer resources at various prices for different locations. Towards this end, I propose a cost-aware geo-elasticity that combines a regression-based workload model and a queueing network capacity model for database clouds.

In summary, hosting a diverse set of applications in an increasingly distributed cloud makes it interesting and necessary to develop new, efficient and dynamic resource management approaches.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1
INTRODUCTION

Cloud platforms, from the outset, provide numerous benefits to hosted applications such as pay-as-you-go cost models and flexible, on-demand resource allocation. The virtue of on-demand server resource provisioning is magnified with the uptake and usage of virtualization techniques. Therefore, virtualized clouds have revolutionized traditional provisioning approaches by providing the ability to adjust server allocation in minutes instead of weeks or months.

Such benefits have nurtured cloud adoption, enabling applications with dynamic resource requirement to offload their computation as well as management tasks to cloud providers. As a result, cloud computing has become a popular paradigm for hosting a plethora of applications, ranging from e-commerce, news, social media and data analytics. Many of these applications are serving global users and as a result observe both temporal and spatial workload dynamics. Temporal variations are due to reasons such as time-of-day effect and spatial dynamics are caused by uncorrelated workload variations among different geographic regions.

To better handle applications with a geographically distributed workload, cloud platforms have become more distributed over the past few years. Distributed clouds offer a flexible choice of data center locations across the globe. However, running applications on distributed clouds can be a complex task. For example, application developers have to decide which cloud locations to place application replicas and how many servers to provision in order to fulfill performance goal. Therefore, in my thesis, I investigate automatic approaches to efficiently and dynamically manage server resources for various applications in distributed clouds. There are three key challenges associated with this research question and these challenges lie at the heart of my thesis research.

First, current cloud computing imposes restrictions on how resource management can be done. Although distributed cloud platforms provide multiple locations at application developers' disposal, most resource management abstractions exposed were designed to work within a single data center. In other words, there are no ready-to-use mechanisms that can assist cloud applications to take advantage of distributed clouds. Moreover, because cloud providers operate in isolation with different programming abstractions, it is challenging for cloud customers to pool cloud resources from multiple providers even if they offer complementary services.

Second, there is no silver bullet for managing resources for all cloud applications due to their distinct workload demand and service requirement. For example, applications might need to serve a mix of CPU-intensive, I/O-intensive, or network-intensive workloads and those mixes can vary over time. Further depending on current workload characteristics, we might need to adjust the magnitude or location of allocated resources accordingly. When making those adjustments, cloud platforms need to take into account both application performance requirements and cloud resource availability. In addition, cloud resource provisioning also has to circumvent the black-box restrictions—that is cloud providers do not have visibility to what applications are running inside rented VMs.

Third, it is necessary to dynamically manage resources for cloud-based applications. This is because applications experience workload variations that can be caused by numerous reasons such as time-of-day effects, regional events, user mobility or a steady workload shift from one location to another. As a result, these workload variations exhibit not only temporal but also spatial dynamics that can be either transient or long-lasting. Moreover, cloud applications usually do not provision server resources based on their peak workload to take advantage of cloud cost benefits.

1.1 Thesis Statement

The central theme of my thesis is to study novel resource management techniques that tackle the above challenges. Although most aforementioned challenges are inherently classical resource management problems, they are exacerbated by the need to operate within massively distributed clouds and the requirement to manage Internet-scale applications that are replicated in multiple cloud locations.

Specifically, I first look at how to automatically place latency-sensitive virtualized applications in heterogeneous distributed clouds. In addition, I also study how to seamlessly adjust server placement when application latency requirement changes using VM migration. I use virtual desktops—virtualized desktop PCs that host desktop applications and data—as example applications because they exhibit dynamic latency requirements.

Next, I study how to provision resources in distributed clouds to handle applications that observe both temporal and spatial workload variations for a given Service Level Agreements (SLAs), such as a 95^{th} percentile response time. I argue the need of providing geo-elasticity for replicable applications and examine the performance benefits. Geo-elasticity, the ability to provision server resources in *any* cloud locations dynamically, is achieved by combining queueing-based capacity models and provisioning algorithms that are designed for distributed clouds.

Last but not least, I investigate challenges of performing dynamic provision for database clouds that have emerged as a popular paradigm for hosting applications' backend tier. I focus on differentiating characteristics of database servers such as obliviousness to client workload distribution and a large amount of data state. Again, I am concerned about making provisioning decisions based on response time SLAs.

Thesis Statement: *Tailored resource management techniques for distributed clouds can significantly improve cloud applications' performance while fully exploit the cost benefits.*

1.2 Thesis Contributions

In this thesis, I propose novel techniques that combine analytical models and virtualization-based system mechanisms to tackle resource management problems that are compounded by distributed modern clouds and dynamic requirement of cloud applications. I build middleware systems that are tailored to needs of three different types of applications. These management systems can be integrated with public clouds and improve end-to-end response time when comparing to traditional elasticity approaches.

The central thesis of my dissertation is about achieving better performance and cost trade-off through novel resource management techniques in distributed clouds for modern cloud-based applications. To this end I design and implement three key systems that embody new algorithms and mechanisms for managing resources for different application classes:

- VMShadow which focuses on transparently and dynamically managing the location and performance of latency-sensitive VMs by using cost-aware placement algorithms, nested hypervisors, and WAN live VM migration technique [67, 68].
- GeoScale which dynamically provisions server resources for replicable applications in distributed clouds to satisfy SLA, using queueing-based capacity models and resource provisioning algorithms for distributed clouds [73].
- DBScale which allocates database servers in distributed database clouds to satisfy network and response time SLA, leveraging regression-based workload prediction, SLA-aware workload assignment, and queueing-based capacity models [71].

1.2.1 Automated Latency-aware Server Placement

Distributed clouds are well-suited to host latency-sensitive applications because they can host applications at locations close to users. Today, the choice of where to host one's application is made manually—a tedious and error-prone task. Manual placement does not work well because latency requirements vary for different applications and are also impacted by their clients' mobility.

Therefore I develop a system called VMShadow that automatically determines the best location for applications and transparently and seamlessly adjust these mappings over time based on changing application needs. Because cloud providers do not have visibility into applications running inside VM, VMShadow resorts to a black-box VM fingerprinting algorithm to determine VMs' latency-sensitivity and then uses a cost-benefit aware algorithm that chooses benefit-proportional VMs to a closer location at the lowest cost. Further, VMShadow implements VM migrations that are optimized for WAN and can transparently migrate VMs across different cloud providers, leveraging nested virtualization technique.

I evaluate VMShadow's efficacy using virtual desktops— desktop PC that runs inside VM and allows users to access their desktop applications and data files via a remote desktop protocol (and via thin clients—as an example application. Virtual desktops are ideal candidates for latency optimization due to their dynamic latency requirements. For example, users might switch between latency-sensitive applications (e.g. multi-player games) and latency-insensitive applications, e.g., mail or text editor. Or users might access virtual desktops with thin clients from a different location when traveling.

1.2.2 Dynamic Application Provisioning in Distributed Clouds

Today's cloud applications that serve a geographically diverse client base are observing both temporal and spatial workload variations. Temporal variations occur when workload volume fluctuates from time to time, due to effects such as time-of-day or flash crowd spikes. Spatial variations appear when workload volume differs across geographic regions, caused by reasons such as regional event spikes.

To handle applications' dynamic workload in order to meet a certain level of performance guarantee, cloud providers usually resort to two common approaches. The first approach involves using traditional elasticity techniques to vary server numbers at manually chosen distributed cloud sites. Application servers are allocated merely based on local workload volume, without regarding to replicas at other cloud locations. The second technique involves deploying the application at a single cloud location and using a global content distribution network (CDN) to serve user requests from different regions. However, this first technique has drawbacks when handling spatial variations. For instance, system administrators have to select a new cloud site manually if application sees a surge of requests from a new geographic location. The CDN-based method is not well suited for provisioning applications that serve dynamically-generated (non-cacheable) content or involve a significant amount of request processing.

To address the drawbacks of both techniques, I propose a new provisioning technique called geo-elasticity to dynamically provision server resources at *any* geographic location whenever needed. I implement geo-elasticity in a system called GeoScale that can effectively provision server resources in distributed clouds dynamically to handle workload variations caused by geographically distributed clients. GeoScale monitors client-facing applications' global workload distribution and distributes application workload into the closest cloud location. Then, GeoScale employs an empirical-driven queueing model to proactively provision required application server capacity at each cloud site. GeoScale also can react to sudden workload changes or predicted errors by quickly provisioning server resources within a data center.

1.2.3 Geo-elastic Dynamic Provisioning for Database Servers

The task of maintaining and managing backend databases is time-consuming and even challenging as cloud applications become larger scale and more geo distributed. Therefore, more applications are starting to host their backend tiers using Database-as-a-Service (DBaaS) clouds. DBaaS clouds provide basic autoscaling and elasticity properties to handle some level of workload variations. However, such system lacks support for handling applications with geographic user workload, and the temporal and spatial variations associated with it.

Providing geo-elasticity for DBaaS clouds exhibits several key challenges compared to our previous application-agnostic approach. For example, database servers do not directly observe client workload distributions that are obscured by front-end tiers. In addition, because front-end tier is dependent on database servers, we need to coordinate provisioning decisions between both tiers for better end-to-end response time.

To tackle these new challenges, I present DBScale, a system that specifically designed to dynamically provision database servers to fulfill both network and server response SLA. DBScale predicts database spatial workload variations using regression and provisions for both CPU and I/O resources using an open feedback queuing network. DBScale supports multiple consistency policies and mechanisms, such as batch update and master-slave configuration, to suit different application needs. DBScale also consider performance and cost trade-offs when provisioning in distributed clouds with different pricing models.

1.3 Thesis Outline

The remainder of this thesis is structured as follows. Chapter 2 provides background and related work on virtualized cloud platforms, cloud-based applications, and dynamic resource provisioning techniques. Chapter 3 presents the problem of virtual machine placement in distributed clouds, and introduces VMShadow for automatically optimizing the location and performance of latency-sensitive VMs in distributed clouds. Chapter 4 describes the importance of handling both temporal and spatial workload dynamics of applications with geo-distributed client workload, and proposes GeoScale that implements geo-aware provisioning approaches. This is followed in Chapter 5 with a discussion of challenges and importance of implementing geo-aware provisioning in Database clouds, and introduces DBScale that employs a regression-based model and a queueing-network model to perform database resource provisioning in the distributed clouds. Finally, Chapter 6 summarizes thesis contributions and concludes with some future directions.

CHAPTER 2
BACKGROUND AND RELATED WORK

In this chapter, I introduce the necessary background on cloud platform development, cloud-based applications and existing approaches for dynamic resource provisioning. The interplay of increasingly complex cloud offerings and more resource demanding large-scale applications require rethinking how to efficiently manage resources, either to meet certain performance expectations or to reduce operation costs.

2.1 Virtualized Cloud platforms

Modern data centers can contain networked servers in the scale of tens of thousands. Managing such large scale data centers is no trivial tasks and much research has focused on designing data centers to be more reliable, secure, and energy efficient [74, 131, 135, 162]. As a result, today's data centers have become a popular venue for hosting enterprise ITs and internet applications.

Cloud computing, a popular paradigm for hosting various applications, offers computing as a utility and allows customers to rent data center resources [16, 31, 163]. Public commercial clouds benefit from the economy of scale and can offer resources to their customers at a reasonably cheap price. Depending on the level of abstractions exposed, cloud platforms can be categorized into Infrastructure-as-a-service (IaaS) [11], Platform-as-a-service (PaaS) [63], and Software-as-a-Service (SaaS) [13, 64, 144]. In addition, cloud platforms also provide a pay-as-you-go pricing model that allow customers to pay only for their usages. In another word, customers do not need to worry about up-front capital investment and can request cloud resources whenever they need. To support a large number of customers and create an illusion of unlimited resources, cloud providers have resorted to virtualization techniques [19,91] to achieve statistical multiplexing [113]. Virtualization, in essence, is a layer of abstraction that hides the underlying hardware and software resources and can provide a certain level of resource isolation.

Over the past decade, cloud platforms have matured greatly in terms of service offerings and have become increasingly distributed. Distributed clouds offer customers a flexible choice of locations to host their applications and have the promise of delivering low network latency. The increasing number of service offerings and the geographic expansions make cloud platforms very attractive [84, 105, 154], but at the same time more complicated to use [9, 43, 100, 124, 149, 150, 158]. For example, different cloud platforms might require slightly different machine image formats or expose different APIs. This limits customers' choice of which cloud providers to choose from or might require non-trivial works to use resources from different cloud providers [44, 69]. In addition, mostly out of security concerns and the multi-tenancy models, cloud providers do not provide their customers privileged accesses, which include hypervisor accesses. This limits and restricts the types of resource management a customer can perform. For example, this means customers can not perform VM migration [28,39], which is a commonly used technique to consolidate live systems transparently from end users.

This thesis proposes new resource management techniques that are driven by application-specific models to guide the usages of virtualization techniques and cloud exposed APIs [67, 68]. Our proposed techniques allow customers to pool together resources from distributed clouds and seamlessly perform dynamic resource management tasks to fulfill performance requirements.

2.2 Internet-Scale Geo-replicated Applications

Today's cloud platforms are home to a plethora of different types of applications, all having unique characteristics. Depending on their latency requirements, cloud-based applications can be categorized into either interactive or batch.

Interactive applications usually refer to the type of applications that interact with a human who has a certain response time requirement. For example, thin client computation [18, 146] where client relies on remote server resources for web browsing through protocols such as Remote Desktop Protocol (RDP), and e-commerce [12, 54] websites are examples of traditional interactive applications. Over the past decades, we observe a more diverse type of interactive applications that come with the ubiquitous of mobile devices [147, 169] and smart embedded devices, such as smart Internet of Things (IoT) sensors [93, 106]. The widespread adoption of computing devices has created workloads that are more geographically distributed and dynamic for interactive applications.

Batch applications are often referred to *best-effort* workload and have relaxed latency requirement. Aside from traditional batch applications, e.g. analyzing cluster performance or understanding online users behaviors [96,159], we are seeing a slightly different flavor of batch applications that are associated with large-scale sensor deployments. These sensors, either from mobile devices or IoT devices, collect a large amount of data from physical surroundings when environment changes in some way [82, 138]. Therefore, modern batch applications are also experiencing a shift of workload characteristics—from a static and centralized workload to a dynamic and distributed one.

In all, today's cloud platforms have enabled applications with unprecedented scale and distributed user bases [56]. These emerging Internet-scale applications are increasingly serving geographic users [38, 121] and therefore experience both temporal and spatial workload dynamics [73]. In this thesis, I study how to manage resources for three types of virtualized cloud applications with new workload dynamics to meet a certain level of performance [71, 73].

2.3 Dynamic Resource Provisioning

Dynamic resource provisioning—the ability to adjust resource allocation based on workload variation— is essential for today's cloud-based applications and a key benefit provided by modern clouds. Dynamic provisioning [25,151], also known as elasticity, helps applications with varying workload meet their Service Level Agreements (SLAs) without the need for over-provisioning. For example, an SLA could be specified in terms of a high percentile (95^{th}) response time [162].

To enforce an SLA, one first needs to detect any violation [53, 116] and then devise new provisioning plans [151, 172]. Prior efforts have studied different approaches to efficiently identify SLA violations empirically [10, 107] or proposed predictive models to estimate any violations [57, 142]. In addition, an extensive body of work has looked at developing performance models for different applications [61, 75, 119], especially for web applications [22, 85]. Those performance models are intended to capture the relationship between server resources and corresponding response time, given certain client workload mix. Researchers have proposed various approaches, such as queueing-based models [32, 119, 162], classical feedback control theory [50, 101], or data-driven machine learning models [86, 110] as means to build such models that are in turn used as a basis of provisioning.

Virtualization techniques such as OS virtualization [103] and para virtualization [19, 91], have made provisioning much easier by providing resource isolation mechanisms [74, 81] and support for migration [20, 28, 29, 39, 49]. However, most prior work on dynamic resource provisioning [32,119,162] has only looked at virtualized resources within a single data center. This is in part because cloud platforms used to offer fewer locations and the absence of techniques such as efficient VM migration across WAN [21,165,167].

Given the increasingly distributed nature of clouds and emerging complexity of application demands, in this thesis, I look at new ways to perform dynamic resource provisioning. Specifically, I use model-driven approaches to building middleware systems that can be integrated to current cloud platforms. These systems implement end-to-end provisioning tasks in distributed clouds and also consider the trade-off between performance and cost.

CHAPTER 3

LATENCY-AWARE VIRTUAL MACHINE PLACEMENT

Distributed clouds that offer choice of data center locations enable efficient hosting of latency-sensitive applications. However, decisions of which location to house application and when to adjust placement are left entirely to application developers. In addition, those applications exhibit dynamic workload characteristics that make manual decisions less ideal and more error-prone. In this chapter, we present a system called VMShadow that automatically optimizes the location and performance of latency-sensitive VMs in distributed clouds. VMShadow combines algorithms and virtualization techniques for understanding VM's latency requirements and enabling efficient WAN migration on public clouds. We evaluate the efficacy of VMShadow using virtual desktops—a virtualized application that exhibits dynamic latency requirement.

3.1 Motivation

Hosting online applications on cloud platforms has become a popular paradigm. Applications ranging from multi-tier web applications, gaming and individual desktops are being hosted out of virtualized resources running in commercial cloud platforms or in a private cloud run by enterprises. The wide range of supported applications have diverse needs in terms of computation, network bandwidth and latency. To accommodate this and to provide geographic diversity, cloud platforms have become more *distributed* in recent years. Many cloud providers now offer a choice of several locations for hosting a cloud application. For instance, Amazon's EC2 cloud provides a choice of eleven global locations across four continents. Similarly, enterprise-owned private clouds are distributed across a

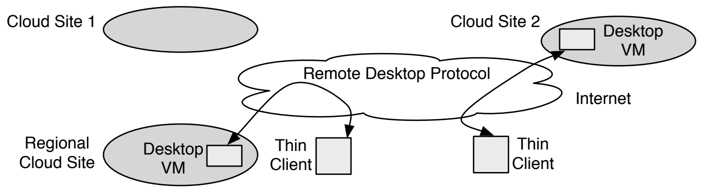


Figure 3.1: **Illustration of a distributed desktop cloud.** Users can access their desktop VMs hosted in the regional cloud sites or global cloud sites through remote desktop protocol such as virtual network computing(VNC).

few large data centers as well as many smaller branch office sites. Such distributed clouds enable application providers to choose the geographic region(s) best suited to the needs of the particular application.

A concurrent trend is the growing popularity of virtual desktops (VDs) where the desktop PC of a user is encapsulated into a virtual machine (VM) and this VM is hosted on a remote server or the cloud; users then access their desktop applications and their data files via a remote desktop protocol such as VNC (and via thin clients). This trend—known as virtual desktop infrastructure (VDI)—is being adopted by the industry due to numerous benefits. First of all, virtualizing desktops and hosting them on remote servers simplifies the IT manager's tasks, such as applying security patches, performing data backups. Secondly, it also enables better resource management and reduces costs, since multiple desktop VMs can be hosted on a high-end server, which may still be more cost-effective than running each desktop on a separate PC. At the same time, in addition to their use for business purposes in enterprise settings, desktop VMs hosted in the cloud are beginning to be offered for consumer use. Notably, commercial services such as *Onlive Desktop* even offer a "free Windows PC in the cloud" that can be accessed from tablets.

The confluence of these trends—the emergence of both distributed clouds and popularity of virtual desktops—creates both opportunities and challenges. Today a virtual desktop provider needs to *manually* choose the best data center location for each end-user's virtual desktop. In the simplest case, each VD can be hosted at a cloud data center location that is *nearest* to its user (owner). However such manual placement becomes increasingly challenging for several reasons. To start with, while this may be straightforward in cloud platforms that offer a choice of a few locations (e.g., with Amazon, one would host all VDs for US east coast users at the east coast data center), it becomes progressively more challenging as the number of locations continues to grow in highly distributed clouds that already offer a large number of locations. Additionally, different data center locations may have varying hosting capacities. Regional locations may have comparatively smaller capacities than the "global" locations; this implies that naïvely placing all VDs from a location at their nearest regional site may not be practical due to resource constraints. More interestingly, not all VDs are sensitive to network latency. Therefore, users may not see significant performance improvement when their VDs are placed at the closet location. Specifically VDs that run latency-sensitive applications such as multi-player games or video playbacks will see *disproportionately greater benefit* from nearby placement compared to those that run simple desktop applications such as e-mail or a text editor. Further, VDs will see dynamic workloads—users may choose to run different applications at different times and this workload mix may change over time. In addition, users may themselves move locations, particularly those that access their VDs via mobile devices, or go from work to home. This set of challenges implies that a *static* and *manual* placement of VDs at the nearest cloud location may not always be enough or even feasible. We argue that the cloud platform should incorporate intelligence to *automatically* determine the best location for hosting each application, and transparently and seamlessly adjust such mappings over time with changing application needs.

Towards this end, we present VMShadow, a system that transparently and dynamically manages the location and performance of virtual desktops in distributed clouds. Our system automates the process of placing, monitoring and migrating cloud-based virtual desktops across the available cloud sites based on the location of users and latency-sensitivity of the applications. VMShadow performs black-box virtual desktop fingerprinting to assign different latency-sensitive scores based on the packet-level statistics collected from hypervisor. It then employs either an ILP algorithm or a cost-aware greedy algorithm, depending on the problem scale, to pick new locations for latency-sensitive VMs that balance the cost-benefit tradeoffs. Both algorithms are able to make placement decisions while considering the existing virtual desktop locations. VMShadow executes the new VM placement plan using live migration across the WAN, optimized by techniques such as delta encoding and content-based redundancy elimination [167]. More specifically, to migrate a VM to a new location across the WAN, VMShadow first live migrates the disk and memory state of a VM using the optimized WAN live migration. In the scenario where the public IP address of the virtual desktop changes, VMShadow seeks to maintain existing TCP connections between the clients and server VMs by using connection proxies. The connection proxies communicate the changes of IP address and port number and rewrite the network packet headers to ensure that the migration is transparent to applications. As a result, VMShadow allows a client to stay connected irrespective of whether the server or even the client moves, whether or not the client or server is behind a NAT, and whether network entities such as routers and NAT devices are cooperating.

Although VMShadow is designed to be a general platform, in this chapter we employ it primarily to optimize the performance of *desktop clouds*, as illustrated in Figure 3.1. Desktop clouds offer an interesting use-case for VMShadow since desktops run a diverse set of applications, not all of which are latency-sensitive. We implement a prototype of VMShadow in a nested hypervisor, i.e., Xen-Blanket [165], and experimentally evaluate its efficacy on a mix of latency-sensitive multimedia and latency-insensitive VDs running on a Xen-based private cloud and Amazon's EC2 public cloud. Our results show that VMShadow's black-box fingerprinting algorithm is able to discriminate between latency-sensitive and insensitive virtual desktops and judiciously moves only those VDs that see the most benefit from migration, such as the ones with video activity. For example, VDs with video playback activity see up to 90% improvement in refresh rates due to VMShadow's

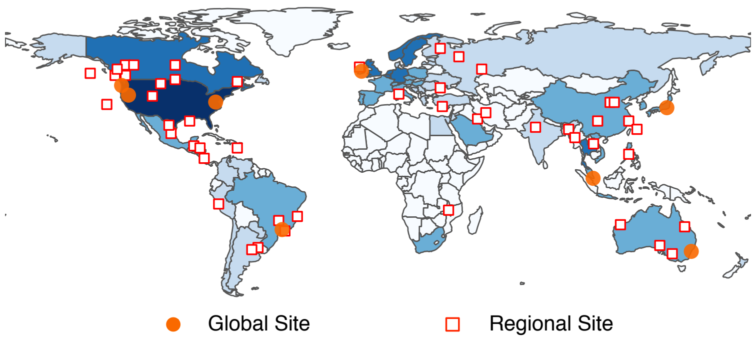


Figure 3.2: **A hypothetical distributed cloud.** Circles denote global cloud location while squares denote regional sites.

automatic location optimizations. We demonstrate the live migration of VDs across Amazon EC2 data centers with trans-coastal VM migrations of Ubuntu desktops with 1.4GB disk and 1GB RAM take 4 minutes. We show that our connection migration proxy—based on dynamic rewriting of packet headers— imposes an overhead of 13μs per packet. Our results show the benefits and feasibility of VMShadow for optimizing the performance of multimedia VDs, and more generally, of a diverse mix of virtual machine workloads.

3.2 Background

An *infrastructure-as-a-service (IaaS)* cloud allows application providers to rent servers and storage and to run any virtualized application on these resources. We assume that our IaaS cloud is highly distributed and offers a choice of many different geographic locations ("cloud sites") for hosting each application. For example, in Amazon's EC2, an application provider may choose to host their application at any of their global locations such as Virginia and Singapore. We assume that future cloud platforms will be even more distributed and offer a much larger choice of locations (e.g., one in each major city or country). A distributed cloud is likely to comprise heterogeneous data centers —some locations or sites will be very large ("global") data centers, while many other regional sites will comprise smaller data centers as depicted in Figure 3.2. Such a heterogeneous distributed cloud maps well to how public clouds are likely to evolve—comprising of a few large global sites that offer economies of scale, while smaller regional sites offer greater choice in placing latency-sensitive applications. The model also maps well to distributed private clouds run by enterprises for their own internal needs—typical enterprise IT infrastructure consists of a few large backend data centers (to extract economies of scale by consolidating IT applications) and several smaller data centers at branch office locations (which host latency-sensitive applications locally).

We focus our attention on a single application class, namely cloud-based desktops (also referred to as *desktop clouds* that host a large number of VDs in data centers) that run on virtual machines (VMs) in the cloud data center. Each desktop VM represents a "desktop computer" for a particular user. Users connect to their desktop from a thin client using remote desktop protocols such as VNC or Windows RDP. We treat the VMs as black boxes and assume that we do not have direct visibility into the applications running on the desktops; however since all network traffic to and from the VM must traverse the hypervisor or its driver domain, we assume that it is possible to analyze this network traffic and make inferences about ongoing activities on each desktop VM. Note that this black-box assumption is necessary for public clouds where the VDs belong to third party users.

To provide the best possible performance to each desktop VM, the cloud platform should ideally host each VM at a site that is nearest to its user. Thus a naïve placement strategy is to determine the physical location of each user (e.g., New York, USA) and place that user's VM at the geographically nearest cloud site. However, since nearby regional cloud cites may have a limited server capacity, it may not always be possible to accommodate all VDs at the regional site and some subset of these desktops may need to be moved or placed at alternate regional sites or at a backend global site. Judiciously determining which VDs see the greatest benefit from nearby placement is important when making these decisions.

Fortunately, not all desktop VMs are equal in terms of being latency-sensitive. As we show in Section 3.4, the performance of certain desktop applications is significantly impacted by the geographic distance between VD and its user. While for other applications, the location is not a major factor for good performance. In particular, network games require high interactivity or low latencies; video playback or graphics-rich applications require high refresh rates or high bandwidth while using remote desktop protocol. Such applications see the greatest benefits from nearby placement since this yields low round-trip time between the user and her VM or ensures higher bandwidth or less congested links. Thus identifying the subset of desktops that will benefit from closer placement to users is important for good end-user experience. Further since users can run arbitrary applications on their desktops, we assume that VM behavior can change over time (in terms of its application mix) and so can the locations of users (for instance, if a user moves to a different office location). The cloud platform should also be able to adjust to these dynamics.

3.3 VMShadow Design Goals

Our goal is to design VMShadow, a system that optimizes the performance of cloud-based VDs via judicious placement across different sites in a distributed cloud. VMShadow seeks to dynamically map *latency-agnostic* VMs to larger back-end sites for economies of scale and *latency-sensitive* ones to local (or nearby regional) sites for a better user experience. To do so, our system must fingerprint individual VMs' traffic in order to infer their degree of latency-sensitivity while respect the black-box assumption. Our system must then periodically determine which group of VMs need to be moved to new sites based on recent changes in their behaviors and then transparently migrate the disk and memory state of these desktops to new locations without any interruption. Typically VDs running latency-sensitive applications, such as games or multimedia applications (video playback),

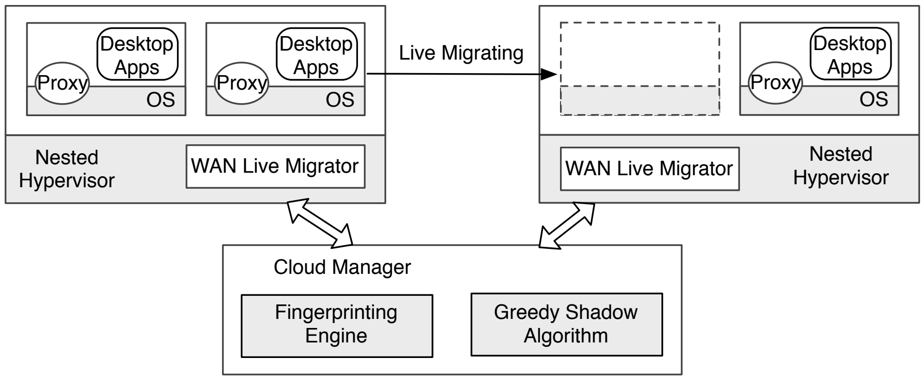


Figure 3.3: **VMShadow Architecture.** The central cloud manager performs latency-sensitivity fingerprinting for each desktop VM and employs a greedy algorithm that migrates highly latency-sensitive VMs to closer cloud sites at least cost. For each hypervisor, we implement a live migration technique that achieves WAN-specific optimizations. For each desktop VM, we use proxy to transparently migrate TCP connection.

are the best candidates for such migration. Finally, our system should transparently address networking issues such as IP address changes when a VM is moved to a different data center location, even if the client or desktop is behind a network address translation (NAT) device.

VMShadow architecture. Figure 3.3 depicts the high-level architecture of VMShadow. Our system achieves the above goals by implementing four components: (i) a black-box VM fingerprinting technique that infers the latency-sensitivity of VMs by analyzing packet-level network traffic, (ii) an ILP and an efficient greedy algorithms that judiciously move highly latency-sensitive VMs to their ideal locations by considering latency, migration cost as well as latency reduction. (iii) an efficient WAN-based live migration of a VM's disk and memory state using WAN-specific optimizations, and (iv) a connection migration proxy that ensures seamless connectivity of currently active TCP connections—despite IP address changes—in WAN live migration. We describe the design of each of these components in Sections 3.4 – 3.6 and the implementation of VMShadow in Section 3.7.

	LAN (Second)				US-WEST (Second)			
	Avg	Max	Min	Std	Avg	Max	Min	Std
Text Editor	0.191	0.447	0.094	0.0705	0.288	0.605	0.149	0.121
Web Browser	0.059	0.269	0.009	0.050	0.174	0.472	0.099	0.071

Table 3.1: **Statistics of VNC frame response time for latency-insensitive applications.** For both online text editing and web browsing, users see acceptable latencies [125].

3.4 Black-box VM Latency Fingerprinting

VMShadow uses a black-box fingerprinting algorithm to determine each virtual desktop's latency-sensitivity score. This approach is based on the premise that certain applications perform well, or see significant performance improvements, when located close to their users. We first describe our observations of distinct network characteristics of latency-sensitive and insensitive applications running inside virtual desktop.

Latency-sensitive applications. Consider desktop users that play games; clearly the closer the VD is to the user, the smaller the network round-trip-time (RTT) between the desktop and the user's thin client. This leads to better user-perceived performance for such latency-sensitive gaming. Similarly, consider users that watch video on their virtual desktops—either for entertainment purposes from sites such as YouTube or Netflix, or for online education via Massive Online Open Courses(MOOCs) or corporate training. Although video playback is not latency-sensitive per se, it has a high refresh rate (when playing 24 frames/s video, for example) and also causes the remote desktop protocol to consume significant bandwidth. As the RTT between the thin client display and the remote VD increases, the performance of video playback suffers (see Figure 3.4). Many VNC players, for instance, perform pull-based screen refresh and each refresh request is sent only after the previous one completes. Hence the RTT will determine the upper bound on the request rate. Thus if the RTT is 100ms (not unusual for trans-continental distances in the US), such a player is limited to no more than 10 refresh requests per second, which causes problems when video playback requires 20 or 30 frames/second. In this case, locating the

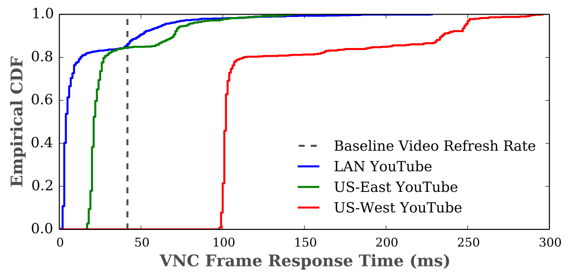
VD closer to the end-user yields a lower RTT and potentially higher refresh rates and better performance. This is depicted in Figure 3.4 which shows a CDF of the VNC refresh rate of a client in Massachusetts when the desktop VM is on a LAN, or at US-East and US-West sites of Amazon EC2. More specifically, in Figure 3.4a, when watching YouTube, we observe about 82% of the frame requests of LAN local streaming are served in less than 41.7 ms—the update frequency for 24 FPS video. However, in Figure 3.4b, when a user is watching video on the virtual desktop hosted at US-West about 70% VNC frames are updated after more than 125 ms, with the potential loss of video frames. Thus, proper placement of desktops with video applications significantly impacts user-perceived performance; similar observations hold for other application classes such as network games or graphics-rich applications.

Latency-insensitive applications. In contrast to the above, applications such as simple web browsing and word processing as shown in Table 3.1 are insensitive to latency. Although these are interactive applications, user-perceived performance is not impacted by larger RTT since they are within the human *tolerance for interactivity* (as may be seen by growing popularity of cloud-based office applications such as Google docs and Office 360).

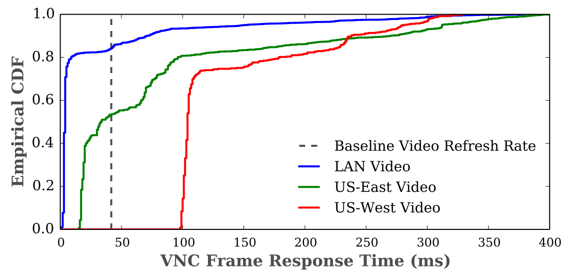
Based on our observations of different latency requirements of VD applications, we conclude that different VDs will have different degrees of latency-sensitivity depending on the collection of applications they run. Next, we will describe VMShadow's black-box latency fingerprinting algorithm that recognizes this diversity.

3.4.1 Black-box Fingerprinting Algorithm

The goal of our black-box fingerprinting algorithm is to assign a latency sensitive score S to a virtual desktop based on its network characteristics without explicitly looking inside each VM. For a particular virtual desktop, the end user (via a thin client) can run arbitrary applications simultaneously. This indicates virtual desktops will exhibit dynamic latency requirements and these requirements will be reflected in the network traffic. For a total of



(a) Watching online video.



(b) Watching local video.

Figure 3.4: **CDF comparisons of VNC frame response times for latency-sensitive applications.** Users have a better experience with watching videos when the VNC server is closer.

N virtual desktops, we are mainly interested in finding the relative latency scores for each one. We use the normalized network traffic throughput h^*, the normalized remote desktop protocol throughput e^*, and the latency-sensitive percentage of normal internet traffic f^* to infer the latency score. The rationale behind our choice of these three indicators is as follows. First, a "chatty" virtual desktop is more likely to be sensitive to placement. Second, a virtual desktop that interacts with thin client frequently is more likely to benefit from closer placement. Third, based on our observations, a virtual desktop that runs graphic-rich applications, e.g. videos, is more likely to benefit from placement optimization.

To calculate these values for i^{th} desktop VM, we collect packet-level traffic traces for a time window of size T_i. The traces are collected by observing the incoming and outgoing traffic of a VM from the driver domain of the hypervisor (e.g. Xen's dom0). We denote the total network traffic observed for i^{th} VM as H_i and obtain the throughput h_i and normalized throughput h_i^* in Equation (3.1) and Equation (3.2).

$$h_i = \frac{H_i}{T_i} \tag{3.1}$$

$$h_i^* = \frac{h_i}{\hat{h}} \tag{3.2}$$

$$\hat{h} = \max\langle h_1, h_2, \ldots h_N \rangle$$

Next, we identify the total amount of remote desktop traffic E_i using the default ports, e.g., port 5901 (server port for the VNC protocol) or port 3389 (server port for the Windows RDP protocol). Similarly, we can calculate the protocol throughput e_i and normalized throughput e_i^* in Equation (3.3) and Equation (3.4).

$$e_i = \frac{E_i}{T_i} \tag{3.3}$$

$$e_i^* = \frac{e_i}{\hat{e}} \tag{3.4}$$

$$\hat{e} = \max\langle e_1, e_2, \ldots e_N \rangle$$

Last, to calculate the latency-sensitive percentage of internet traffic for i^{th} virtual desktop f_i^*, we first use our list of latency-sensitive server ports and addresses to identify the amount of latency-sensitive traffic F_i and then obtain f_i^* as in Equation (3.5).

$$f_i^* = \frac{F_i}{(H_i - E_i)} \tag{3.5}$$

To obtain the list of latency-sensitive server ports and addresses, we assume that the administrator provides this initial information based on prior experience. Notably, "youtube.com" or other online video streaming sites would be included in the initial list. VMShadow then evolves this list by adding or removing information from the list using classification results. Currently,VMShadow uses K-nearest-neighbors (KNN) classifier to label each new TCP connection as latency-sensitive or not. When building up KNN model, we represent each TCP connection as a d-dimension feature vector[1] $\mathbf{g} \in \mathbf{R}^d$ and classify the new connections as latency sensitive or not based on majority vote of its K nearest neighbors. Here, we choose K to be 3. To collect training data, we manually run various selected applications (for which we know their latency-sensitivity) inside virtual desktops and collect the feature vector for each connection. If a new connection is labeled latency-sensitive, VMShadow will then add the corresponding server port and address to the maintained list. Otherwise, the information will be removed from the maintained list if exists. Finally, we calculate the desktop VM's latency scores S in Equation (3.6).

$$S = w_h * h^* + w_e * e^* + w_f * f^* \tag{3.6}$$

where $W = \langle w_h, w_e, w_f \rangle$ represents the weights we assign to each normalized term. Currently, we use $W = \langle \frac{1}{3}, \frac{1}{3}, \frac{1}{3} \rangle$.

Thus, VMShadow keeps track of each virtual desktop's latency score for a time-window of length M, denoted as $\langle S(t-M), S(t-M+1), \cdots S(t) \rangle$ and uses the moving average $\frac{1}{M} \sum_{j=0}^{M} S_i(t-j)$ to represent the rank of i^{th} VD. VDs with consistently high rank become candidates for latency optimization—in cases where they are not already in the best possible data center location—as described next.

3.5 VMShadow Algorithm

In this section, we explain VMShadow's algorithm that enables virtual desktop deployments to "shadow", i.e., follow their users through intelligent placement. Given a distributed cloud with K locations, placements of N active desktop VMs and their latency-sensitive ranks $\langle S_1, S_2 \ldots S_N \rangle$, our shadowing algorithm periodically employs the following steps.

[1]Example features include throughput, connection duration or inter-packet latency.

Step 1. Identify potential candidates to move. VMShadow determines which VMs are good candidates for migration to a different location—either relocated to a closer cloud location or evicted from a regional site with limited resource. We define a high threshold S_{up} and a corresponding low threshold S_{lo} to identify VMs for either relocation or eviction. Note that we can obtain S_{up} and S_{lo} by setting up two benchmark virtual desktops, one that runs latency-sensitive applications and the other that runs latency-insensitive applications, and measure their network traffic. In particular, for i^{th} VM with a latency score of S_i, if $S_i > S_{up}$, it becomes a candidate for relocation; if $S_i < S_{lo}$, it is a candidate for eviction. As an example, a desktop VM with consistent video or gaming activities will become a candidate for optimization and those that have not seen such activities for long periods will become candidates for eviction.

Step 2. Determine new locations for each candidate. For each VM that is flagged as a candidate for relocation, VMShadow next identifies potential new cloud locations for that VM. To do so, it first determines the location of the user for that desktop VM (by performing IP geo-location of the VNC thin client's IP address [89]). It then identifies the k closest cloud sites by geographic distance and then computes the network distance (latency) of the user to each of these k sites. These sites are then rank-ordered by their network distance as potential locations to move the VM. Candidate VMs that are already resident at the "best" cloud site are removed from further consideration.

Step 3. Analyze VMs' cost-benefit for placement decision. For each candidate VM for relocation, VMShadow performs a cost-benefit analysis of the possible move. The cost of a move to a new location is the overhead of copying the memory and disk state of the VM from one location to another over the WAN. The benefit of such a move is the potential improvement in user-perceived performance (e.g., latency reduction). In general, the benefit of a move is magnified if the VM has a relatively small disk and memory footprint(cost) and a high latency-sensitive rank. Since regional/local cloud sites may have smaller capacities, VMShadow must perform the cost-benefit analysis to identify VMs that yield the greatest benefit at the least cost. Also VMShadow could evict low-ranked VMs to free up resources when necessary. We formulate the above problem as an Integer Linear Program (ILP) optimization in Section 3.5.1. Since an ILP can have an exponential running cost, we also devise an efficient greedy heuristic that incorporates cost-benefit trade-off in Section 3.5.2.

Step 4. Trigger VMShadow Migrations. The final step involves triggering migrations of the disk and memory state of VMs to their newly chosen locations. Our approach is built upon prior work CloudNet [167] that provides an end-to-end and optimized solution for live migrating virtual machines in the context of Wide Area Network. Our work extends CloudNet in two ways. First, we re-implement all optimizations inside a nested hypervisor, i.e. Xen-Blanket [165]. This is an important extension because it provides us the flexibility to live migrate virtual machines between two nested hypervisors, eliminating the needs for hypervisor privilege and cloud provider lock-in. In another words, VMShadow can seamlessly migrate virtual machines between different cloud platforms with geographically diverse data center locations. Second, we propose an alternative method to ensure TCP connections staying active after VM migrations. Unlike CloudNet [167], our method does not require specialized hardware support. Our VM and connection migration techniques are detailed in Section 3.6.

3.5.1 VMShadow ILP Placement Algorithm

In this section, we describe our ILP algorithm that places above-threshold VMs—virtual desktops that have latency-scores larger than S_{up}—in better cloud locations by considering the migration cost and latency reduction. Assume we have access to K data center locations, and a total of J server hosts. Our goal is to pick the ideal data center for all I VDs within the resource constraints of the hosts. Essentially, we can translate the problem into selecting hosts with different network latencies to run the VDs.

Let $\langle U_j, M_j, D_j, N_j \rangle$ denote the available resource vector of Host$_j$ representing CPU cores, memory, disk and network bandwidth respectively. Note such resource vectors not only account for currently available server resources, but also include resources that are occupied by below-threshold VMs. These insensitive VMs are ones with scores lower than S_{lo} and are candidates for eviction. Representing available server resources in the above way enables us to prioritize the need of high latency-sensitive VMs in resource-constrained regional sites by moving insensitive VMs to a larger/global site. Similarly, let $\langle c_i, m_i, d_i, n_i \rangle$ denotes the resource vector of VM_i.

Let A_{ij} be the binary indicator such that:

$$A_{ij} = \begin{cases} 1 & \text{if } i^{th} \text{ VM is on } j^{th} \text{ host} \\ 0 & \text{otherwise} \end{cases}$$

Our goal is then to find an appropriate assignment to each A_{ij} that minimizes the sum of normalized latency, migration cost and maximizes latency reduction while satisfying the constraints. Intuitively, the new VD placement should incur low migration cost and have large latency reduction. Similarly, we use $\bar{A}$ to represent the current placement of VMs among J hosts. More specifically, let us denote the current placement of i^{th} VM as p_i, i.e., it is running in $Host_{p_i}$, we then have $\bar{A}_{ip_i} = 1$ (and all other element in vector $\bar{A}_i$ as 0) for i^{th} VM. We formulate the ILP problem as following:

$$\min \quad \sum_{i,j} \frac{A_{ij}L_{ij}}{L_i} + \sum_{i,j} \mathbb{1}\{A_{ij} = \bar{A}_{ip_i}, j \neq p_i\}(\frac{C_i}{C} - \frac{B_{ij}}{B}) \tag{3.7}$$

subject to:

$$\sum_{i=1}^{I} A_{ij}u_i \leq U_j, \quad \forall j = 1 \ldots J \tag{3.8}$$

$$\sum_{i=1}^{I} A_{ij}m_i \leq M_j, \quad \forall j = 1 \ldots J \tag{3.9}$$

$$\sum_{i=1}^{I} A_{ij}d_i \leq D_j, \quad \forall j = 1 \ldots J \tag{3.10}$$

$$\sum_{i=1}^{I} A_{ij}n_i \leq N_j, \quad \forall j = 1 \ldots J \tag{3.11}$$

$$\sum_{j=1}^{L} A_{ij} = 1, \quad \forall i = 1 \ldots I \tag{3.12}$$

$$A_{ij} \in \{0,1\}, \quad \forall i = 1 \ldots I, j = 1 \ldots J \tag{3.13}$$

Where L_i is the maximum latency of placing the i^{th} VM among all J hosts, i.e., $L_i = \max\langle L_{i1}, \ldots L_{ij}\rangle$. Specifically, L_{ij} denotes the expected network latency between a thin client that connects to i^{th} VDs and j^{th} host. C_i and B_{ij} denote the cost and benefit of migrating the i^{th} VM from its current host to a new one. We consider the cost of migrating the i^{th} VM to be the amount of data to be moved and the benefit of migrating to host j be the latency reduction. Further, we use C and B to denote the maximum cost and benefit of migrating all I VMs. That is, $C = \max\langle C_1, C_2 \ldots C_I\rangle$ and $B = \max\{B_{ij} | \forall i = 1 \ldots I, \forall j = 1 \ldots J\}$. Our objective function (3.7) not only considers the normalized latency associated with new placement decision, but also uses indicator function $\mathbb{1}$ to capture the relation between new placement decision and current virtual desktops to hosts mapping. We normalize each term to balance the impacts of metric on determining the placement decision. Constraints (3.8) to (3.11) ensure the placement decision of VMs satisfy the physical resource constraints of the hosts while constraints (3.12) to (3.13) together ensure each VM will only be placed in one host at every time point. Our ILP can require long time to compute placement decisions for large problem sizes. In the next section, we propose three different greedy heuristics that efficiently compute with new placement decisions.

3.5.2 VMShadow Greedy Heuristics

Rank-ordered greedy. In this approach, we consider all desktop VMs whose latency-sensitive ranks exceed a threshold S_{up} and consider them for relocation in rank order. Thus the highest ranked desktop VM is considered first for optimization. If the closest regional cloud site to this VM has insufficient resources, the greedy heuristic attempts to free up resources by evicting VMs that have been flagged for reclamation. If no VMs can be reclaimed or freed-up resources are insufficient to house the candidate VM, the greedy approach then considers the next closest cloud site as a possible home for the VM. This process continues until a new location is chosen (or it decides that the present location is still the best choice). The greedy heuristic then considers the next highest ranked desktop VM and so on. While rank-ordered greedy always moves the most needy (latency-sensitive) VM first, it is agnostic about the benefits of these potential moves—it will move a highly ranked VM from one data center location to another even if the VM is relatively well-placed and the move yields a small, insignificant performance improvement.

Cost-oblivious greedy. An alternate greedy approach is to consider candidates in the order of relative benefit rather than rank. This approach considers all VMs that are ranked above a threshold S_{up} and orders them by the relative benefit B of a move. We define the benefit metric as the weighted sum of the absolute decrease in latency and the percentage decrease. If l_1 and l_2 denote the latency from the current and the new (closest) data center to the end-user, respectively, then benefit B is computed as:

$$B = w_1 \cdot (l_1 - l_2) + w_2 \cdot \frac{(l_1 - l_2) * 100}{l_1} \tag{3.14}$$

where w_1 and w_2 are weights, $l_1 - l_2$ denotes the absolute latency decrease seen by the VM due to a move and the second term is the percentage latency decrease. We do not consider the percentage decrease alone, since that may result in moving VMs with very small existing latencies. For example, one VM may see a decrease from 100ms to 60ms, yielding a 40% reduction, while another may see a decrease from 2ms to 1ms, yielding a 50% reduction. Although the latter VM sees a greater percentage reduction, its actual performance improvement as perceived by the user will be small. Consequently the benefit metric considers both the percentage reduction and the absolute decrease. The weights w_1 and w_2 control the contribution of each part—we currently use $w_1 = 0.6$ and $w_2 = 0.4$ to favor the absolute latency decrease since it has more direct impact on improving performance.

Once candidate VMs are ordered by their benefit, the cost-oblivious greedy heuristic considers the VM with the highest benefit first and considers moving it using a process similar to rank-ordered greedy approach. The one difference is that if the VM cannot be relocated to the best location, this approach recomputes the benefit metric to the next best site and re-inserts the VM into the list of VMs in benefit order, and picks the VM with most benefit. Ties are broken by rank (if two candidates have the same benefit metric, the greedy considers the higher ranked VM first).

Cost-aware greedy. Cost-oblivious greedy only considers the benefit of potential moves but ignores the cost of such migrations. Since the disk and memory state of VMs will need to be migrated over a WAN, and this may involve copying large amounts (maybe gigabytes) of data, the costs can be substantial. Consequently, the final variant of greedy, known as cost-aware greedy heuristic, also considers the cost of moving a VM as:

$$C = (S_{disk} + S_{mem}) \cdot \frac{1}{1-r} \tag{3.15}$$

where S_{disk} and S_{mem} denote the sizes of the disk and memory state of the virtual machine and parameter r captures the dirtying rate of the VM relative to the network bandwidth.[2] The dirty rate r could be either estimated by the network traffic to VD or monitored from hypervisor as the disk I/O write rates.

The cost-aware greedy approach then orders all candidate VMs using $\frac{B}{C}$ (i.e., the benefit weighted by the cost). A candidate with a higher $\frac{B}{C}$ offers a higher performance improvement benefit at a potentially lower migration cost. The VM with the highest $\frac{B}{C}$ is considered first for possible movement to the closest cloud site. Like before, if this site has insufficient server resources, then VMs marked for reclamation are considered for eviction from this site to make room for the incoming VM. Note, Equation (3.15) implicitly consider the potential cost of reclamation as one has to at least free up C amount of disk and memory spaces by evicting VMs. If no such reclamation candidates are available, the VM is considered for movement to the next closest site. The benefit metric to this next site is recomputed and so is the $\frac{B}{C}$ metric and the VM is reinserted in the list of candidate VM as per its new $\frac{B}{C}$ metric. The greedy heuristic then moves on to the next VM in this ordered list and repeats. Ties are broken using the VMs' rank.

Our VMShadow prototype employs this cost-aware greedy heuristic. It is straightforward to make the cost-aware greedy implementation to behave like the cost-oblivious or the rank-ordered greedy variants by setting the cost (for cost-oblivious) and benefit (for rank-ordered greedy) computation procedures to return unit values.

Avoiding Oscillations: To avoid frequent moves or oscillatory behavior, we add "hysteresis" to the greedy algorithm — once a candidate VM has been moved to a new location, it is not considered for further optimization for a certain hysteresis duration T. Similarly,

[2]Live migration of a VM takes place in rounds, where the whole disk and memory state is migrated in the first round. Since the VM is executing in this period, it dirties a fraction of the disk and memory, and in the next round, we must move $(S_{disk} + S_{mem}) \cdot r$, where r is the dirtied fraction. The next round will need an additional $(S_{disk} + S_{mem}) \cdot r^2$. Thus we obtain an expression: $(S_{disk} + S_{mem}) \cdot (1 + r + r^2 + \ldots)$. This expression can be further refined by using different disk and memory dirtying rates for the VM.

any VM which drops in its latency-sensitivity rank is not evicted from a local site unless it exhibits consistently low rank for a hysteresis duration T'. Moreover, the cost-benefit metrics avoid moving VMs that see small performance improvements or those that have a very high data copying cost during migration.

3.6 Transparent VM and Connection Migration

While VMShadow attempts to optimize the performance of latency-sensitive VMs by moving them closer to their users, it is critical that such moves be transparent. The desktop VM should not incur downtime when being moved from one cloud site to another or encounter disruptions due to a change of the VM's network address. VMShadow uses two key mechanisms to achieve this transparency: live migration of desktop virtual machines over the WAN, and transparent migration of existing network connection to the VM's new network (IP) address. We describe both mechanisms in this section.

3.6.1 Live Migration Over WAN

When VMShadow decides to move a VD from one cloud site to another, it triggers live migration of the VM over the WAN. While most virtualization platforms support live VM migration within a data center's LAN [39], there is limited support, if any, for a migration over the wide area. Hence, we build on the WAN-based VM migration approach that we proposed previously [167], but with suitable modifications for VMShadow's needs.

The WAN-based VM migration that we use in VMShadow requires changes to the hypervisor to support efficient WAN migration. It is possible to implement these modifications of the hypervisor in private clouds where an enterprise has control over the hypervisor. Similar modifications are also possible in public clouds where the cloud provider itself offers a desktop cloud service to users. However, the desktop cloud service may also be implemented by a third-party that leases servers and storage from a public IaaS cloud provider (e.g., if Onlive's Desktop service were implemented on top of Amazon's EC2). In such scenarios, the third party should not expect modifications to the hypervisor.

To support such scenarios also, we employ a nested hypervisor to implement VMShadow's migration techniques. A nested hypervisor runs a hypervisor h' inside a normal virtual machine that itself runs on a typical hypervisor h; actual user VMs run on top of hypervisor h'. Since the nested hypervisor is fully controlled by the desktop cloud provider (without requiring control of the underlying hypervisor), it enables hypervisor-level optimizations. Note that using a nested hypervisor trades flexibility for performance due to the additional overhead of running a second hypervisor; however, Xen-Blanket [165], which we use in our prototype has shown that this overhead is minimal. As a result, VMShadow can run over unmodified public cloud instances, such as Amazon EC2, and live migrate desktop VMs from one data center to another. In addition, VMShadow's WAN migration needs to transfer both the disk and memory state of the desktop virtual machine (unlike LAN-based live migration which only moves the memory state since disks are assumed to be shared). VMShadow uses a four step migration algorithm, summarized in Fig. 3.5.

Step 1: VMShadow uses Linux's DRBD module to create an empty disk replica at the target data center location. It then begins to asynchronously transfer the disk state of the VM from the source data center to the target data center using DRBD's asynchronous replication mode. The rate of data transfer can be controlled, if needed, using Linux' traffic control (*tc*) mechanisms to avoid any performance degradation for the user during this phase. The application and VM continue to execute during this period and any writes to data that has already been sent must be re-sent.

Step 2: Once of the disk state has been copied to the target data center, VMShadow switches the two disk replicas to DRBD's synchronous replication mode. From this point, both disk replicas remain in lock step—any disk writes are broadcast to both and must finish at both replicas before the write returns from the disk driver. Note that disk writes will incur a performance degradation at this point since synchronous replication to a remote WAN site increases disk write latency.

Step 3: Concurrent with Step 2, VMShadow also begins transferring the memory state of the VM from the source location to the target location. Like LAN-based live migration approaches, VMShadow uses a pre-copy approach which transfers memory pages in rounds [39]. The first round sequentially transfers each memory page from the source to the destination. As with the disk, VMShadow can control the rate of data transfer to mitigate any performance impact on front-end user tasks. Since the application is running, it continues to modify pages during this phase. Hence, each subsequent round transfers the only pages that have been modified since the previous round. Once the number of pages to transfer falls below a threshold, the VM is paused for a brief period and the remaining pages are transferred, after which the VM resumes execution at the destination.

Since substantial amounts of disk and memory data need to be transferred over the WANs, VMShadow borrows two optimizations from our prior work [167] to speed up such transfers. First, block and page deltas [46] are used to transfer only the portion of the disk block or memory page that was modified since it was previously sent. Second, caches are employed at both ends to implement content based redundancy(CBR) [8, 167]—duplicate blocks or pages that have been sent once need not be resent; instead a pointer to the cached data is sent and the data is picked up from the receiver cache. Both optimizations have been shown to reduce the amount of data sent over the WAN by 50% [167].

Step 4: Once the desktop VM moves to a new data center, it typically acquires a new IP address using DHCP. Changing the IP address of the network interface will cause all existing network connections to break and disrupt user activity. To eliminate such disruptions, VMShadow employs a connection migration protocol to "migrate" all current TCP connections transparently to the new IP address without any disruptions (TCP connections see a short pause during this transfer phase but resume normal activity once the migration completes). The connection migration is triggered after desktop VM is successfully migrated

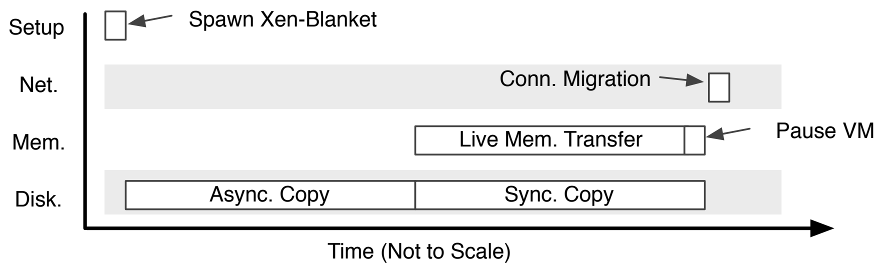


Figure 3.5: **VMShadow migration phase using Xen-Blanket.** Upon WAN live migration, a Xen-Blanket (nested hypervisor) VM is spawned first to receive disk and memory state from source WAN live migrator. It is then followed by a live memory and disk transfer before briefly pausing the VM. The VM is successfully migrated to the new Xen-Blanket and ready to use after executing connection migration protocol.

and then paused. Immediately afterwards, VMShadow updates the new mapping rules at proxies. Once the rules are updated, the migrated VM will be resumed with the new public IP address, and all subsequent packets will be rewritten. In summary, the actual traffic switching occurs after the connection migration protocol is successful. Once both the VM and connection migration phases complete, the desktop VM begins executing normally at the new cloud location. We describe VMShadow's connection migration protocol next.

3.6.2 Connection Migration Protocol

Different cloud locations are typically assigned different blocks of IP addresses for efficient routing. As a result, when a VM moves from one cloud location to another, it is typically assigned an IP address from the new location's IP block and will not retain its original IP address. This will cause TCP connections to be dropped and result in disruptions to end users' sessions. To prevent such disruptions, VMShadow employs a connection migration protocol that "migrates" these connections to the new IP address.

The issue of mobility, and having to change the IP address as a result, is a well known problem. There have been several proposals including HIP [2], LISP [4], ILNP [3] and Serval [126] that try to address this problem by separating the host identifier from the network address. With these approaches, the application connects at the TCP layer using the host identifier, while the packets are routed using the network address. When the user (i.e., host) moves, the network address changes, but the host identifier stays the same. As a result, TCP connections are not disrupted. Unfortunately, all these approaches require modifications to the application to take advantage of seamless mobility.

Instead, here we take a more pragmatic approach so that VMShadow works seamlessly with existing applications as they are. VMShadow makes use of a local proxy to implement a network connection migration protocol. VMShadow assumes that both end-points for every active connection on the migrated VM run this proxy (thus, both the thin client and the desktop VM need to run the proxy, as do other servers elsewhere with active TCP connections to the desktop VM). However, in the cases where we don't have control over servers, for example YouTube streaming servers, we can set up in-network proxy servers that are closer to VDs. We envision the virtual desktop cloud providers will be in charge of maintaining these proxy servers. In summary, as long as the proxy is in the data path for the TCP connection between end points, it can masks any address changes by dynamically re-writing the IP headers of the packets.

To ensure transparency, the desktop VM uses two logical network interfaces: an *internal* interface with a fixed, private IP address and an *external* interface with the "real", but potentially changing, IP address. All socket connections are bound to the internal interface as the local source address; as a result, active socket connections never directly see the changes to the external IP address. The proxy acts as a bridge between the internal and external network interfaces for all packets as shown in Fig. 3.6. Internally generated packets have a destination address that is the external IP address of the remote end host.

The proxy employs dynamic rewriting of packet headers (analogous to what is done in NAT devices) to bridge the two interfaces. For all outgoing packets, the default rewriting rule replaces the source IP of the internal interface with that of the external interface: $(IP_{int}, *) \rightarrow (IP_{ext}, *)$. Thus when the external IP address changes after a WAN migration, the rewrite rule causes any subsequent packet to have the new external IP address rather

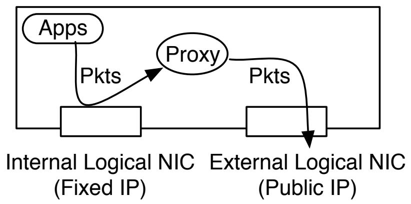


Figure 3.6: **Illustration of proxy IP bridging.** Inside each VM, the proxy bridges an internal logical NIC with the external one, masking the potential IP address changes from the higher-level applications.

than the old one. Incoming packets headers are rewritten with the reverse rule, where the current external IP address is replaced with the fixed internal IP.

After an IP address change of a desktop VM, other end-points with connections to the desktop VM will begin seeing packets arriving from the new external IP address. However connections on these machines expect packets from the old external IP address of the desktop VM. To ensure transparent operation, the local proxies in other end-points intercept packets with the desktop VM and apply new rewrite rules beside the *default* one. For example, with new rewrite rules, incoming packets arriving from the desktop VM are rewritten as $(IP_{new}, *) \rightarrow (IP_{old}, *)$ while outgoing packets to the desktop VM see rewrites to the destination IP address as $(*, IP_{old}) \rightarrow (*, IP_{new})$. These two rules ensures that outgoing packets go to the new address of the desktop VM (and thus are not lost), while incoming packets from the new IP address are rewritten with the old address before delivery to applications (that are still given the illusion of communicating with the old IP address). We illustrate various scenarios in Fig. 3.7.

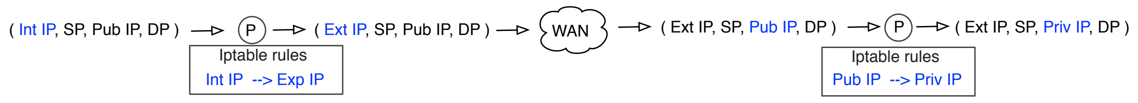
To achieve this transparent migration, the proxies at both end points use control messages to signal each other about the change in IP address. This is done by having the desktop VM send a cryptographically signed message to the corresponding proxy informing it of the IP address change. The cryptographic signing avoids malicious third-parties from sending bogus IP address change messages and causing a denial of service. A typical

IP address change control message will include the old IP address and request subsequent packets to be sent to the new address.

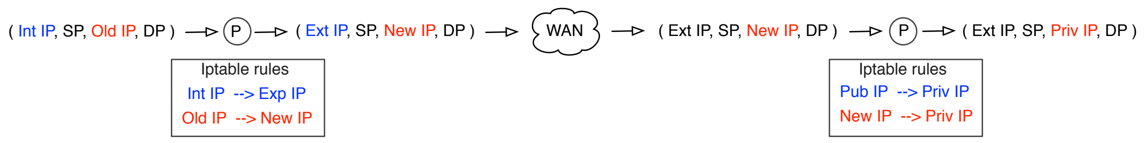
Note that the connection migration protocol is symmetric—it assumes an fixed internal interface and an external interface on all machines. Thus, the protocol can also handle IP address changes of the thin client or other machines that the desktop VM communicates with. Further, the extra rewrite rules are generated on a per-socket basis rather than a per-machine basis to support dynamic connection setup. In particular, connections established before the IP address change requires rewriting based on both default and extra rules to maintain connectivity. Connections opened after the address change talk to the new address and only need default packet rewriting. However, for incoming packets, we use the port information of the connections to distinguish between ones that need a re-write (connections opened prior to the change) versus those that do not (those opened after the change). A general rewrite rule of an outgoing packet is of the form: $(IP_{int}, srcPort, IP_{old}, dstPort) \rightarrow (IP_{ext}, srcPort, IP_{new}, dstPort)$.

3.6.2.1 Handling NAT Devices

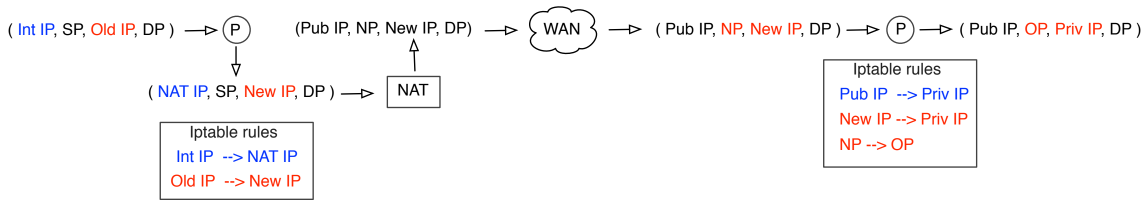
Our discussion thus far assumes that all end points have publicly-routable IP addresses. However in many scenarios, one or both end-points may be behind NAT devices. We first consider the scenario where the thin client is behind a NAT (e.g., in a home) while the desktop VM resides in a public cloud and has a public IP address. In this case, when the desktop VM is moved from one location to another, it will no longer be able to communicate with the thin client since the NAT will drop all packets from the new IP address of the desktop. In fact, the desktop VM will not even be able to notify the proxy on the thin client of its new IP address (since a "strict NAT" device drops all packets from any IP address it has not encountered thus far). To address this issue, we resort to NAT hole punching [59], a method that opens ports on the NAT to enable the desktop VM to communicate with the thin client.


(Int IP, SP, Pub IP, DP)
P
(Ext IP, SP, Pub IP, DP)
WAN
(Ext IP, SP, Pub IP, DP)
P
(Ext IP, SP, Priv IP, DP)
Iptable rules
Int IP --> Exp IP
Iptable rules
Pub IP --> Priv IP

(a) Both entities have public IP addresses.


(Int IP, SP, Old IP, DP)
P
(Ext IP, SP, New IP, DP)
WAN
(Ext IP, SP, New IP, DP)
P
(Ext IP, SP, Priv IP, DP)
Iptable rules
Int IP --> Exp IP
Old IP --> New IP
Iptable rules
Pub IP --> Priv IP
New IP --> Priv IP

(b) One entity's public IP changes.



(c) A behind-NAT end point tries to communicate to an entity with new public IP address.

Figure 3.7: **Dynamic rule-based packet headers' rewriting sequences.** We use four-tuple, i.e. the source IP, source port, destination IP and destination port, to represent a packet. Packets are matched based on the iptable rules of the proxy (Default rule is in rule while new rules are in red). This ensures that high-level applications only see fixed internal IP without breaking the TCP connections.

VMShadow's NAT hole punching is part of the connection migration process. It works by notifying the client proxy of the IP address change from the old IP address of the desktop VM. In some scenarios, the desktop VM may be able to determine its new IP address at the destination *before it migrates*. This may be possible in enterprise private clouds where an IP address is pre-allocated to the VM, or even in public clouds where one can request allocation of an elastic IP address independent of VM instances. In such cases, the proxy on the desktop VM notifies the proxy on the thin client of its future IP address and requests hole punching for this new IP address. In scenarios where the IP address cannot be determined *a priori*, we assume that the newly migrated VM will notify the driver domain of the nested hypervisor at the old location of its new address. The driver domain can use the old IP address to notify the proxy at the thin client of the IP address change and consequently request hole punching.

Once the new IP address has been communicated to the client proxy, it proceeds to punch holes for each active socket port with the desktop VM. This is achieved by sending a specially marked packet from each active source port to each active destination port but with the new IP address as the destination IP of these specially marked packets. These packets causes the NAT device to open up these ports for accepting packets from the new IP address of the desktop VM. NAT devices typically rewrite the source port number with a specially allocated port number and create a forwarding rule; packets arriving on this NAT port are forwarded to the source port at the thin client device. Thus, a regular outgoing packet from the client to the desktop VM will see the following rewrites: the source proxy peforms the first rewrite $(IP_{int}, srcPort, IP_{old}, dstPort) \rightarrow (IP_{NAT}, srcPort, IP_{new}, dstPort)$. The NAT device then further rewrites this packet as $(IP_{NAT_{Ext}}, natPort\ , IP_{new}, destPort)$.

When the first specially marked packet of this form is received at the desktop VM, it creates a mapping of the *old* natPort of the source to the *new* natPort. Then port numbers of any outgoing packets are rewritten by replacing the *old* natPort with the *new* natPort created by the hole punching. Note that the specially marked hole punching packet is only processed by the proxy and then dropped and never delivered to the application. In our implementation, we simply assign a TCP sequence number of 1 and have an iptables rule for dropping potential RST packet. This extension enables the connection migration protocol to work even when one of the end-points is behind a NAT device. The protocol can be similarly extended with hole punching packets in both directions when both end-points are behind NAT devices. Note in this scenario, the entity that moved from one NAT to another will need to find out the IP address of the new NAT device first before proceeding hole punching. We omit the details here due to space constraints.

3.7 VMShadow Implementation

We have implemented a prototype of VMShadow using Linux 3.1.2 and modified Xen-Blanket 4.1.1 [165]. Our prototype is written in C and Python and consists of several interacting components as shown in Figure 3.3. In the following, we describe design tradeoffs, functionalities and implementation details of each component.

Fingerprinting Engine. Our fingerprinting engine includes a distributed traffic collector in each host and a central fingerprinting engine running inside the cloud manager. Its main tasks include collecting network-level traffic information from each host and calculating the latency-sensitive score for each virtual desktop. We implement the traffic collector component in Xen-Blanket's driver domain (dom0). It uses python interfaces to the Linux $netfilter$ library, more specifically $libnetfilter\_queue$ to copy packets queued by the kernel packet filter into user-space for analysis; it periodically samples the traffic and sends the statistics to the fingerprinting engine running inside the cloud manager. The hypervisor-based fingerprinting system has negligible overhead, and does not interfere with a virtual desktops' normal performance. Specifically, the overhead can be broken down into copying packets, generating and sending statistics to the fingerprinting engine. The dominant overhead comes from copying every network packet, but can be dramatically reduced by mapping kernel buffers to user space. This allows sharing buffers between kernel and user space applications, and essentially achieves zero-copy overhead. The caveat is the kernel needs to support zero-copy optimizations. For each active virtual desktop, the cloud manager then analyzes the normalized network traffic, normalized protocol traffic and percentage of normal internet traffics (as described in Section 3.4) based on the collected network traffics and the maintained list of latency-sensitive ports and server addresses. A relative latency-sensitive scores is assigned to each virtual desktop at the end of fingerprinting process.

WAN Live Migrator. Our WAN live migrator takes any running virtual desktop and migrates it to a different host as fast as possible without disrupting its functionalities. We implement the migrator on top of the nested hypervisor, i.e., the Xen related code in Xen-Blanket, by modifying live migration code in Xen. More specifically (refer to Figure 3.5 for a pictorial detail), we include DRBD-based disk state migration to concurrently transfer virtual machine disks asynchronously. For transferring memory, we employing multiple optimizations, i.e., zero page, memory page deltas and content-based redundancy elimination [167] to optimize the transferring over WAN. To mitigate the live migration impact's on the client traffic, we also implement the rate control mechanisms to control the rate of state transfer over WAN links.

Connection Proxy. Our connection proxy implements our connection migration protocol discussed in Section 3.6.2 as a python process. We design and implement the proxy in a way that make it easy and flexible to run on any end point such as a VD and thin client. The proxy listens on a well-known port, to receive (and send) cryptographically signed messages for announcing IP address changes. It uses the `libnetfilter_queue` library to intercept outgoing and incoming packets and rewrites the corresponding TCP headers as specified by the current rewrite rules in `iptable`. Packets are reinserted into the queue once the headers have been rewritten. We use the python `scapy` library to generate the appropriate packets for NAT hole punching. Our choice of implementing the proxy in user-space is based on the trade-off of the implementation ease and overhead compared to a kernel implementation. In production use where efficiency has higher priority, one should implement the protocol in the kernel space to reduce the data copy overhead as well as `iptable` rules matching. We evaluate the overhead of our user-space proxy in Section 3.8.4.

Cloud Manager. We use a centralized-architecture in implementing the cloud manager that runs periodically. It has interfaces to both fingerprinting engines and WAN live migrators that run distributed on each host. After each time period, our cloud manager feeds the latency-sensitive scores calculated by fingerprinting engine to the algorithm engine, to figure out the new virtual desktop placement. Our algorithm engine implements both the ILP and cost-aware greedy algorithm. The migration manager then compares the new placement plan with the current placement to figure out a migration table that has three columns, i.e., the source host, the destination host and the target virtual desktop. Each row

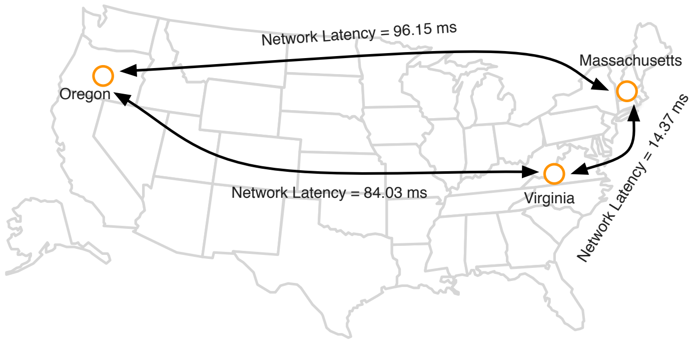

Network Latency = 96.15 ms
Massachusetts
Oregon
Network Latency = 14.37 ms
Network Latency = 84.03 ms
Virginia

Figure 3.8: **Illustration of cloud sites setup in our experiments.** Three cloud sides used for our experiments: a private cloud side in Massachusetts, and EC2 sites in Virginia and Oregon.

in the table represents a migration that needs to be actuated to improve the user-perceived performance. Our migration manager executes the specified migrations by contacting the WAN live migrator as well as the connection proxy on each source host. To avoid unnecessary performance degradation, our cloud manager employs two intuitive methods that both aims at reducing the percentage of live migration bandwidth usage. The first one is to limit the number of concurrent live migrations between the same hosts and the second one is to control the total bandwidth usage of live migration.

3.8 Experimental Evaluation

In this section, we first describe our experimental setups and then present our experimental results. In designing our experiments, we are interested in answering the following key questions.

1. How accurate is our black-box fingerprinting algorithm in distinguishing latency-sensitive desktop VMs from the rest?

2. What is our proposed cost-aware greedy algorithm compared to ILP in optimizing the location of desktop VMs?

3. What are the potential overhead of using live migration to move desktop VMs from one cloud location to another and the performance benefits to desktop VMs' users?

4. How efficient is our connection migration proxy in seamlessly transferring the TCP connections?

5. Last, how does our prototype VMShadow work in resolving complex scenarios by detecting latency-sensitive desktop VMs and improving their performances within resource constrains ?

Experimental Setup: The testbed for our evaluation consists of hybrid clouds with a private cloud in Massachusetts and Amazon EC2 public clouds across different locations as shown in Fig 3.8. The private cloud consists of 2.4GHz quad-core Dell servers running Centos 6.2 and GNU/Linux kernel 2.6.32. On Amazon EC2, we use extra-large instances (m3.xlarge), each with 4 VCPUs, at two sites: US-West in Oregon and US-East in Virginia. All machines run modified Xen-Blanket 4.1.1 and Linux 3.1.2 as Dom0.

Our desktop cloud consists of Ubuntu 12.04 LTS desktops that are installed with vnc4server as the VNC server. Each desktop VM will have only one desktop session and accept connections from one thin client. We modify an open-source version of VNC viewer [160] and use it to allow laptop-based thin client machines to connect to VNC servers. For mobile phone, i.e. iPhone, thin client, we use VNC viewer acquired from App Store and manually perform user activities. Users that connect to the Ubuntu desktop will be able to run a variety of desktop applications, including OpenOffice for editing documents, Google Docs for online editing, Chrome browser [3] for web-browsing and watching various online streaming, i.e. Youtube, Hulu and Netflix, Thunderbird email client and Movie Player for local video playback. Each desktop VM is assigned 1GB memory, 1 VCPU and has a 8 GB disk of which 1.32GB is used and runs inside Xen-Blanket dom0.

[3]We choose Chrome browser due to the fact that Netflix is not supported in the default Firefox browser.

	Uplink traffic				Downlink traffic			
	Youtube	Local Video	Browsing	Text Edit	Youtube	Local Video	Browsing	Text Edit
Non-VNC Traffic (%)	37.7	0	0.67	0	53.7	0	0.62	0
Non-VNC Bandwidth (KB/s)	1.85	0	0.0083	0	63.6	0	0.0059	0
Total Bandwidth (KB/s)	74.6	54.5	17.94	17.14	65.8	1.54	0.454	0.86

Table 3.2: **Characterization of Desktop VMs' network activities.** Virtual desktops running different applications exhibit different network characteristics, i.e., remote protocol traffic and Internet traffic.

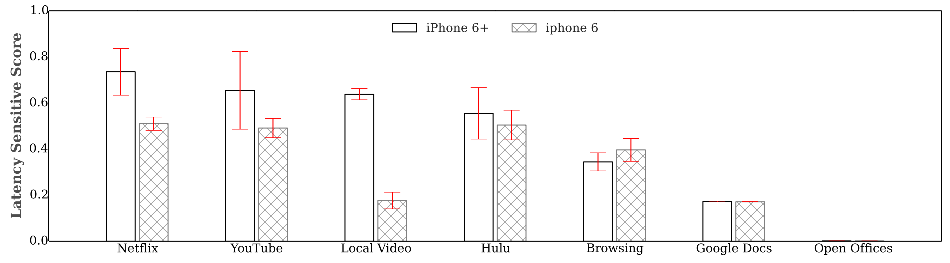


Figure 3.9: **Comparison of latency sensitive scores for different desktop VMs.** Online streaming applications have higher scores compared to the other types of applications for both mobile clients. For both clients, all but one latency score match our hypothesis. But this "out-of-order" ranking can be remedied using threshold scores as discussed in Section 5.5.

3.8.1 Accuracy of Black-box VM Fingerprinting

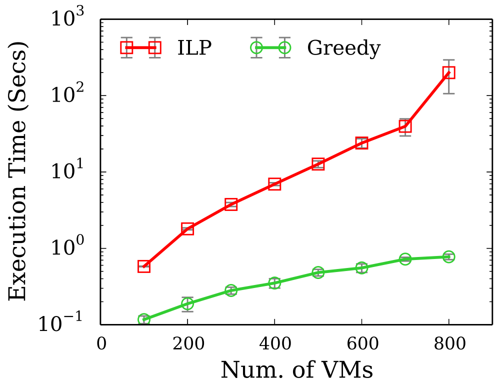
Black-box VM fingerprinting provides us latency-sensitive scores for each desktop VMs without peeking inside the user actives. In this experiment, we first show that VDs with different latency-sensitive requirement exhibit vastly different network-level characteristics and then we demonstrate that our approach is able to assign correct relative latency-sensitivity scores to different VMs running various applications. We use Wireshark running on Dom0 of Xen-Blanket to collect packet-level traces for each VMs during the experiment periods.

Characterizing Network Activities of Desktop VMs. We use VNC viewer from our laptop-based thin client to perform four distinct types of user actives, i.e., watching Youtube video, browsing graphic-rich websites, text editing using OpenOffices and watching video locally on the desktop VM. In each case, we sample the traffic generated by the VM in a 3-minutes measurement window after a warmup period and then repeat the process five times. We compute the average statistics across all five-runs and use them to characterize the network activities of each VMs.

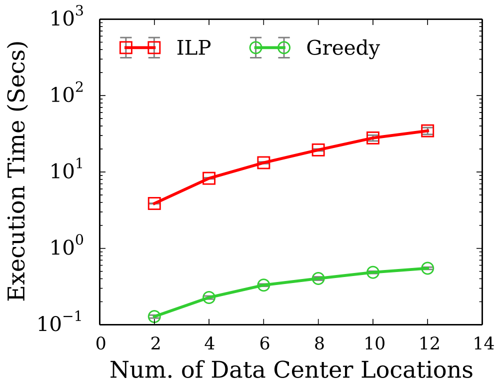
Table 3.2 summarizes the different network activities of desktop VMs for these four activities. As expected, YouTube viewing consumes higher network bandwidth both from YouTube servers and for the remote desktop protocol display; video playback from a local file does not consume network bandwidth, but the data transfer for VNC is still high due to the video playback. Web browsing and text editing consume very little bandwidth. Note in our fingerprinting algorithm, both YouTube and graphic-rich browsing will be labeled as latency-sensitive traffic based on the server ports and addresses.

Assigning Latency-sensitive Scores to Desktop VMs. Based on the above observations, we next evaluate our fingerprinting algorithm that favors and assign relative high scores to VMs based on their latency sensitivities. We use two models of iPhones, i.e., iPhone 6 and iPhone 6+, as our mobile thin clients and collect necessary network-level data using the same setup as in previous experiment. Our measurement data, together with our list of latency sensitive ports and server addresses, are provided as input to our fingerprinting algorithm in Section 3.4 to calculate the scores.

Figure 3.9 compares the different latency scores assigned for desktop VMs that are running various applications. In general, our fingerprinting algorithm is able to assign “correct” relative scores to VMs running different applications. Specifically, both online streaming applications, disregarding the service providers, and local video playback are assigned with high latency scores. However, the score of watching local video using iPhone 6 is much lower than its counterparts. Recall that the relative latency score is calculated based on normalized throughput, protocol throughput and latency-sensitive throughput. Because local video does not generate internet traffic, it has a lower relative score compared to online streaming. In addition, with adaptive bitrate streaming, the amount of data transferred depends on screen size. This means an iPhone 6 with a smaller screen will have lower



(a) The number of cloud locations is set to 40.



(b) The number of VMs is set to 2000.

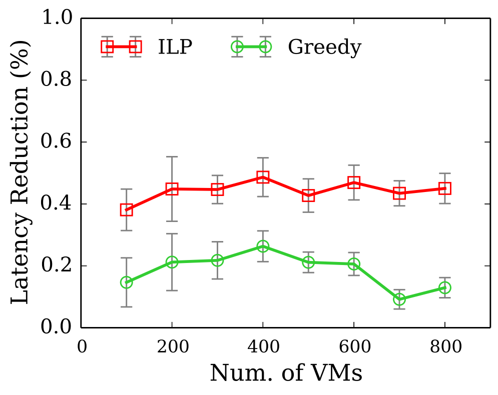
Figure 3.10: **Execution time comparisons between ILP and greedy algorithms.** In general, greedy algorithm takes significant less time compared to ILP algorithm. For both algorithms, as the number of VMs to be assigned or the number of candidate cloud locations to be picked increase, the running time increases accordingly.

relative score than an iPhone 6+. Since it heavily relies on the application bandwidths demand in calculating the scores, the results will be biased for thin clients with different screen size. To further improve the accuracy of latency scores, we could apply the algorithm based on the screen size of thin clients. Graphic-rich browsing, such as “imgur.com”, is considered more latency-sensitive compared to online editing using Google docs. Last, local editing, with a score of 0.001, is regarded to be not sensitive to latency at all and is potential candidate for resource reclamation.

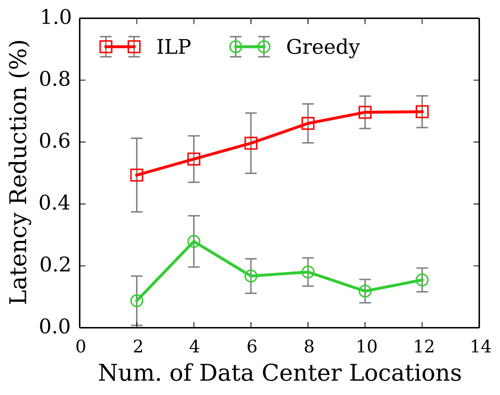
*R*esult: Desktop VMs that run different applications exhibit different level of network activities. Based on this observation, our fingerprinting algorithm is able to correctly favor and distinguish latency-sensitive desktop VMs, i.e. the ones that run online streaming or local video playback, from non-sensitive desktop VMs, i.e, local text editing.

3.8.2 Comparing Greedy Shadow Algorithm to ILP

In this experiment, we study the performance differences between our cost-aware greedy algorithm and integer linear program (ILP) algorithm that is able to provide optimal results but with longer execution time. Both algorithms are implemented in VMShadow’s Cloud



(a) The number of cloud locations is set to 40.



(b) The number of VMs is set to 2000.

Figure 3.11: **Comparison of latency reduction percentage between ILP and greedy algorithms.** When the number of VMs to be assigned increase, the reduction is bounded by the collections of data center locations. As the number of data center locations increases, ILP is able to utilize the data center locations to find optimal solutions for each VMs.

Manager. Specifically, we implemented the ILP algorithm using Python's Convex optimization package CVXOPT that aims to minimize the latency reductions. We compare the greedy algorithm with the ILP approach in terms of scalability, i.e., execution time, and effectiveness, i.e., latency decrease percentage of desktop VMs.

To stress test both algorithms, we create synthetic scenarios with increasing numbers of desktop VMs and cloud locations and measure the execution time and effectiveness of both algorithms. In one case, we fix the number of desktop VMs to 2000 and vary the number of available cloud locations from 2 to 12. In another case, we fix the number of cloud locations to 40 and vary the number of desktop VMs in the cloud from 100 to 800. For each scenario, we run both version of algorithms ten times by assigning uniformly generated latency-sensitive scores to each VM and uniformly pick a set of data centers from a pool of forty locations. We average the results across all runs to represent the performance and effectiveness for each scenario.

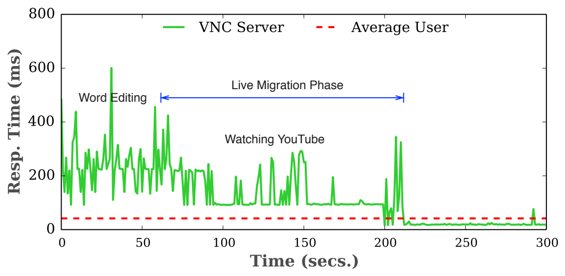
Figure 3.10 compares the execution time of these two algorithms in these two cases separately. As expected, the execution time of the ILP approach increases significantly

	Word+YouTube	Word
Mem (GB)	0.56	0.54
Disk (GB)	1.36	1.34
Total Time(s)	265	249
Pause Time(s)	2.48	2.8

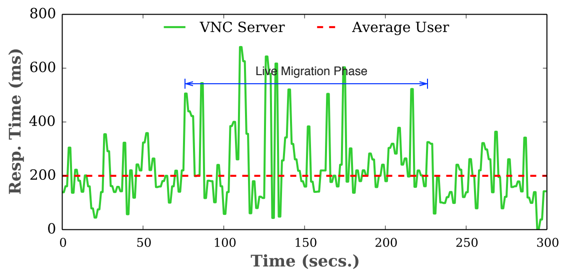
Table 3.3: **Comparison of transcontinental WAN migration of desktop VMs.** Desktop VMs are migrated from Amazon EC2's Oregon data center to Virginia data center using VMShadow that is optimized with delta-based and CBR techniques. We observe a slightly more memory and disk data transfer for desktop VM that runs more applications. The migration pause time is due to the last iteration of memory transferring and TCP connection migration.

with both increasing location choices(as in Figure 3.10b) and increasing VMs(as in Figure 3.10a); the execution time of the greedy approach, in comparison, remains flat for both scenarios. Figure 3.11 evaluates the effectiveness of the two algorithms in reducing the latency of desktop VMs via migrations. Latency reduction achieved by our greedy approach is within 51-56% of the "optimal" ILP approach when our greedy algorithm has access to all forty data center locations. In the case of assigning all 2000 VMs to cloud locations, the effectiveness of our greedy approach is impacted either by the limited amount of cloud locations, (in the case of only 2 cloud locations), or the complexity growths. In general, the ILP approach is a better choice for smaller settings (where it remains tractable), while greedy is the only feasible choice for larger settings. Note also that our experiments stress test the algorithms by presenting a very large number of migration candidates in each run. In practice, the number of candidate VMs for migration is likely to be a small fraction of the total desktop VMs at any given time; consequently the greedy approach will better match the choices made by the ILP in these cases.

Results: VMShadow's greedy algorithm is able to achieve around 51-56% effectiveness with marginal execution time compared to "optimal" ILP approach, even presented with a large number of migration candidates and potential cloud locations.



(a) Latency-sensitive desktop VM.



(b) Latency-insensitive desktop VM.

Figure 3.12: **Comparison of WAN live migration's impact on desktop VMs running various applications.** After migration, latency-sensitive desktop VM that runs online streaming achieves higher VNC frame update frequency due to lower RTT, directly improving user experience. On the other hand, latency-insensitive desktop VM that runs text editing application does not see a obvious improvement after migration.

3.8.3 Live Migration and Virtual Desktop Performance

VMShadow's WAN live migrator actuates the migration decisions generated by the greedy shadow algorithm on hybrid cloud platforms by leveraging nested virtualization. In this experiments, we study the overheads of our WAN-based live migration approach, in terms of migration costs, as well as the user perceived performance benefits. We use the two Amazon sites in Oregon (US-West) and Virginia (US-East) for this experiment. The thin client is located in the Massachusetts private cloud. We run two desktop VMs in US-West. The first desktop represents a user running a text editing application and watching a YouTube video, while the second desktop represents a user only performing word editing. We perform live migration of each VM from Oregon data center to Virginia one, which is a site closer to the Massachusetts-based thin client, with the help of VMShadow's WAN migration component. For each live migration, we measure the total amount of data transferred and the time taken for the live migration as well as the time intervals between every VNC frame request and update.

As shown in Table 3.3, the delta-based and CBR optimizations used by VMShadow allow WAN migrations to be efficient; VMShadow can migrate a desktop VM with 1 GB memory coast-to-coast in about 250 to 265 seconds, depending on the workload. It is useful to note that the pause time (i.e., the time when a user may perceive any unresponsiveness) for the applications as a result of the migration is relatively small, between 2.5 and 2.8 seconds. Total migration time is determined by how much memory to transfer and how fast memory is dirtied. Therefore, it takes longer to migrate the virtual desktop that runs YouTube than Word editing. But for the pause time, it is determined by the amount of dirty memory to transfer in the last iteration. When watching YouTube video, data is being streamed and prefetched before the last iteration. Thus, the downtime of the YouTube virtual desktop is sightly smaller.

Figure 3.12 shows the response time before and after the migration for both desktop VMs. We define the response time to be the time interval between sending a refresh request and receiving a response. Therefore, the lower the response time, the higher the refresh rate. Note also that the VNC player only sends a refresh request after receiving a response to its previous request. Thus the response time for such players is upper bounded by the network round-trip time. As shown in the Figure 3.12a, initially the refresh rate is low since word editing does not require frequent refresh. The refresh rate increases when the user begins watching YouTube, but the refresh rate is bounded by approximately 100ms RTT between Oregon and Massachusetts, which limits VNC to no more than 10 refreshes per second (which is not adequate for a 24 FPS video clip). Once the VM has migrated from US-West to US-East, the RTT from the thin client to the desktop VM drops significantly (and below the dotted line indicating the minimum refresh rate for good video playback), allowing VNC to refresh the screen at an adequate rate. Figure 3.12b depicts the performance of the Word editing desktop before and after the live migration. As shown, word editing involves key - and mouse-clicks and do not require frequent refreshes due to the relatively slow user activities. Thus, the refresh rate is once every few hundred milliseconds; further a 100ms delay between a key-press and a refresh is still tolerable by users for interactive word editing. Even after the migration completes, the lower RTT does not yield a direct

	Total Time	Copy Time	Rewrite Time
Average (ms)	3.375	3.36	0.0133
Std. Dev.	0.022	0.034	0.0042

Table 3.4: **Per-packet proxy overhead.** We analyze the data copying and header rewriting of all packets that need to go through migration proxy.

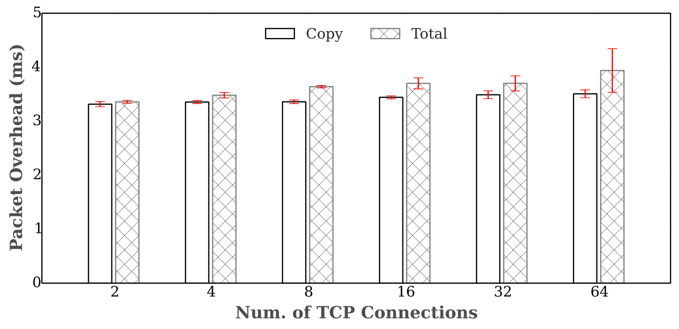


Figure 3.13: **Proxy processing overhead of TPC packet.** Proxy overhead comprises of copying network data between kernel and user-space and manipulating packets, i.e. iptable rule matching and header rewriting. The left bar group demonstrates the dominating copying overhead that is relatively constant to the number of active TCP connections.

benefit since the slow refresh rate, which is adequate to capture screen activities, is the dominant contribution to the response time.

Results: Migrating a desktop VM trans-continentally takes about 4 minutes depending on the workload while incurring 2.5 to 2.8 secs pausing time. Further, not all desktop applications see benefits from migrating to a closer cloud site, demonstrating our premise that not all desktop applications are latency-sensitive.

3.8.4 Connection Migration Proxy Overhead

Our connection migration proxy handles TCP connection migration in the case of public IP address change caused by WAN live migration. In this experiment, we evaluate the overhead of running our proxy at each desktop VM, specifically the overheads of processing each packet and rewriting their headers. To conduct this micro-benchmark, we have the desktop VM connect to a server machine and establish an increasing number of TCP socket connections. The desktop VM then sends or receives 10,000 packets over each socket connection and record the overheads incurred by the proxy as we increase the number of concurrent socket connections from 8 to 64. For each measurement data, We repeat this experiment for 10 times to gather all the measurement data for results in Table 3.4 and Fig 3.13.

The proxy overhead includes (i) data copying overhead incurred by *libnetfilter Queue* in copying packets from kernel-space to user-space and copying back to re-insert packets, (ii) matching a packet to rewrite rules, and (iii) rewriting packet headers. Table 3.4 depicts the per-packet overhead incurred by the proxy across all runs. As shown in the table, our user-space proxy adds a 3.37ms processing latency to each outgoing and incoming packet, and 13.2 μs packet header rewriting-related latency. This means that 98.5% of the additional latency is due to the overhead of copying packets between kernel and user space; the table shows a mean 3.36ms overhead of data copying. This overhead can be eliminated by moving the proxy implementation into kernel space. Figure 3.13 depicts the total processing time and copying overheads as the number of connections varies from 8 to 64. As expected, the per-packet copying overhead is independent of the number of connections. So is the overhead of rewriting headers for a given packet. As the number of connections grows, the number of rewrite rules grow in proportion, so the overhead of matching a packet to a rule grows slightly, as shown by the slight increase in the total processing overhead; this total overhead grows from 3.485ms to 3.976ms. Note that our implementation uses a naïve linear rule matching algorithm and this overhead can be reduced substantially by using more efficient techniques such as those used in routers to match ACLs.

Result: The dominant overhead of our proxy is due to data copying between kernel and user-space, with relatively efficient per-packet header rewrites and rule matching.

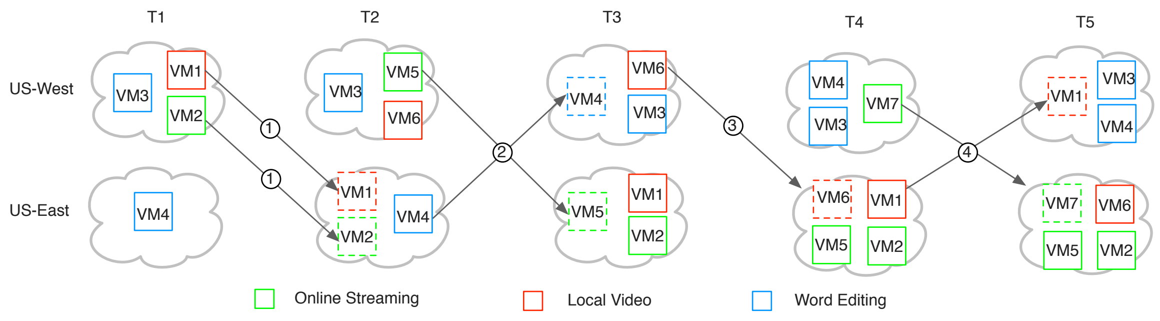
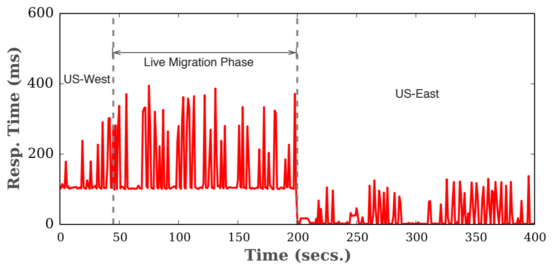


Figure 3.14: **Illustration of a series of migrations to improve the performance of Desktop VMs.** We consider a simplified scenario with two cloud locations, one is closer and the other is further to our thin clients in Massachusetts. VMShadow automatically identifies and prioritizes latency-sensitive desktop VMs, i.e. VDs that run local video and online streaming applications, and migrates them to the US-East cloud location. To accommodate latency-sensitive VMs in a resource-constrained cloud location, VMShadow reclaims the resource by migrating non latency-sensitive VDs to further cloud.

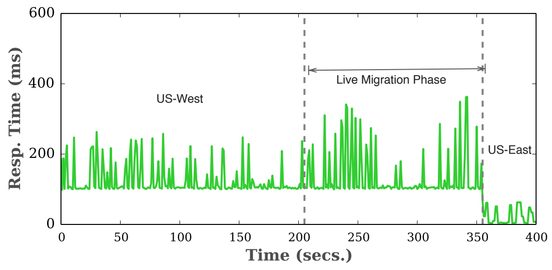
3.8.5 VMShadow Case Study

Last, we evaluate and show the work progress of VMShadow in fingerprinting and assigning latency-sensitive scores, and using WAN live migrations in resolving complex scenarios for improving the VDs' performance. The series of migration are depicted in Figure 3.14.

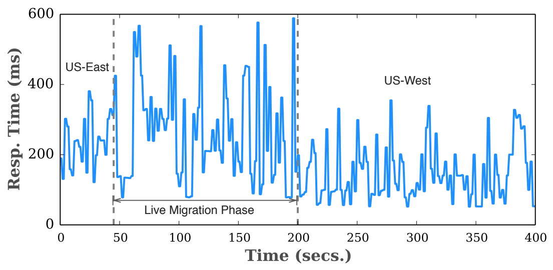
In this experiment, we consider three different types of applications, i.e., local video, text editing and online streaming running inside four identical VMs. For experimental purpose, we constrain US-East and US-West sites to both have a capacity of hosting 4 VMs each. Initially only the word editing VM is located at US-East, while the other three are located in US-West. At time T_1, VM_1 and VM_2 with local video and online streaming are ranked high as latency-sensitive and VMShadow triggers their migrations to closer Virginia cloud site. At time T_2, two new desktop VMs, i.e. VM_5 and VM_6, running video applications are requested and started in Oregon data center. Both of these VMs are also flagged as latency-sensitive and VM_5 is assigned higher latency-sensitive score. To accommodate both of these two VMs in the Virginia data center (currently only has capacity for one more VM), VMShadow first migrates higher rank VM_5 while at the same time reclaims resource



(a) Migration of VM_1 that ran Internet Streaming application.



(b) Migration of VM_2 that ran local video application.



(c) Migration of VM_3 that ran local word editor.

Figure 3.15: **Performance case study of migrating different latency-sensitive VMs.** Decisions are made to migrate VM_1 and VM_2 to US-East, to be closer to user. When US-East is resource-constrained, low-ranked VM_3's resources are reclaimed by migrating it back to US-West to free up resources for latency-sensitive VM_2.

by moving lower ranked VM_4 running text editing from Virginia to Oregon. At time T_3 after VM_4 has been successfully migrated, VM-Shadow then continues the process of migrating VM_6. At time T_4, we repeat the event of requesting a new virtual desktop for the user to watch a video streamed from YouTube. This leads to another swap between the newly requested online streaming VM_7 in US-West and the slightly lower ranked VM_1 in US-East. Eventually at time T_5, we end up having all the highly ranked desktop VMs running close to their end-users on the east coast, with lower ranked VMs running in US-West.

Figure 3.15 depicts the VNC response time for the three desktop VMs running different applications, before, during and after their migrations in the above scenario. As shown, the first two VMs have latency-sensitive video activities, and the VNC performance improves significantly after a migration to the US east coast (from 300ms to 41.7ms). The third VM has document editing activity, which does not suffer noticeably despite a reclamation and a migration to west coast, which is further away to its user.

Results: In this case study, we demonstrate VMShadow's ability to discriminate between latency-sensitive and latency-insensitive desktop VMs and to trigger appropriate WAN migrations to improve VNC response time in an artificially constrained cloud environments.

3.9 Related Work

The problem of placing VMs in data centers has been extensively studied. However, much of the focus has been, and continues to be, on placing VMs within a given data center. Approaches include devising heuristic algorithms [17, 65] or formulating placement as a multi-resource bin packing problem [40]. Others [133] have even proposed placement and migration approaches that minimize data transfer time within a data center.

Placement of VMs in a distributed cloud is complicated by additional constraints such as the inter-data center communication cost [30, 137, 155, 156, 171]. For example, Steiner et al. [156] demonstrate the challenges of distributing VMs in a distributed cloud using virtual desktop as an example application. There have been a few recent efforts aimed at addressing placement in the distributed cloud [9, 69, 70]. These approaches aim to optimize placement using approximation algorithms that minimize costs and latency [9], or through greedy algorithms that minimize costs and migration time [69, 70]. In this work, we dynamically place desktop VMs according to their latency-sensitivities. We seek to balance the performance benefit with the migration cost by taking multiple dimensions into account, including the virtual desktop user behavior, traffic profiles, data center locations and resource availabilities.

It has become a common practice to pair thin clients, devices with limited computation resources, with remote servers to accomplish compute-intensive tasks [45, 140]. The network distances between these remote servers and thin-clients directly impact user perceived performance and is a crucial metric in determining VM placement, especially for latency-sensitive applications. There has been prior work that evaluated the efficiency of thin-client computing over the WAN and showed that network latency is a dominant factor affecting performance [33, 99]. More recently, Hiltunen [79] et al. proposed per-user models that capture the usage profiles of users to determine placement of the front-ends and back-ends in a desktop cloud.

The ability to manipulate the VM locations agilely, either by cloning [97, 98] or migrating, is the primitive that allows us to adapt to changing latency-sensitivity of VMs. Virtualization platforms provide mechanisms and implementations to achieve LAN live migration with minimal disruption [39, 122]. Multiple efforts [29, 83, 87] have also sought to improve efficiency by either minimizing the amount of data transferred [87] or optimizing the number of times data was iteratively transferred [29].

Disruption-free WAN live migration [28, 80, 165, 167] is challenging due to lower wide area bandwidths, larger latencies, and changing IP addresses. Moreover different cloud locations can run different virtualization platforms. Xen-Blanket [165] provides a thin layer on top of Xen to homogenize diverse cloud infrastructures. CloudNet [167] proposed multiple optimization techniques to dramatically reduce the live migration downtime over the WAN. It also tried to solve the problem of changing IP addresses by advocating "network virtualization" that involved network routers. Others [76] have suggested using Mobile IPv6 to reroute packets to the new destination. There have also been several proposals [2–4, 126] that attempt to address the general problem of seamless handover of TCP connections across IP address changes. In general all these approaches require changes; either to the applications, the network, or both. In our work, we implement a prototype of VMShadow in Xen by reusing some ideas from CloudNet [167] and Xen-Blanket [165] and use a light-weight connection migration proxy that rewrites packet headers to cope with IP address changes and also to penetrate NATs.

3.10 VMShadow Summary

In this chapter, we presented VMShadow, a system that automatically optimizes the location and performance of VM-based desktops, with dynamic changing needs, running different types of applications. VMShadow performs black-box fingerprinting of a desktop VM's network traffic to infer latency-sensitivity and employs a greedy heuristic based algorithm to move highly latency-sensitive desktop VMs to cloud sites that are closer to their end-users. We empirically showed that desktop VMs with multimedia applications are likely to see the greatest benefits from such location-based optimizations in the distributed cloud infrastructure. VMShadow employs WAN-based live migration and a *new* network connection migration protocol to ensure that the desktop VM migration and subsequent changes to the VM's network address are transparent to end-users. We implemented a prototype of VMShadow in a nested hypervisor and demonstrated its effectiveness for optimizing the performance of VM-based desktops in our Massachusetts-based private cloud and Amazon's EC2 cloud. Our experiments showed the benefits of our approach for latency-sensitive desktops VMs, e.g. those that are running multimedia applications.

CHAPTER 4

PROVIDING GEO-ELASTICITY FOR MULTI-TIER APPLICATIONS

Cloud-based Internet applications are serving geographically distributed clients. As a result, cloud platforms, when provisioning resources, not only need to handle temporal workload variations but also spatial dynamics to satisfy a certain level of performance. In this chapter, we present GeoScale that combines model-driven proactive provisioning with agile reactive provisioning to handle long- and short-term spatial and temporal workload variations. GeoScale runs on top of public distributed clouds and provides a new property called geo-elasticity that can provision application server capacity at any cloud location based on the observed workload dynamics.

4.1 Motivation

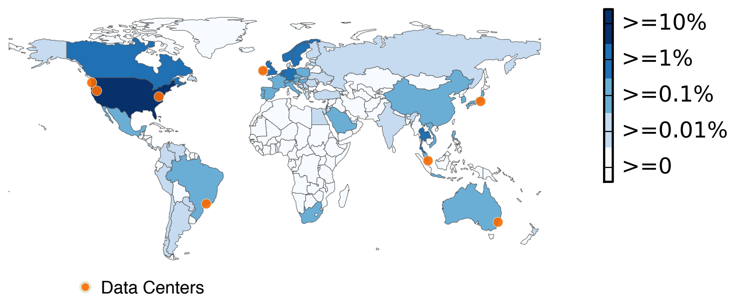
Today's cloud platforms provide numerous benefits to hosted applications such as a pay-as-you- go cost model and flexible, on-demand allocation of resources. Since many Internet applications see a dynamically varying workload, a key benefit of a cloud platform is its ability to autonomously and dynamically provision server resources to match a time-varying workload—a property that we refer to as *elasticity*. Cloud platforms such as Amazon support elasticity in the form of "auto-scaling" [10] where an application provider can choose thresholds on any system metric to automatically scale server capacity during workload spikes.

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.

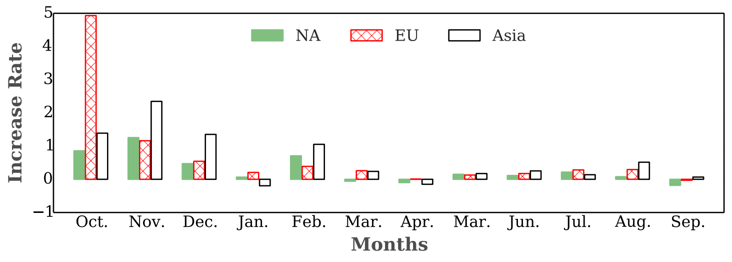
A distributed cloud that offers a choice of multiple locations to host cloud applications provides two key benefits. First, an application provider can choose a location closest to its user base to optimize user-perceived performance. Second, for those applications that have its users spread across multiple geographic regions, such a 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 will experience a dynamically varying workload, and the workload will include *both temporal* and *spatial* variations. Temporal variations include fluctuations 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 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 4.1, which shows the client distribution of Gowalla, a popular social application, and illustrates the global client base of such applications [38]. Figure 4.1a 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 4.1b 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 in users from Europe 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 [153], provisioning techniques for handling both temporal and spatial fluctuations in the workload are needed for applications with a geographically diverse user base.

In this chapter, we argue that a distributed cloud platform should support *geo-elasticity* to efficiently meet provisioning needs of Internet application with *geo-dynamic* workloads. 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 [102,152,153], 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, we make the following 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 that is seeded with empirical measurements to determine 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. *Finally*, 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.



(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.

Figure 4.1: Geo-dynamics seen by the Gowalla social media application's check-ins data.

4.2 Background and Problem Statement

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

Distributed Clouds. Our work assumes a 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 interfaces (APIs) 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 APIs. 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 distributed diverse 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). We further assume the task of maintaining consistency among back-end replicas is handled by application using any method that suits its need. For example, application can draw upon techniques [94, 130, 139] 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 application. We look at both model-based and reactive provisioning techniques that enable horizontal scaling by dynamically replicating the applications 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 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 imple-

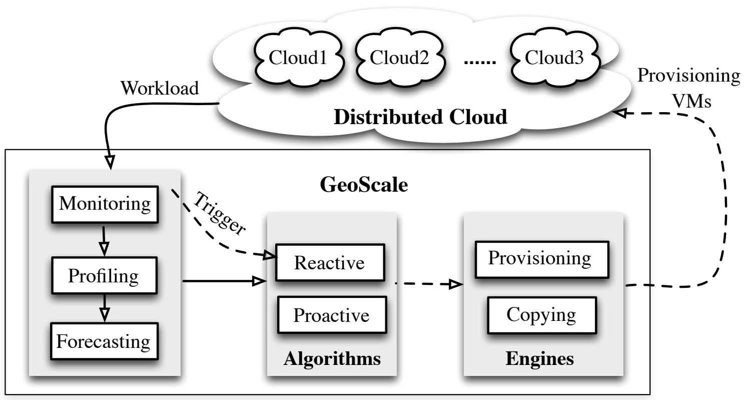


Figure 4.2: **GeoScale architecture.** GeoScale is comprised of three key components, 1) workload monitoring, profiling, and forecasting engine, 2) model-driven proactive and reactive geo-elastic provisioning algorithms, and 3) geo-elastic cloud provisioning and copying engine.

ment 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 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 end-to-end response time is minimized.

4.2.1 GeoScale System Architecture

We design GeoScale, as depicted in Figure 4.2, to provide geo-elasticity in distributed clouds. GeoScale is implemented as a middleware layer that uses cloud APIs to programmatically provision servers on the 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 of the application behavior to provision sufficient capacity to meet the application-specified SLA. Finally both the copying and provisioning engines use cloud APIs 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 4.3 and Section 4.4, followed by a detailed experimental evaluation in Section 4.5.

4.3 Providing Geo-Elasticity Using GeoScale

4.3.1 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 IP geolocation.[1] The client locations are then mapped to pre-configured *geographic bins* that are specified by 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 moni-

[1] GeoScale uses Maxmind [111] GeoIP location service.

tored. 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 [123], between each bin—represented by a weighted center of all requests within the bin—and data center pair by using the Heversine formula [1]:

$$\begin{aligned} a &= \sin^2(\frac{\Delta\phi}{2}) + \cos\phi_1 \cdot \cos\phi_2 \cdot \sin^2(\frac{\Delta\varphi}{2}) \\ d &= 2R \cdot \arcsin(\min(1, \sqrt{a})) \end{aligned} \tag{4.1}$$

where ϕ_i and φ_i represent the latitude and longitude of location i and R is the mean radius of earth. This yields a sorted list of data centers, in ascending order of distance, for each geographic bin.

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 not receive sufficient traffic (less than a threshold τ) to justify deploying an application replica, it is removed from the sorted list of the corresponding 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 following:

$$\begin{aligned} &\text{Minimize} && \sum_{i=1}^{M}\sum_{j=1}^{K} w_i X_{ij} D_{ij} \\ &\text{Subject to} && \\ &\sum_{i=1}^{M} X_{ij} T_i \geq \tau && \forall j = 1 \ldots K \qquad (4.2) \\ &\sum_{j=1}^{K} X_{ij} = 1 && \forall i = 1 \ldots M \qquad (4.3) \\ &X_{ij} \in \{0,1\} && \forall i = 1 \ldots M, j = 1 \ldots K \qquad (4.4) \end{aligned}$$

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, and T_i represents i^{th} bin's traffic volume. Constraint (4.2) ensures the chosen cloud location receive a threshold τ amount of traffic; Constraints (4.3) and (4.4) together ensure that each bin is mapped and to one cloud site only. 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 [78]. 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.

4.3.2 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 at longer time scales of hours or days and provisions server capacity to handle long-term workload trends observed at these time scales. We discuss reactive provisioning next in Section 4.3.3.

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 another existing close 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. We use Kingman's theorem [90] for G/G/1 queue under heavy traffic that states 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. Suppose SLA y is defined as the 95th percentile of server response time, we derive the upper bound on the maximum rate λ_j^c under heavy traffic as shown in Equation (4.5). A sketch of the derivation is provided in Appendix A and additional details can be found in [72].

$$\lambda_j^c < \left[\bar{x}_j + \frac{3({\sigma_{ja}}^2 + {\sigma_{jb}}^2)}{2\left(y - \bar{x}_j\right)}\right]^{-1} \tag{4.5}$$

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 95th percentile of server response time. Since all terms of Equation (4.5) 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 (4.6). 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,

$$S_j = \left\lceil \frac{\lambda_j^p}{\lambda_j^c} \right\rceil . \tag{4.6}$$

4.3.3 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 pre-copied 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). 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.

4.3.4 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 APIs to start a new server from the VM image. Once the server starts up, any additional network storage 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 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 require manual input from the application provider or involve making intelligent guesses based on which geographic regions 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. As noted earlier, in this case, GeoScale may choose to start up new servers in the other nearby location where VM disk images are already available, rather than copying data.

4.3.5 Discussion

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.

Whenever GeoScale makes provisioning plans that involve changing the set of active data centers, i.e., data centers that host application servers, it also decreases the time-to-live (TTL) value of DNS records on an authoritative name server. A shorter TTL value causes downstream DNS servers and caches to expire records faster. Therefore, DNS lookups can take longer, e.g., taking 1 second instead of 100 ms [7], because downstream caches might have expired. 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 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 is offset by hundreds of milliseconds network reductions for *every* client request.

4.4 GeoScale Implementation

We have implemented a prototype of GeoScale on Amazon's EC2 cloud. GeoScale is written in Python and Java, and is implemented as a middleware that uses EC2 APIs to provide *geo-elasticity* to applications across Amazon's global network of cloud data centers. GeoScale runs daemons on each of the application's servers to collect system statistics by using standard Linux utilities, e.g. sar. The application-level information is collected from application logs. We modified the Apache Tomcat server to log request service times, in addition to standard information such as response times and client information. If the application uses a load balancing software component, request logs can be gathered from

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

Table 4.1: Amazon's Distributed EC2 Cloud

this component, rather than from application logs at each server replica. All monitored data is sent to a central GeoScale server for archiving and further analysis. GeoScale uses the GeoIP-location service to map clients' IP addresses to their geographic locations and stores historical workload, system utilization and application-level information in SQLite database residing in the central node. GeoScale employs off-the-shelf ARIMA time-series model from Python's StatsModels to predict the future workload for the application and uses our geographic workload clustering to generate temporal and spatial workload distribution for the application. Periodically, GeoScale analyzes the predicted application workload distribution, and upon identifying either temporal or spatial changes, it uses proactive provisioning algorithm to determine a new set of cloud locations and the queueing model to estimate the number of servers needed for each location. Both the provisioning and copying engines are then invoked to spawn new servers or terminate existing ones using `boto`, an Amazon web service APIs. In the case of reactive provisioning, GeoScale makes tradeoff between provisioning time and achievable network latency by spawning up application replicas only on existing set of cloud locations. Last, our copying engine uses EC2 snapshot techniques to create and transfer full and incremental application-agnostic snapshots within and across data centers.

4.5 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 to current approaches such as using a CDN or 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.

Experimental Setup. Our experiments use Amazon EC2 as our distributed cloud. As shown in Table 4.1, 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 4.1. 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 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 [55] to automate the running of our experiments.

Application Workloads. We use a java implementation of TPC-W[3] 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

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

[3]http://jmob.ow2.org/tpcw.html

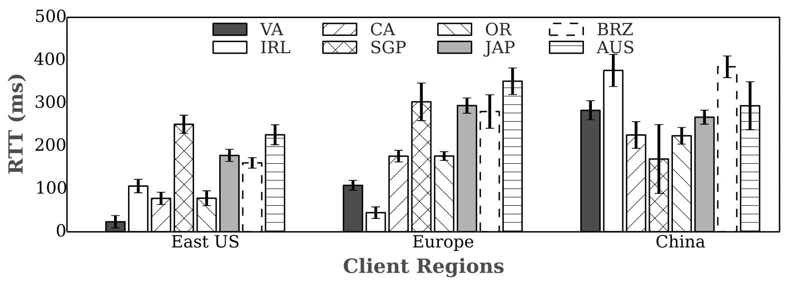


Figure 4.3: Clients from different regions observe minimal mean round-trip time at different cloud locations.

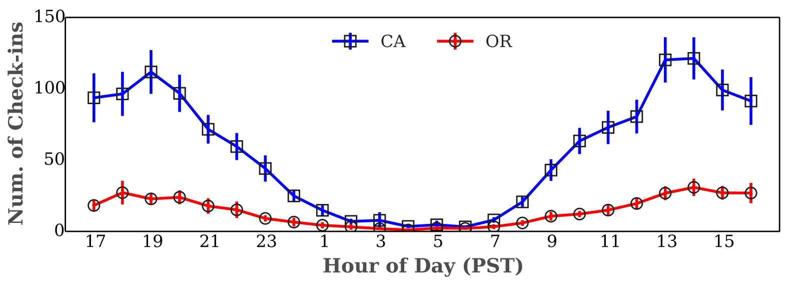
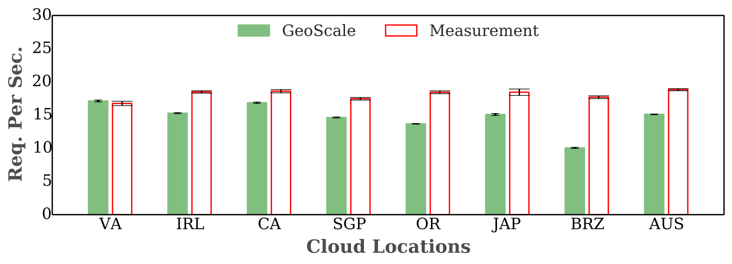
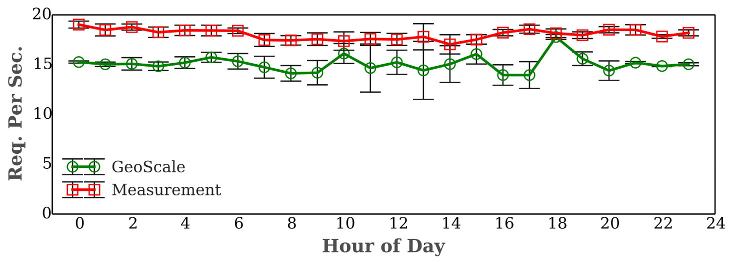


Figure 4.4: Gowalla workload's *temporal* time of day effects.

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 [38]. 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.



(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.

Figure 4.5: **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.

4.5.1 Exploiting Workload Dynamics

In Figure 4.1, 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 4.3 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 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 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 *geographically binning* its clients to the closet available cloud locations. Figure 4.4 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.

4.5.2 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.[4] 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 as the server capacity. At the same time, we collect the application service time and request inter-arrival rate statistics and use the queueing model described in Section 4.3.2 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.

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

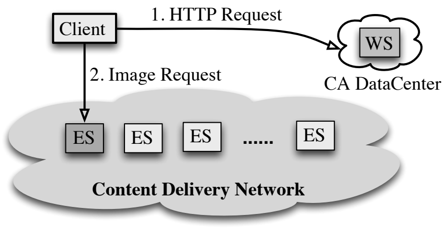
Figure 4.5a compares the server capacity predicted by GeoScale with the empirically measured ones for each cloud location. The measured server capacity varies across cloud locations, with up to a 12% difference in capacity across locations for the same type of server. Since the data centers were built in different years (see Table 4.1), we attribute these difference to variations in the underlying server hardware deployed at different locations. Figure 4.5a also emphasizes the need for a separate location-specific model to fully capture the hardware idiosyncrasies across location for the same type of server.

Figure 4.5b 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. We attribute these differences to interference from other co-located VMs on each physical server. Again GeoScale's models make conservative predictions with accuracies ranging from 75.19% to 97.91%. The larger 95^{th} confidence interval around noon is associated with the significant performance interference from other co-located VMs. This results indicate that, 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.

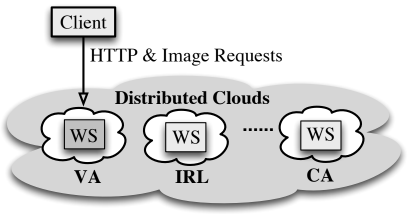
4.5.3 Comparison with a CDN

This section and then next compares 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. We begin with a comparison to a CDN-based approach.

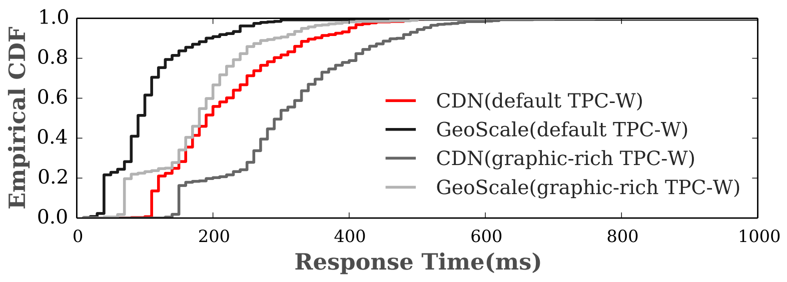
In the CDN case, we assume that the TPC-W application is deployed at Amazon's California data center and that all static content such as images are replicated in CDN servers



(a) Setup for CDN.



(b) Setup for GeoScale.



(c) CDF of client-perceived response time.

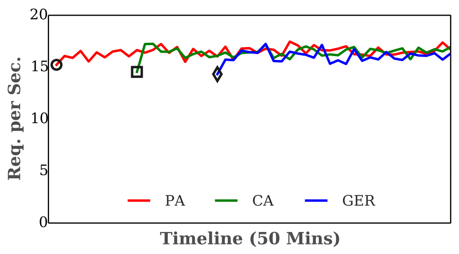
Figure 4.6: **A comparison of GeoScale's geo-elasticity to a CDN-based approach.** GeoScale outperforms the CDN-based approach for both default and graphic-rich browsing workload.

that are deployed in all 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 4.6a). In case of GeoScale, the TPC-W application is itself replicated across cloud locations as shown in Figure 4.6b 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).

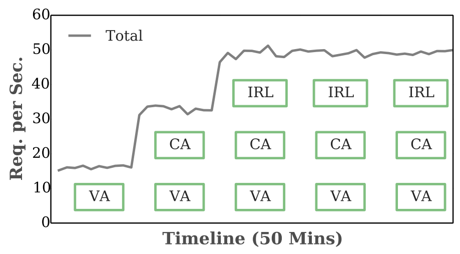
Figure 4.6c depicts the CDF of the response times seen by the client to load a page and all of its content for the TPC-W browsing workload mix. Table 4.2 depicts various percentiles of the response times for the two approaches. As shown, GeoScale outperforms the CDN approach, with a 40.43% reduction 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

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

Table 4.2: Different percentiles of the client-perceived latency for the CDN and GeoScale approaches.



(a) Client workload.



(b) Proactive Provisioning.

Figure 4.7: Illustration of proactive provisioning set up and result.

CDN approach. We then increase the size of static images served by the TPC-W application and repeat this experiment. As shown in Figure 4.6c and Table 4.2, 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 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.

4.5.4 Geo-elastic Proactive Provisioning Benefits

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

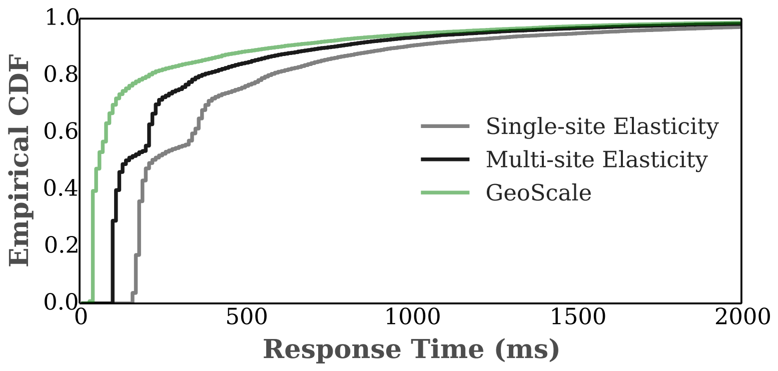


Figure 4.8: ECDF Comparison of client-perceived response time to single-site and multi-site elasticity. By employing geo-elastic proactive provisioning, GeoScale yields the best response times, with 95% of the requests finishing in less than 1060ms.

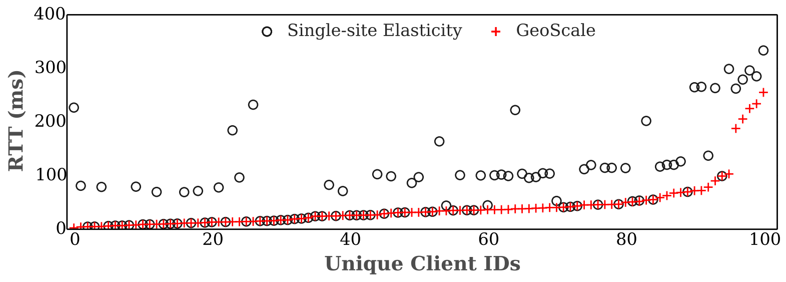


Figure 4.9: GeoScale improves the latency seen by more than half the clients over Single-site Elasticity.

cloud locations is made manually: (i) single-site elasticity (SSE) and (ii) multi-site elasticity (MSE). SSE is a centralized approach that hosts all replicates 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 4.3.2 to handle temporal workload fluctuations.

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 4.7a. Initially only the Pennsylvania clients send

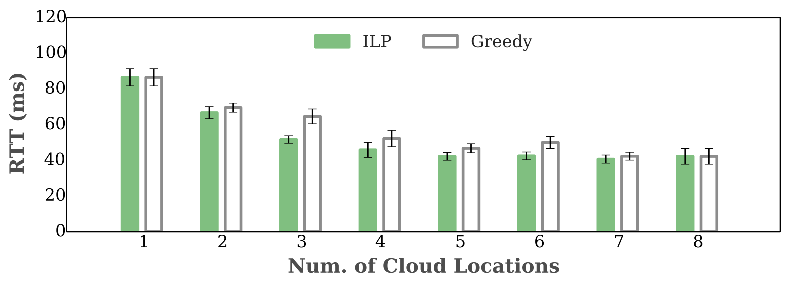
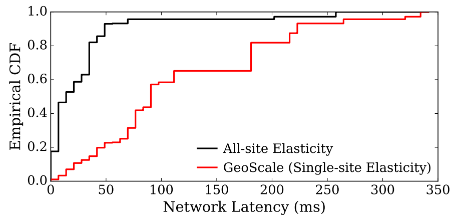


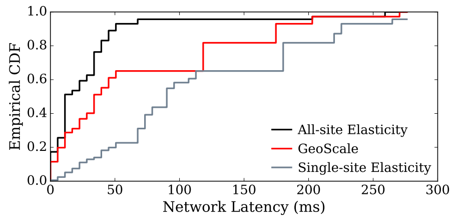
Figure 4.10: GeoScale's greedy workload clustering has comparable performance to the more expensive ILP approach.

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 4.7b.

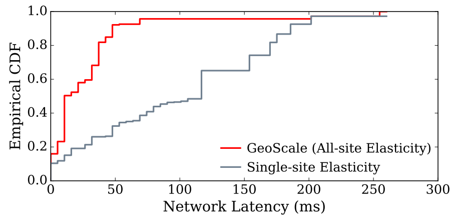
Figure 4.8 shows the CDF of the client response times across all clients for the three provisioning approaches. SSE has the highest response times since it uses a single cloud location to serve global traffic, causing distant clients to see worse response times. MSE approach uses a couple of fixed locations 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



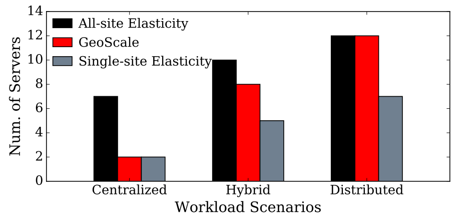
(a) Centralized workload.



(b) Hybrid workload.



(c) Geo-distributed workload.



(d) Provisioning cost.

Figure 4.11: 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.

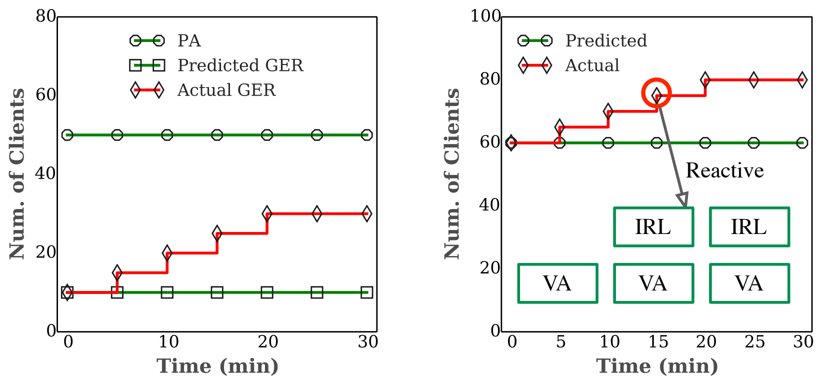
clients. The 95th percentile response time provided by GeoScale is around 1060 ms, a 31.17% improvement over SSE and a 13.11% improvement when compared to MSE.

Aside from the above experiment, we also use the network latency data collected between PlanetLab nodes and servers from 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 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 4.11, we plot the network latency distribution and provisioning cost achieved by three different techniques. In the first scenario (Figure 4.11a), e.g., applications that have regional popularity, GeoScale (equivalent to

SSE) yields network latencies with a 95^{th} percentile that is two times of the one 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 4.11d). In the last workload scenario (Figure 4.11c), 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 average network latencies seen by clients between SSE and GeoScale. In Figure 4.9, we plot 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 closest location 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 Plan-



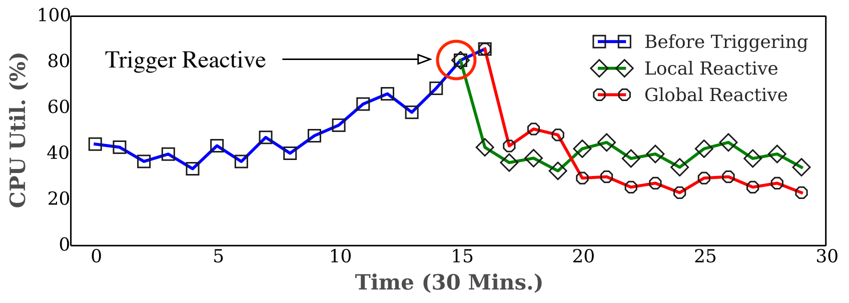
(a) Client workload. (b) Global reactive.

Figure 4.12: Illustration of reactive provisioning set up and result.

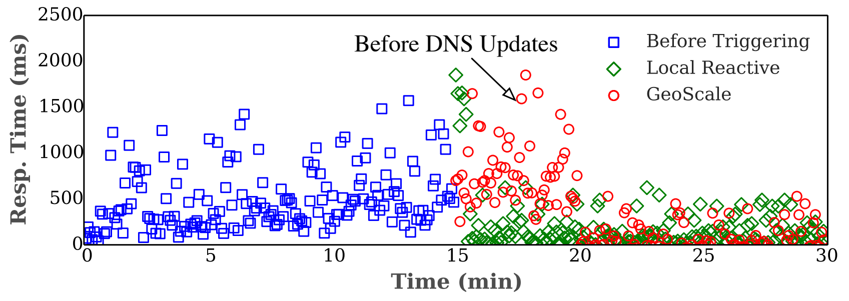
etLab client and Amazon data center, to produce mapping between each bin and best cloud location. We repeat each experiment ten times for different number of available cloud locations. Figure 4.10 depicts the mean network latency among all clients with increasing number of cloud locations. As shown in the figure, the greedy approach exhibits performance that is close to the more expensive ILP approach, indicating that greedy workload clustering algorithm will yield good results in practice.

4.5.5 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 4.12a. Since the predicted traffic from Germany is too low to justify deploying a server in Europe, no server is provisioned in 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



(a) Average CPU utilization of replicas.

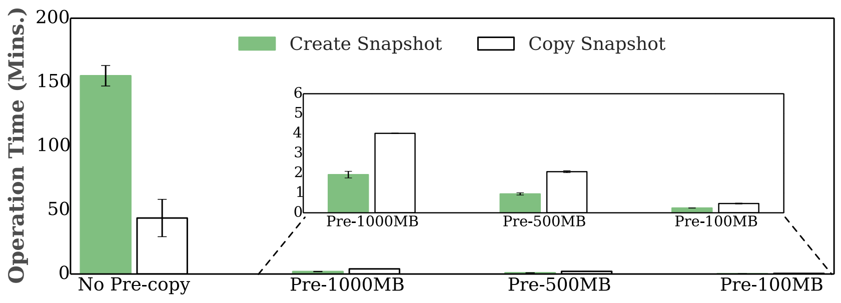


(b) Response times of clients.

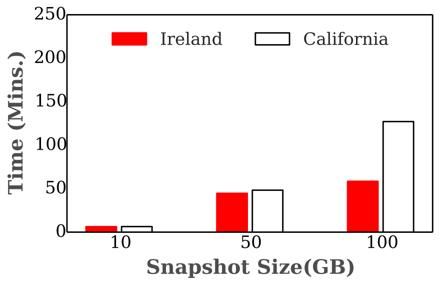
Figure 4.13: **Comparison of local and global reactive provisioning.** Global reactive provisioning leads to lower CPU utilization and better client response time compared to local reactive one.

reactive approach obtains workload data from the workload monitoring engine, and upon seeing the workload increase from Germany, provisions a second server in the Ireland data center that is closest to Germany, as shown in Figure 4.12b.

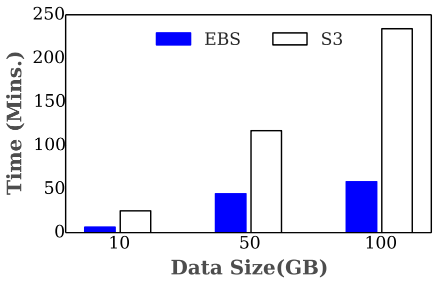
As shown in Figure 4.13a, the workload increase causes the server utilization to rise and SLA violations to increase steadily, until it reaches a point (orange circle) where reactive provisioning is invoked. It takes longer for server utilization to drop when using global reactive provisioning since Germany clients progressively switch to the new server due to DNS propagation delay of around 5 minutes. Similarly, in Figure 4.13b, 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.



(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.

Figure 4.14: Benefits of pre-copy optimizations.

4.5.6 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 4.14a 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 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 4.14a 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 4.14b 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 4.14c 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.

4.6 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" [92] techniques using VM live migration [39] or "scale out" [97] techniques by spawning replicas locally or in the public cloud [109]. 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. Application-level elasticity has been studied extensively in the context of databases; proposed techniques include the use of live database migration techniques [49, 51] for relational database and stop and migrate techniques [42] for key-value stores. Performance-aware replication of data in geo-distributed cloud stores has also been studied in this context [127]. Other related efforts target specific scenarios for elasticity that include transparent load balancing [102, 141] by moving per-flow state in the middle-box or rebalancing data among cloud storage systems.

Model-driven Approaches. Past work on model-driven techniques have focused on clustering techniques such as k-means [62, 153] or independent component analysis [148] to characterize dynamic workload or workload spikes [24], which is then used for providing elasticity. GeoScale uses geographic clustering of the workload into geographic regions, which is then mapped to the closest possible cloud location to achieve geo-elasticity. Other model-driven approaches [60, 108, 129] have relied on queueing theory with specific optimizations, such as using multi-model [108] or Kalman filtering [60] to provision virtualized resources in the same data center. 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 [41, 127, 145]. 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 [58], and application-layer optimizations [157].

4.7 GeoScale Summary

In this chapter, we presented GeoScale, a system that provides geo-elasticity to replicable multi-tier web application in a distributed cloud platform. 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 closet possible cloud locations to distributed user base. Our experimental evaluation of the GeoScale prototype on Amazon EC2 yielded up to a 40% reduction in the 95th percentile response times and up to a 58% reduction in SLA violations for representative web applications, when compared to traditional local elasticity mechanisms.

CHAPTER 5

PROVIDING GEO-ELASTICITY FOR DATABASE CLOUDS

In addition to exhibiting temporal and spatial workload variations, more applications are hosting their backend tiers separately for benefits such as ease of management. To provision for such applications, traditional elasticity approaches that only consider temporal workload dynamics and assume well-provisioned backends are insufficient. Instead, In this chapter, we propose a new type of provisioning—geo-elasticity, by utilizing distributed clouds with different locations. Centered on this idea, we build a system, called DBScale, that tracks geographic variations in the workload to dynamically provision database replicas at different cloud locations across the globe. Our geo-elastic provisioning approach comprises a regression-based model that infers database query workload from a spatially distributed front-end workload, a two-node open queueing network model that estimates the capacity of databases serving both CPU and I/O-intensive query workloads, and greedy algorithms for selecting the best cloud locations based on latency and cost.

5.1 Motivation

Cloud platforms are increasingly popular for hosting web-based applications and services. Studies have shown that more than 4% of Alexa top million websites [77] are now hosted on cloud platforms and these contribute to more than 1% of the Internet traffic. Cloud platforms come in many flavors. Today's Infrastructure-as-a-service (IaaS) clouds support flexible allocation of server and storage resources to their customers using virtual machines (VMs). Recently Database-as-a-service (DBaaS) clouds have become popular as a method for hosting databases for cloud applications. In a DBaaS cloud, a customer leases

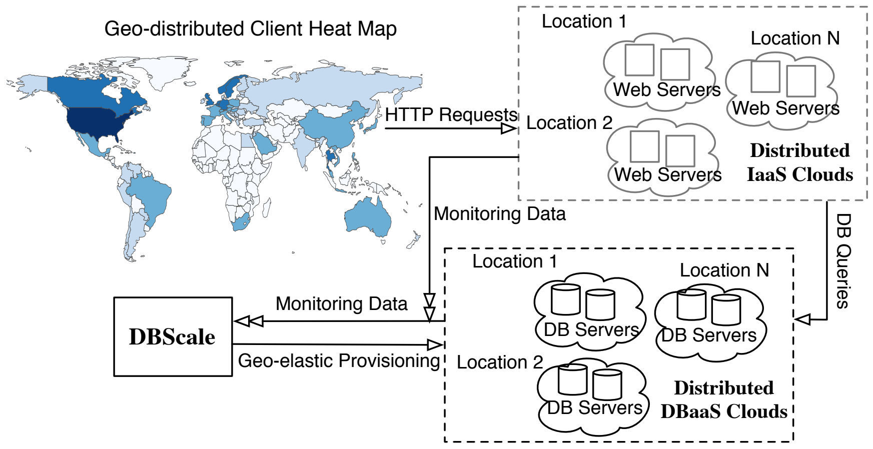


Figure 5.1: **Illustration of using DBScale to manage a multi-tier application.** In this example, the multi-tier application serves different amount of client workloads from different regions. The multi-tier application has its front-end web servers deployed in distributed IaaS clouds and its back-end database servers deployed in distributed DBaaS clouds.

a database from the cloud provider for storing and retrieving their data and offloads the tasks of managing and provisioning ("right-sizing") the database to DBaaS cloud provider. Since the application provider no longer needs to deal with the complexity of scaling their database to dynamic application workloads, DBaaS clouds simplify the task of building cloud applications. In such a scenario, a multi-tier web application is built by hosting the front-end tiers on servers leased from an IaaS cloud, while the back-end database tier of the application is hosted on a DBaaS cloud. A key benefit of IaaS and DBaaS cloud platforms is their ability to provide *elasticity*, where the cloud platform dynamically and autonomously scales the capacity allocated to the application or database tiers based on observed workload dynamics.

A concurrent trend is that today's cloud platforms are becoming increasingly distributed by supporting data centers in different geographic regions and continents. For instance, Amazon's EC2 and Microsoft's Azure offer a choice of eleven and seventeen global locations respectively to their customers today. Distributed clouds are especially well suited for deploying cloud applications that service a geographically diverse workload. For such applications, network latency between end users and server replicas still plays an important role in affecting overall performance [66, 154]. Therefore, a distributed cloud with a large set of locations provides the flexibility to deploy application replicas so that users can be serviced from the nearest cloud replica for the best performance. Studies [170] have shown that such *geo-distributed* application see geo-dynamic workloads, where the workload sees both spatial and temporal fluctuations. Thus, in addition to well-known temporal fluctuations such as time-of-day effects or seasonal fluctuations [15, 23], the application sees *spatial* fluctuations where workload volume in one geographic region (e.g., North America) fluctuates independently of the workload volume seen from other regions (e.g., Asia or Europe).

However, existing elasticity mechanisms, in the form of autoscaling within a physical cloud location boundary, are not well suited for handling spatial fluctuations seen in today's geo-distributed applications. The limitations are mainly two-fold. First, local elasticity mechanisms, when provisioning resources, are constrained to a single cloud location or a static subset of all available cloud locations. Second, current approaches are oblivious to the spatial workload dynamics associated with the geo-distributed applications. For instance, if an application that is deployed in two locations, say North America and Europe, sees a spatial increase in workload volume in Asia, current elasticity mechanisms will attempt to increase the provisioned capacity in the existing locations, whereas the proper response is to deploy new replicas in Asian cloud locations.

Instead, a different elasticity approach is needed that can handle both the temporal and spatial variations seen in today's geo-distributed applications' workload—we refer to such an approach as geo-elasticity. A geo-elasticity mechanism handles temporal changes by varying the provisioned capacity locally and handles spatial changes by provisioning replicas across regions and at new locations. Our paper specifically targets Database-as-a-service (DBaaS) clouds and focuses on designing a geo-elasticity mechanism for DBaaS clouds. Figure 5.1 presents a high level illustration of how DBScale interacts with a multi-tier application that is deployed using both IaaS and DBaaS.

We identify four key challenges in designing geo-elasticity for DBaaS clouds. First, because application database tiers only see workload traffic from front-end tiers but do not handle end client traffic directly, inferring geographic workload distributions and associated spatial fluctuations for database servers is more challenging than for front-end tiers. Second, prior work on dynamic provisioning [162] for multi-tier applications usually make simplified assumptions about CPUs being bottleneck resources. Those approaches may not be well-suited for database tiers because database can either be compute-intensive or I/O-intensive, or a mix of the two, depending on database query computational and I/O demands. Third, when a DBaaS cloud provisions database replicas, the task of maintaining consistency across replicas needs to be handled. Database consistency is a complicated task, especially in the presence of WAN replicas [5, 14]. In addition, consistency requirements are application-specific and therefore need to be handled differently for different applications. Finally, because end-to-end client performance depends on both front-end and back-end provisioning configurations, it is therefore very important to coordinate between IaaS and DBaaS to agree upon geo-elastic provisioning decisions and policies such as synchronizing provisioning completion time or using precopying.

Contributions. In this chapter, we tackle all four challenges with a middleware system DBScale. DBScale provides an end-to-end solution to providing geo-elasticity for Database-as-a-service cloud, from inferring dynamic database workloads of geo-distributed applications to provisioning database replicas in the "best" cloud locations. In designing and implementing DBScale, we make the following contributions:

First, we propose a regression-based technique that uses the observed geographic distribution of the workload seen by the front-end tier to infer the resulting geographic distribution of the queries seen by the database tier. This regression model is used as the basis to predict future spatial workload for geo-elastic provisioning.

Second, we present a technique that models each database replica as a two-node open queueing network with feedback, with the CPU modeled as a M/G/1/PS queue and the disk modeled as a M/G/1/FCFS queue. In doing so, our model can effectively identify the resource bottlenecks that hinder the server response time and provide the basis for provisioning enough amount of servers without violating T^{R}_{SLA}, response time SLA.

Third, we analyze the performance and cost trade-offs of database workload assignment and propose two greedy algorithms that prioritize different objectives within the constraints of network latency SLA, T^{N}_{SLA}. We also formulate an assignment problem using quadratic programming that minimizes operation cost (Appendix B).

Fourth, we implement a prototype of DBScale on Amazon EC2's distributed clouds and conduct detailed evaluations. Specifically, we run our experiments by injecting geo-distributed workloads from PlanetLab servers to a multi-tier application that are managed by DBScale in Amazon's distributed clouds. We compare the effectiveness of DBScale, in handling dynamic workload, to two other elasticity approaches—a local elasticity approach and a distributed caching approach. Our results show a 55% and a 36% improvement in mean response time when compared to local elasticity and the caching-based approach. In addition, we also evaluate our models and algorithms performance by comparing to benchmark measurements and through empirical data-driven simulations.

5.2 Background and Problem Statement

In this section, we first provide a background on distributed database clouds and then describe the application model assumed in our work and the specific problem of geo-elasticity in DBaaS clouds addressed in this work.

5.2.1 Distributed Database Clouds

Our work assumes a Database-as-a-service (DBaaS) cloud that allows application providers, also referred to as *tenants*, to lease one or more databases from the cloud platform. The DBaaS cloud provides SLAs on performance (e.g., response times) seen by the application and handles the task of configuring and provisioning sufficient capacity for each tenant.

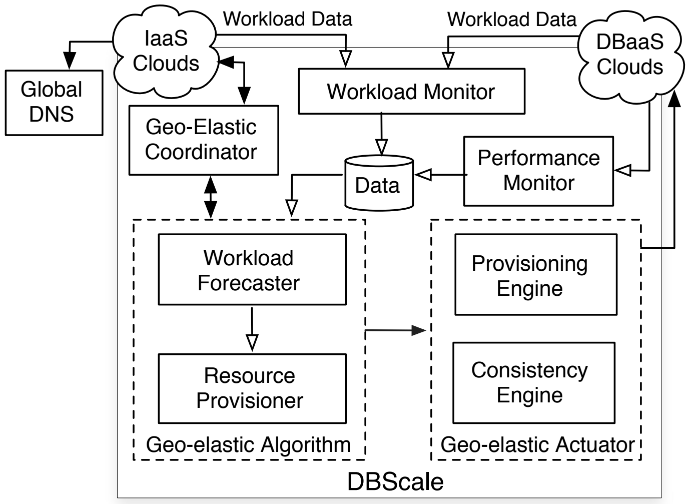


Figure 5.2: **Key components of DBScale.** For simplicity, we only demonstrate the design and architecture of DBScale's central controller and omit light-weight daemons that report back workload and performance data from within DBaaS clouds. Here, arrows with solid head represent control decisions made by DBScale.

Just as Infrastructure-as-a-service (IaaS) clouds support server instances of different sizes (e.g., small, medium or large servers), a database cloud also supports different types of database tenants. Small tenants, who have smaller storage and workload requirements, are hosted using a shared model where multiple small tenants share the resources of a single physical server. Large tenants, on the other hand, are hosted using a dedicated model, where each tenant is allocated all the resources of a physical server to support larger database or more-intensive workloads.

The DBaaS cloud itself can be implemented on top of a IaaS cloud where database tenants are housed in virtual machines ("server instances") of the IaaS cloud. While we assume such a virtualized environment for ease of prototyping, our approach could be easily generalized to non-virtualized setting. We assume that the DBaaS cloud is distributed and offers a choice of multiple locations to each application provider. Thus, an application may choose a particular cloud location that is best suited to its needs or a set of locations where application replicas are placed.

5.2.2 Application Model

Our work focuses on multi-tier web applications that consist of a front-end web tier and a backend database tier. We assume that the front tiers (HTTP and application tiers) are hosted on servers of a IaaS cloud, while the backend tier runs on a database in a DBaaS cloud. For simplicity, we assume applications use a single cloud provider that provides IaaS and DBaaS services. This assumption allows the front-tiers and the backend tier replicas to be hosted in the same data center of that cloud provider at each location where the application has a presence. We assume that the multi-tier application has users that are spread across multiple geographic locations and hence the application services a geographically diverse workload. Further, in addition to temporal variations such as time-of-day effects, such a geographically diverse workload is assumed to exhibit spatial variations, where the workload volume from different regions may vary independently (e.g., due to regional events or regional differences in the popularity of the application).

We assume the database tier (and the multi-tier application itself) is geo-distributed, with replicas in different regions, to handle the geographically diverse workload. Since the database tier is replicated, both within a particular cloud location and across locations, maintaining consistency of backend replicas is an important issue. We assume that consistency policies and mechanisms implemented by backend tier (and the DBaaS cloud) are dependent on the application's needs. In case of predominantly read-intensive database query workloads, such as those seen by databases hosting product catalogs of e-commerce store, a relaxed consistency technique may suffice, where the product catalogs replicas are updated periodically in batched mode. In other scenarios, where stricter consistency is desired, database replicas may need to be organized in a master-slave WAN configuration or a multi-master configuration[1]; these approaches will incur higher overheads, especially in WAN settings.

[1]Percona and EnterpriseDB both provide multi-master replication.

5.2.3 Geo-elasticity

Consider a distributed DBaaS cloud that hosts the database tier of geo-distributed multi-tier application as described above. The cloud platform is assumed to implement elasticity mechanisms where the number of database replicas of the tenant application is scaled up or down in response to workload dynamics. However, given the geographically diverse workload, simply scaling the number of replicas at a given location is insufficient; the distributed DBaaS cloud needs to consider *where* workloads increase or decrease and decide how many replicas are needed at each cloud location. Such elastic provisioning of capacity within and across cloud locations to handle both temporal and spatial workload fluctuations is referred to as *geo-elasticity*.

Our work focuses on providing geo-elasticity in a DBaaS cloud while we do not assume any knowledge of provisioning mechanisms used by IaaS clouds that hosts front-end tiers. However, we assume a cooperative IaaS cloud that is able to incorporate provisioning decisions from a DBaaS cloud. Coordinating provisioning decisions between front-end and back-end tiers can be beneficial for good end-to-end user performance. Further, we also assume DBaaS customers specify their desired server types and our algorithm only considers provisioning additional servers of the same type.

5.2.4 Problem Statement

Given a DBaaS cloud that has access to n data center locations L^k, and a cloud tenant that serves client workload from m geographic regions L^c, we want to figure out the corresponding database workload dynamics, $\boldsymbol{\lambda} = [\lambda_1, \lambda_2, \dots \lambda_m]$, for each provisioning period. Here, $\boldsymbol{\lambda}$ is a global database workload vector and each element λ_i represents peak query rates from i^{th} client location.

After obtaining $\boldsymbol{\lambda}$, we want to assign those workload to cloud locations that satisfy our objective whether it is performance or cost, given network latency service level agreement, T^N_{SLA}. Commonly, we will have more client geographic regions than data center locations—

that is $m > n$. Therefore, the above assigning process aggregates client workload into n cloud locations and yields workload $\boldsymbol{\omega} = [\omega_1, \omega_2, \ldots \omega_n]$, where ω_j represents workload that needs to be provisioned for j^{th} cloud.

Then we want to determine $\boldsymbol{D} = [d_1, d_2, \ldots d_n]$, number of database replicas to provision for each cloud location, based on server response time service level agreement T^R_{SLA}. Let d'_j denotes the number of database replicas already existing in cloud j, where $d'_j \leq 0$. By comparing d_j with d'_j for each cloud location j, we will decide to provision (or deactivated) $\mid d_j - d'_j \mid$ replicas.

5.2.5 Overview of our approach—DBScale

Figure 5.2 shows the high level design of DBScale that interacts with both IaaS clouds and DBaaS clouds. For completeness, we also depict a global DNS that should be notified whenever client-facing servers' IP addresses change. DBScale is responsible for dynamically provisioning databases in DBaaS clouds to handle temporal and spatial workload variations. Specifically, DBScale illustrated here is a central controller that monitoring/analyzing global workload, deciding how to assign client workload to cloud locations, coordinating these decisions with IaaS provisioning engine, and provisioning/configuring database servers. More implementation details can be found in Section 5.7.

In Figure 5.3, we also provide a typical DBScale workflow to proactively provision database servers based on collected workload and performance data. For each provisioning period, DBScale performs the following four actions: *training*, *predicting*, *aggregating*, and *provisioning* in sequence to generate provisioning decisions for each DBaaS clouds customer. For the rest of design-related sections, we explain in detail based on the above workload about monitoring and predicting geo-dynamics database workload in Section 5.3; handling CPU-intensive and I/O-intensive database workloads with queueing-based capacity model in Section 5.4; figuring out where to provision database servers based on latency-first and cost-first greedy algorithms in Section 5.5 (and a Quadratic Programming

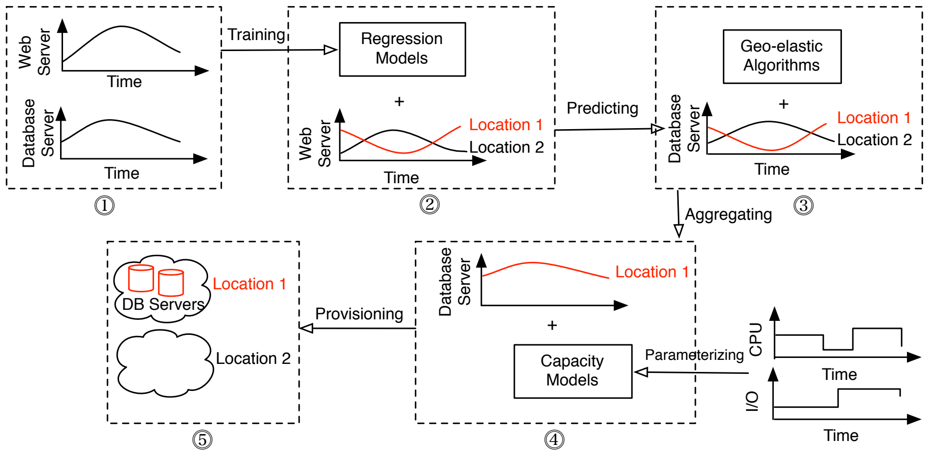


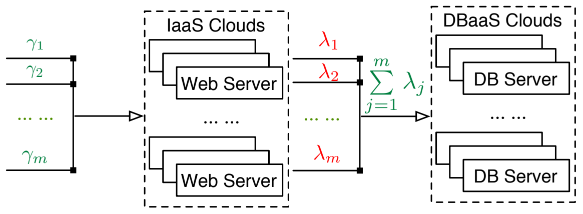
Figure 5.3: **Typical workflow of DBScale.** We generalize DBScale's actions into four categories: training, predicting, aggregating and provisioning. In this chapter, we take model-driven approaches using both regression and queueing models to estimate database workload and database server capacity. For choosing the best cloud locations, we use insights gained from linear programming formulation and a threshold-based greedy algorithm.

formulation in Appendix B); and providing a step-by-step procedure to provide geo-elastic database clouds in Section 5.6.

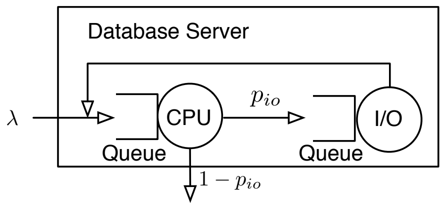
5.3 Geo-dynamic Database workload: where and how much?

In this section, we look at how to obtain database workload dynamics $\boldsymbol{\lambda} = [\lambda_1, \lambda_2, \dots \lambda_m]$ for each provisioning period. To do so, we need to be able to group queries to any one location j defined in L^c based on network proximity. However, queries are not directly associated with their originating clients—we do not know from which client location queries are generated. This is because client requests are first routed to web servers who then issue queries on behalf of clients. Therefore, the relationship between each query and its originating location is obscured by web servers.

One way to overcome this challenge is to define client regions L^c to be the same as IaaS cloud locations. Assuming IaaS has access to m' locations and clients are routed to the closest location among m' locations. By aggregating and analyzing database query



(a) A regression-based model for HTTP workload $\boldsymbol{\gamma}$, and database workload, $\boldsymbol{\lambda}$.



(b) A two-nodes queueing-model for a single database server.

Figure 5.4: **Model-driven geo-elastic approaches**. We model the relationship between front-end requests and database queries using regression-based models, and leverage these models to predict temporal and spatial database workload. For estimating a single database server's capacity at a specific, we model the server as a two-node open queueing network.

logs from all active database servers, we can assign queries to IaaS cloud locations based on web server IP addresses. This way, even if we don't know the relationship between individual query and its originating client, we obtain a *coarse* grained $\boldsymbol{\lambda}$ by using IaaS cloud locations as proxies—that is, database workload obtained using this approach is only reflecting spatial variations, if any, of front-end tier.

However, the effectiveness of the above approach depends largely on existing IaaS locations and whether IaaS employs geo-elastic provisioning. Currently, the number of IaaS cloud locations m' is in the low tens and therefore they might not be representative for a global workload distribution. If IaaS clouds only provision using a subset of m' locations, it will further reduce the usability of the above approach. That is, we will not be able to distinguish queries from Europe or US East if IaaS only provisions web servers in US East data center. In addition, this approach lacks the flexibility to produce $\boldsymbol{\lambda}$ for arbitrary client locations L^c, either in city, state or country levels. Such flexibility can lead to *fine* granularity client workload information that can result in better workload assignment decision as described in Section 5.5.

Given the limitations of above approach, we propose an effective regression-based approach that can produce database workload distribution $\boldsymbol{\lambda}$ with configurable precisions, using logs collected from both IaaS and DBaaS clouds.

5.3.1 Regression-based Workload Prediction

In this section, we show how to obtain database query rate $\boldsymbol{\lambda}$ using a regression model that captures the relationship between λ_i and web request rate γ_i, with the help of $\sum_{j=1}^{n} \lambda_j$. In Figure 5.4a, we show the interactions between variables that can be obtained directly using available logs (in green) and unknown variable (in red). Next, we first explain how to obtain γ and aggregate query rate $\sum_{j=1}^{n} \lambda_j$, and then introduce the regression model.

To obtain γ, we first aggregate front-end request logs from all cloud locations; the request logs are assumed to include at least a time stamp and the end client's IP address. We then use an IP Geolocation technique[2] to determine the originating client location of each request. Given client locations L^c, all requests are then mapped to one and only one location in L^c that is closest. We count the number of requests that are mapped to each client location in L^c. For each client location i, we group requests by their time stamps, and calculate the request intensity for specified time unit. At the end of this process, we will obtain $\gamma = [\gamma_1, \gamma_2, \ldots \gamma_m]$ that represents the peak web workload from each client location i. Similarly, we can process all database logs and obtain database query rates from each IaaS clouds and subsequently the total query rates $\sum_{j=1}^{n} \lambda_j$.

For a specific application, the number of database queries triggered by front-end requests might vary depending on the types of requests. For example, a request to search for the best selling products will have different database patterns than a request to finish placing order. Even so, it is still safe to assume that each front-end request will trigger *one or more* database queries—we model this relationship with $\lambda = \alpha\gamma + \beta$ where α captures the linear relationship and β is an error term. Further, to capture the potential regional effect caused by different client workload pattern, we use linear models with different parameters (α_i, β_i) to model the relationship between front-end requests and the corresponding database queries from each client location.

[2]IP Geolocation is a technique that infers user's geographic location from IP address. We use MaxMind GeoIP2 [111] for this task.

$$\lambda_i = \alpha_i \gamma_i + \beta_i, \quad i = 1, 2 \ldots N \tag{5.1}$$

Note we can't obtain query rate λ_i by processing logs without knowing how clients are mapped to IaaS cloud locations. Therefore, we can't solve (5.1) directly. However, relying on the fact that requests generated by all client locations are eventually contributing to the amount of database queries, we have

$$\begin{aligned}
\sum_{j=1}^{n} \lambda_j &= \sum_{i=1}^{m} (\alpha_i \gamma_i + \beta_i) \\
&= \alpha_1 \gamma_1 + \alpha_2 \gamma_2 + \ldots + \alpha_m \gamma_m + \sum_{i=1}^{m} \beta_i \\
&= \alpha_1 \gamma_1 + \alpha_2 \gamma_2 + \ldots + \alpha_m \gamma_m + \beta.
\end{aligned}$$

Here $\sum_{j=1}^{n} \lambda_j$ is the total database queries aggregated from all database replicas in L^k. For each provisioning period of length E, we prepare a data set $\left\{(\sum_j^n \lambda_j)_e, (\gamma_1)_e, (\gamma_2)_e, \ldots (\gamma_m)_e\right\}_{e=1}^{E}$ following above procedures. To find a model $(\alpha_1, \alpha_2, \ldots \alpha_m, \beta)$ that best explains these E data points, we use Least Squares Regression[3] to minimize the sum of squared residuals

$$\min_{\alpha_1, \alpha_2, \ldots \alpha_m, \beta} \sum_{e=1}^{E} \varepsilon_t^2, \tag{5.2}$$

where $\varepsilon_t = \alpha_1 \gamma_1 + \alpha_2 \gamma_2 + \ldots + \alpha_m \gamma_m + \beta - \sum_j^n \lambda_j$.

By solving Equation (5.2), we are only rewarded with a collective value β. To obtain $\{\beta_1, \beta_2, \ldots \beta_m\}$, we use a weighted function (5.3) that distributes β to β_i based on corre-

[3]Other regression techniques could be applied here as well, such as robust linear model with Huber loss function or TukeyBiweight.

sponding workload portion—the more requests from a client location i, the more weight we assign to β_i.

$$\beta_i = \beta \frac{\gamma_i}{\sum_{i=1}^{m} \gamma_i} \tag{5.3}$$

Combining (5.2) and (5.3), we obtain m linear regression models (α_i, β_i) for each client location i and can use them to estimate the number of queries λ_i based on (5.1). To be more specific, to predict the number of queries λ_i from location i at time $E+1$, we first take a series of M data points $[(\gamma_i)_{E-M}, (\gamma_i)_{E-M+1}, \ldots (\gamma_i)_E]$ and use ARIMA models [26], or any other standard time series prediction techniques, to predict $(\gamma_i)_{E+1}$. Then by substituting predicted front-end requests $(\gamma_i)_{E+1}$ into $i^t h$ client location's regression model, we get $(\lambda_i)_{E+1} = \alpha_i(\gamma_i)_{E+1} + \beta_i$. We repeat the above steps for all m client locations and eventually obtain $(\boldsymbol{\lambda})_{E+1} = [(\lambda_1)_{E+1}, (\lambda_2)_{E+1}, \ldots, (\lambda_m)_{E+1}]$, the database query distribution for time $E+1$.

5.4 Provisioning Based on SLA-bounded Database Capacity

To provision enough database servers to handle query workload, we need to have a way to estimate how many queries each database server can handle without violating service level agreement (SLA). One approach is to gradually increasing realistic query workload intensity until server response time is violated. The maximum number of queries the database server can sustain is then be used as server capacity. However, this approach might be less desirable because it requires offline profiling and more importantly, it heavily relies on having perfect knowledge of database workload.

Therefore, we resort to queueing theory to estimate capacity online. In addition, in most scenarios, SLAs are specified as a high percentile of response time distribution. Queuing theory also helps us reason about the tail behavior of the query service time distribution.

Specifically, by using queueing models, we can analyze the response time distribution and obtain tail behavior, and compare it with the pre-specified SLA to obtain server capacity.

5.4.1 Queueing-based Capacity Estimation

Most prior work [118, 162, 164] on using queueing-based models to perform dynamic resource provisioning only focus on estimating capacity for front-end tiers and assume CPU to be the bottleneck resource. Such approaches might not be ideal for database tiers because databases may need to serve queries that are either CPU-intensive or I/O-intensive, or a mix of both. To account for both resource impacts' towards query service time, we present a database-specific queueing-based model that keeps track of both CPU and I/O utilizations. Note, we assume front-end servers will send queries directly to individual database servers—these database servers do not share a centralized queue and therefore are modeled individually.

Specifically, we model the database replica (on a dedicated host) as a two-node open queueing network with feedback, where the CPU is modeled as a M/G/1 processor sharing (M/G/1/PS) queue and the I/O device as a M/G/1 first come first serve (M/G/1/FCFS) queue as in Figure 5.4b. Here, we model the arrival of external queries as a Poisson process with average rate of λ. Immediately following this, queries arrival at CPU and I/O also satisfy poisson distribution with λ_{cpu} and λ_{io}.

$$\begin{cases} \lambda_{cpu} = \dfrac{\lambda}{1-p_{io}} & \text{(5.4a)} \\[2ex] \lambda_{io} = \dfrac{p_{io}}{1-p_{io}}\lambda & \text{(5.4b)} \end{cases}$$

Here, a query arriving at a database server will first be added to the CPU queue. When the query departs from CPU, it will either leave the database server with probability $1-p_{io}$ or continue its processing by joining I/O queue with probability p_{io}—that is, p_{io} is the query visit ratio to I/O. In a high level, a query might alternate between CPU and I/O multiple times before a response is generated.

By modeling queries going through both the CPU and I/O, this two-node queueing network is able to factor in both CPU and I/O's contributions in affecting query response time. Let us denote query response time using T, the mean response time $E[T]$ of database queries is then the sum of time spent in CPU and I/O, i.e., $E[T] = E[T_{cpu}] + E[T_{io}]$. We use a recent result from a queueing literature [27] that provides approximation for both $E[T_{cpu}]$ and $E[T_{io}]$.

$$\begin{cases} E[T_{cpu}] = \dfrac{\bar{s}_{cpu}}{(1-p_{io})(1-\rho_{cpu})} \\ E[T_{io}] = \dfrac{p_{io}}{1-p_{io}}[\dfrac{\bar{s}_{io}}{1-\rho_{io}} + \dfrac{p(\bar{s}_{io}^{(2)} - 2\bar{s}_{io}^{2})}{2(1-\rho_{io})}\lambda], \end{cases} \tag{5.5}$$

where $\bar{s}_{cpu}$ and $\bar{s}_{io}$ denote average service time of CPU and I/O; ρ_{cpu} and ρ_{io} denote average utilizations of CPU and I/O; $\bar{s}_{io}^{(2)}$ is the second moment of the service time distribution of I/O.

Now, given a pre-specified SLA between DBaaS and database customers in the form of 95th percentile response time T_{SLA}^{R}, we need to satisfy the constraint $\alpha_T(95) < T_{SLA}^{R}$. Here $\alpha_T(95)$ denotes the 95th percentile of response time T. If we assume T satisfies an exponential distribution,

$$P\Big(T \leq \alpha_T(95)\Big) = 1 - e^{-\frac{1}{E[T]}\alpha_T(95)}, \tag{5.6}$$

based on the definition of cumulative distribution function. Therefore, we have $e^{-\frac{1}{E[T]}\alpha_T(95)} = 0.05$ and by taking ln of both sides,

$$\alpha_T(95) = \ln 20 E[X] \approx 3E[X] \tag{5.7}$$

Given the relationship[4] (5.7) and the SLA constraint, we then have $E[T_{cpu}] + E[T_{io}] < \frac{T^R_{SLA}}{3}$. By substituting (5.5) into the previous inequality, we obtain an upper bound on the maximum query rate λ^c that can be handled by a *single* database server at a specific cloud location violating the SLA T^R_{SLA}:

$$\lambda^c \leq \frac{\frac{2T^R_{SLA}}{3}(1-p)(1-\rho_{io}) - 2\bar{s}_{cpu}\frac{1-\rho_{io}}{1-\rho_{cpu}} - 2p\bar{s}_{cpu}}{p^2(\bar{s}^{(2)}_{io} - 2\bar{s}^2_{io})}. \tag{5.8}$$

5.4.2 Obtaining Model Parameters

Here, we explain how to obtain all model parameters for estimating $E[T]$ either by direct measurements or reasonable approximations. First, we need to empirically measure the CPU utilization, I/O utilization (using Linux tools such as sysstat) as well as per-query log that includes query timestamps and query execution time(by turning on MySQL slow logging and setting the `long_query_time` to 0 to record every query executed). We can directly estimate ρ_{cpu} from CPU utilization logs at a predefined time granularity, database query arrival rate λ by processing the per-query log and $\bar{s}_{io}$ and λ_{io} from the I/O utilization log. Based on (5.4b), we can estimate p_{io} by substituting λ and λ_{io}. Since we do not have easy access to $\bar{s}_{cpu}$ and ρ_{cpu}, we approximate these two parameters using the Little's Law [104]. Specifically, $\bar{s}_{cpu} = \rho_{cpu}\frac{1}{\lambda_{cpu}}$ and $\rho_{io} = \lambda_{io}\bar{s}_{io}$ where we obtain λ_{cpu} using (5.4a). Note that these are overestimations due to extra logging overhead and resource

[4]For a general distribution, we can use *Markov Inequality* to obtain $\alpha_T(95) \leq 20E[T]$ and follow the same steps to obtain a bound on α^c.

interference—that is, we make conservative estimates of λ^c. Finally, to estimate database server capacity for a new datacenter location, we use the average of measured statistics across all available data centers as an initial approximation.

5.5 Network SLA constrained Workload Assignment

In this section, we look at *where* to provision server resources for client workload $\boldsymbol{\lambda} = \{\lambda_1, \lambda_2, \dots \lambda_m\}$ to satisfy T^N_{SLA}, 95^{th} percentile network latency. Without loss of generality, we normalize $\boldsymbol{\lambda}$ with server capacity λ^c from (5.8) and obtain a new normalized workload vector

$$\boldsymbol{\lambda^N} = \{\lambda^N_1, \lambda^N_2, \dots \lambda^N_m\}, \quad \lambda^N_i = \frac{\lambda_i}{\lambda^c}. \tag{5.9}$$

Effectively, $\lceil \lambda^N_i \rceil$ represents the number of servers needed for client location i. Let us define χ_{ij} as the fraction of client workload λ_i that is assigned to and provisioned in cloud location j. Here $\chi_{ij} \in [0,1]$. An eligible assignment matrix $\chi_{m\times n}$ is one that satisfies T^N_{SLA}. Given the assignment matrix χ, we can express the normalized database workload for cloud location j

$$\begin{aligned} \omega_j &= \chi_{1j}\lambda^N_1 + \chi_{2j}\lambda^N_2 \cdots + \chi_{mj}\lambda^N_m \\ &= \sum_{i=1}^{m} \chi_{ij}\lambda^N_i. \end{aligned} \tag{5.10}$$

Here, the number of servers that need to provision for cloud location j is then $\lceil \omega_j \rceil$. We define a cost and performance metrics as following and use them as guidelines to evaluate the effectiveness of eligible assignments χ.

Cost Analysis. We consider three different cost aspects in calculating the operational expenditure (OPEX) of serving $\boldsymbol{\lambda}$ workload for the next provisioning period (V hours). Our cost analysis is based on current commercial cloud pricing models, specifically, we use Amazon's model as a concrete example. The first cost is hourly cost c^s for renting server resources. Therefore, the total rental cost is $C^s = c^s \lceil \omega \rceil V$, proportional to the number of servers and renting time.

The second cost we consider is data storage cost. The storage need at each server is defined as a continuous random variable D. We denote the database size at the beginning of each provisioning period as d^{DB} and the probability of inserting a new data entry as p^{ins}. The size of new data entry is denoted with a continuous random variable U and we assume knowledge of $\mathbb{E}[U]$. Therefore, the expected storage need of a server at v^{th} hour is $\mathbb{E}(D \mid V = v) = d^{DB} + \lambda^c p^{ins} \mathbb{E}[U] v$. Let us define c^d as the hourly cost for storing unit amount of data. Thus, the total storage cost across all servers is $C^d = \sum_{v=1}^{V} \mathbb{E}(D \mid V = v) \lceil \omega \rceil c^d$.

The last cost we consider is the cost for transferring data needed for provisioing. We define a continuous random variable R to express the size of outbound Internet traffics for each server. The expected data transfer for v^{th} hour in one provisioning period is $\mathbb{E}(R \mid V = v) = \lambda^c \mathbb{E}[U] v$. Let us define c^t as the hourly cost for transferring unit amount of data, we can then express the total transfer cost as $C^t = \sum_{v=1}^{V} \mathbb{E}(R \mid V = v) \lceil \omega \rceil c^t$. In all, by combining all three cost components, we have the formula to calculate cost to serve ω_j workload at cloud location j

$$C(\omega_j) = C_j^s + C_j^d + C_j^t. \tag{5.11}$$

Performance Analysis. We are interested in analyzing the achieved network latency between client and database servers. To do so, we record latency values between communicating client and server and obtain the true network latency distribution. This type of application-level measurements provide good estimates for all individual latency pairs. However it is excessive for our use cases and it might not always be feasible. Instead, we approximate the true distribution with $T^N = \{(A_{ij}, n_{ij}) \mid \forall i \in L^c, \forall j \in L^k\}$ where n_{ij} denotes the number of occurrences of latency A_{ij} between client location i and cloud location j. In essence, T^N is a multiset and each element A_{ij} has multiplicity n_{ij}. T^N is a reasonable approximation for targeted network latency because client location i represents a cluster of clients from nearby geographic locations.

5.5.1 Greedy Algorithms

Next we describe two greedy algorithms that focus on minimizing either network latency or provisioning cost. Each algorithm produces a workload assignment matrix χ that is associated with a provisioning cost C and network latency distribution T^N.

Latency-first Greedy. We first sort all client locations L^c based on their workload intensity in descending order. For each client location j, we first find the set of eligible cloud locations $S_i = \{j \mid A_{ij} \in [0, T^N_{SLA}]\}$. That is, a cloud location j is said to be eligible for client location i if A_{ij}, the network distance between these two locations is smaller than T^N_{SLA}. Note that, $S_i \neq \emptyset, \forall i$ based on our assumption of T^N_{SLA}.

We then assign workload λ_i from client location i to the closest cloud location

$$O_i^{lat} = \arg\min_{j \in S_i} A_{ij},$$

if there are enough resources. Otherwise, we move to the next closest cloud location in S_i until we successfully acquire an eligible cloud location. We repeat the above process for all client locations L^c and eventually reach a valid assignment χ^{lat}. For simplicity, we assume the total resources from all eligible cloud locations are sufficient to satisfy the demand of workload $\boldsymbol{\lambda}$. Based on χ^{lat}, we can express the aggregate workload for each cloud location and the associated cost and network latencies in (5.12) – (5.13).

$$\omega_j^{lat} = \sum_i^m \lambda_i^N \mathbb{1}\{O_i^{lat} = j\}, \quad \forall j \in L^k \tag{5.12}$$

$$C^{lat} = \sum_{j=1}^n C(\omega_j^{lat}) \tag{5.13}$$

$$T_{lat}^N = \{(A_{iO_i^{lat}}, \lambda_i) \mid \forall i \in L^c\}. \tag{5.14}$$

Cost-first Greedy. Observe that we might get a cheaper assignment than *latency-first* algorithm by considering the cost differences between different cloud locations, as shown in Figure 5.5. Based on this observation, we propose the *cost-first greedy* algorithm that assigns client workload to the cheapest eligible cloud.

Instead of choosing the closest available cloud location, we choose the cheapest location $O_i^{cos} = \arg\min_{j \in S_i} C(\lambda_j)$. This yields a different assignment matrix χ^{cos} that is associated with

$$\omega_j^{cos} = \sum_i^m \lambda_i^N \mathbb{1}\{O_i^{cos} = j\}, \quad \forall j \in L^k \tag{5.15}$$

$$C^{cos} = \sum_{j=1}^n C(\omega_j^{cos}) \tag{5.16}$$

$$T_{cos}^N = \{(A_{iO_i^{cos}}, \lambda_i) \mid \forall i \in L^c\}. \tag{5.17}$$

Discussions. Note neither greedy approaches, when choosing the cloud location for client workload, consider the existing client assignment. Thus, we might end up provision $n-1$ extra servers than we should have for workload $\boldsymbol{\lambda}$. This is because in the best case scenario, we only need to provision a total of $\lceil \sum_{i=1}^m \lambda_i^N \rceil$—that is, client workload is aggregated perfectly. While in the worse case scenario, we end up provision one extra server for every cloud location to handle $\omega_j - \lfloor \omega_j \rfloor$ fractional of workload.

To further reduce the cost overhead, we include a quadratic programming (QP) formulation in Appendix B. But in practice, QP formulation might not be desirable due to limited benefits and time complexity. Specifically, the potential cost saving is bounded by the cost of renting $n-1$ servers. In current pricing models, QP formulations do not have effect on reducing either storage or bandwidth costs. When the number of cloud locations is reasonably small compared to provisioned servers, the saving of QP is negligible. As the number of cloud locations grows, the time complexity of solving this QP increases significantly. This makes it impractical as an online provisioning solution.

5.6 Putting It Together

DBScale combines regression-based workload prediction, queueing-baed capacity estimation and greedy workload assignment algorithms to implement geo-elasticity for DBaaS, as summarized below. DBScale periodically involves the following four steps, e.g. every day, or when SLA s are violated.

Step 1: Where to provision? DBScale obtains database workload distribution $\boldsymbol{\lambda}$ from all m client locations using regression-based prediction model described in Section 5.3.1. Then, DBScale uses one of the algorithms from Section 5.5 based on customer's specification to generate workload assignment matrix $\chi_{m \times n}$—each entry χ_{ij} specifies how much workload from client location i is to be assigned to cloud location j. Last, DBScale figures out workload to be provisioned for cloud locations, $\hat{\boldsymbol{\omega}} = \boldsymbol{\lambda}_{1 \times m} \boldsymbol{\chi}_{m \times n}$. Those cloud locations with non-zero $\hat{\omega}_j$ are chosen for provisioning for the next period.

Step 2: How many resources to provision for each location? DBScale first parameterizes queueing-based capacity model, as described in Section 5.4, and then computes the maximum query rate λ^c that a single replica can handle without violating SLA T^R_{SLA}. Given the amount of query workload $\hat{\omega}_j$ a cloud location j is assigned for next provisioning period, DBScale then calculates the number of replicas $d_j = \lceil \frac{\hat{\omega}_j}{\lambda^c} \rceil$. If the number of replicas d_i differs from the value d'_i computed in the previous time interval (i.e. current provisioning), $|d_i - d'_i|$ more replicas need to be provisioned or deactivated at this location. If $d'_i = 0$, this indicates that location i has been newly chosen to provision database replicas.

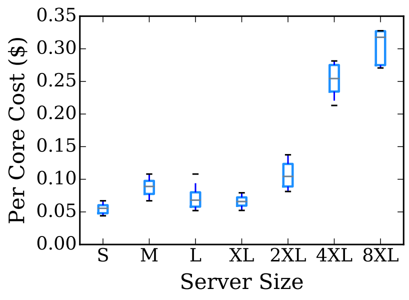
Step 3: Coordinating with front-end tier. DBScale coordinates with front-end tier in order to enforce good end-to-end client performance through and especially after provisioning process. First, DBScale learns about the current configuration of front-end tier and any upcoming provisioning activities and uses such information to refine its provisioning policy. For example, if front-end tier decides to place a web replica at a new cloud location, DBScale needs to evaluate this decision in combination with its plan to make sure no SLAs are violated. Next, DBScale informs front-end tier about its provisioning plan and configuration such as whether snapshots are pre-copied. During provisioning, DBScale periodically updates front-end tier about its progress so as to synchronize provisioning completion time.

Step 4: How to provision database replicas? DBScale starts provisioning database replicas by first making a hot backup from an existing up-to-date replica using XtraBackup [132], a hot backup tool for MySQL database. If a full backup was created and archived in the destination clouds already, DBScale will only request for an incremental backups that contains updated data. The hot backup tool produces a consistent point-in-time snapshot of the database without interrupting normal database processing at that replica. DBScale then transfers the snapshot to a DBaaS cloud server that will host the new replica. In the case where a new cloud location is chosen, the snapshot is transferred over WAN to this site. After transferring, DBScale uses the hot backup tool's crash recovery feature to load the snapshot into the database. DBScale supports two different modes to bring database replicas up-to-date, an offline approach and an online approach. If any updates are made to the database replica in the meantime, DBScale uses an offline approach that acquires a read-lock on current replicas, fetches write queries and applies them to the newly provisioned replica(s). An alternate online approach is to make the new replica a slave and have it receive updates from an existing master (while this approach is suitable for master-salve configurations within a data center, doing so will incur higher overheads for master-slave configuration that run over WAN).

EBS Storage Type(GB-month)	Max($)	Min($)	Std. Dev. ($)
General purpose SSD	0.19	0.10	0.03
Provisioned IOPS SSD	0.24	0.13	0.03
Throughput optimized HDD	0.09	0.05	0.01
Cold HDD	0.05	0.03	0.01
Snapshot (to S3)	0.13	0.10	0.01

EC2 Data Transfer (GB)	Max($)	Min($)	Std. Dev. ($)
Outbound Internet Traffics	0.25	0.09	0.05

(a) Storage and data transfer Costs.



(b) Server costs.

Figure 5.5: We use Amazon's distributed clouds as a case study, and analyze price differences exhibiting in different cloud locations for storage, data transferring and servers. For example, one can save up to 24% in renting *4xlarge* server by choosing a cheaper data center.

5.7 DBScale Implementation

5.7.1 Implementation Overview

We have designed and implemented DBScale as a middleware for managing geo-elasticity in DBaaS cloud. Our DBaaS cloud is built on top of Amazon EC2's distributed clouds that have tens of cloud locations across the globe. To construct our DBaaS cloud, we first lease servers from distributed IaaS cloud and then run database replicas on those IaaS servers. Our prototype is based on the transactional MySQL database platform—that is, database tenants are provided as MySQL databases through the DBaaS cloud.

DBScale is implemented in `Python` and consists of two logical components. The high level architecture of DBScale is shown in Figure 5.2. For simplicity, we only include the architecture of the central controller and omit showing daemons that run on each server inside DBaaS cloud. The central controller, by default, runs inside Amazon's US-east data center in Virginia. DBScale can also be configured to run its central controller in a new cloud location if it yields smaller communication overhead to all daemons.

Light-weight daemons that run inside all database servers are collecting required data, e.g. workload and resource utilizations, and sending these statistics periodically to the central controller. Specifically, resource usage statistics at the database servers hosting the tenant replicas are measured using `sar` and `iostat` utilities, which yield the database server's CPU and I/O utilization. Note, the frequency of communicating with the central controller depends on whether a reactive threshold is triggered. If so, daemons on these overloaded database servers will immediately notify the central controller. Otherwise, frequencies for each server are chosen uniformly from $[\psi, 10\psi]$ where ψ denotes the default provisioning frequency. If frequency is set at 10ψ, daemons will contact the controller ten times within a provisioning window. This simple approach is used to avoid processing bottleneck in a single controller by effectively spreading processing workload into different time slots.

5.7.2 Implementation Details

Next, we describe implementation details for individual function modules of DBScale's central controller. These modules can be roughly divided into four interconnected pieces based on their functionalities. The interaction details between different modules can be found in Section 5.2.5.

First, *workload monitor* and *performance monitor* modules are responsible for gathering workload statistics and resource utilization from both IaaS and DBaaS clouds. We assume application developers who host databases in our DBaaS cloud expose APIs for DBScale to query the workload statistics of the front-end servers hosted in IaaS cloud. Using these APIs, we assume DBScale at least have access to aggregated requests per second for each cloud location. In an ideal scenario, workload monitor can gather web server logs directly from front-end replicas at each location and aggregates them, as discussed in Section 5.3, to analyze geographic distributions of client workload. DBScale can benefit from such fine granularity data and therefore make more informed provisioning and coordinating decisions.

Moreover, to collect data from DBaaS cloud, both *monitor* modules listen on a well-known port and collect data sent from daemons described above. Database workload information can be extracted from database query logs. Each query entry at least contains a query arrival time stamp, a requester's IP address, and a query execution time. Performance data can also be extracted from `CPU` and `I/O` logs, currently represented as averages over a 5-seconds interval. All the workload and performance statistics are written to a SQLite database sequentially as they are processed. We use `ROWID` for primary key, and create additional three indexed columns, i.e. data center location, server identification number and timestamp to represent each data point's attributes.

Next, *workload forecaster* and *resource provisioner* modules read data from SQLite database to construct a regression model and an ARIMA-based time-series model using Python's StatsModels library, and to parameterize queueing models. These models are maintained and updated automatically based on data from a predefined historical time window. Currently, we set the historical time window for ARIMA model as one day and for the other two models as entire data history. Such choices are only based on our limited experiences with benchmark experiments. It would be ideal to select window sizes for each model based on prediction accuracies. In addition, we also implement our two greedy algorithms in the *provisioner* that select cloud locations either based on network latencies to clients or server resource costs. Users can specify either *latency* or *cost* as a priority for assigning client workload. In all, DBScale combines these models and greedy algorithms to make geo-elastic provisioning plans periodically.

Then, *provisioning engine* takes a provisioning plan that specifies the number of different servers required for each cloud site, and makes adjustment through Amazon EC2's APIs based on existing server resources. After each database server is up and running, we then use database hot backup tools to extract archived snapshots and load them into new replicas. Until now, provisioning engine has successfully prepared new database replicas, but these new database replicas might have obsolete data compared to the up-to-date database servers. The amount of such obsolete data depends on how many write requests have been committed since the snapshot is taken. Before handing them over to *consistency engine*, our *provisioning* module needs to contact one of up-to-date servers to fetch the committed transaction log, and replay these transactions on new replicas. Afterwards, our *consistency* module chooses one of the two currently supported modes, i.e. an offline batch mode and an online master-slave mode, depending on application's specific needs to synchronize new database servers. For example, when used for holding data such as product catalogs that see largely read queries, a simple batched update approach may suffice. Specifically, in the batch mode, we update all database replicas in a batch during the scheduled maintenance window. In the master-slave configuration, replicas are configured as either master or slave and write queries are only sent to the master who then relay to all the slaves. Our *consistency engine* picks databases inside a cloud site that has relatively low propagation delays to all other cloud sites, e.g., a geographical central location, and configure them as master databases.

Last, *geo-elastic coordinator* acts as a bridge between IaaS and DBaaS, and notify both entities about any topology changes due to provisioning. The goal of our *coordinator* is to allow DBScale to make informed provisioning decisions, in conjunction with front-ends. As we show through a case scenario in Section 5.8.3.2, uncoordinated provisioning between IaaS and DBaaS might lead to undesired penalty spikes. To avoid scenarios such as running web and database servers in two cloud sites that are faraway, we incorporate a policy that always enforces provisioning database servers to "follow" the front-ends. And ideally, with cooperation from IaaS cloud, we can also synchronize the finish time of provisioning both tiers by delaying web server provisioning.

In summary, with the central controller taking care of different aspects of geo-elastic provisioning and distributed daemons collecting required data, DBScale thus proactively provisions database servers in accordance to the temporal and spatial client workload as well as front-ends' topology changes in the context of distributed clouds.

5.8 Experimental Evaluation

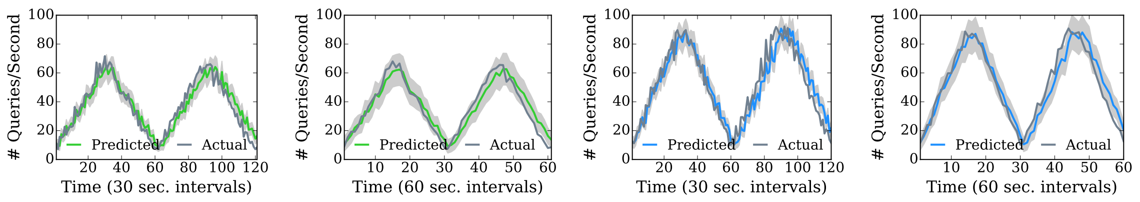
We use end-to-end experiments and empirical-driven simulations to quantify DBScale's performance. First, we evaluate the efficacy of our models and algorithms. Then we demonstrate performance improvement using geo-elasticity and compare DBScale to a caching-based approach. Last, we measure consistency overhead of provisioning database servers using DBScale.

5.8.1 Experimental Setup

Distributed Clouds: We use Amazon EC2's distributed cloud that spans more than ten global data center locations as the infrastructure support. We use EC2 to emulate IaaS clouds and create a DBaaS cloud by running MySQL database engines on rented IaaS servers. We use elastic block stores (EBS) for hosting virtual machine images and database data. Further, we use Amazon's current pricing models (Figure 5.5 lists an exemplary summary) as a basis for our simulations.

Application appliance. We use TPC-W [161], a transactional web benchmark, as our multi-tier application. This java version of TPC-W consists of a front-end that runs on Apache Tomcat and a backend database that runs on MySQL. We create separate appliances, virtual machine images, for its two tiers; unless specified otherwise, each database is configured with 10GB data. Both tiers of TPC-W are assumed to be replicable both within and across EC2 cloud locations. All the front-end VMs were running inside IaaS cloud and the back-end ones are running in DBaaS cloud. DBScale manages the replicas of each database tenant in DBaaS cloud and coordinates with an IaaS cloud manager.

Distributed Clients: We run the emulated clients on PlanetLab nodes that are globally distributed. We choose around one hundred PlanetLab locations from North America, Europe, and Asia that were accessible at the time the experiments were performed. In our experiments, we use three workload mixes that represent different compositions of read and



(a) Browsing: 30 secs. (b) Browsing: 60 secs. (c) Ordering: 30 secs. (d) Ordering: 60 secs.

Figure 5.6: **Comparison of regression-based model predicted rates with empirical measurements.** Predicted and actual query rates over time for the browsing and ordering workload mixes. The shaded areas represent the 95^{th} percentile confidence interval. For both workload types, the prediction accuracy is higher for a larger prediction window.

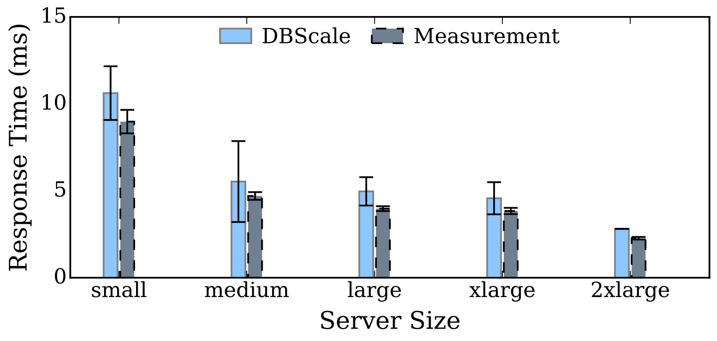


Figure 5.7: **Efficacy of the queueing model.** For each database server size, we compare the empirically measured response times with queueing model predictions.

write requests, i.e., default browsing and ordering workload from TPC-W, and a modified read-only browsing workload.

For each experiment run, we have TPC-W clients running on PlanetLab's nodes from different locations to send HTTP requests to the emulated online bookstore; requests are routed to the closest front-end replicas using a custom DNS-based load redirection. We warm up each replica for five minutes before starting to collect data.

5.8.2 Geo-elastic Models and Algorithms

In this section, we evaluate all the models and algorithms used by DBScale as a basis to provide geo-elasticity.

5.8.2.1 Regression Model Prediction

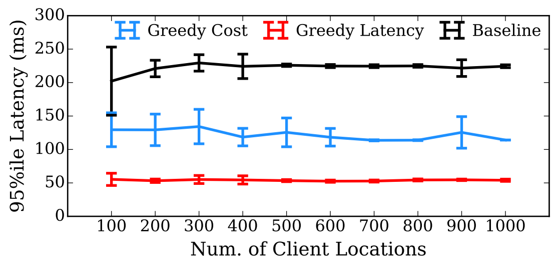
This experiment evaluates the effectiveness of the regression model proposed in Section 5.3.1 to predict database workload for distributed clients. We set up front-end web and back-end database servers in three cloud locations, i.e., California, Virginia and Ireland. For each cloud location, we then run TPC-W clients in a PlanetLab node that has a small network distance in terms of round trip time. Clients only send their requests to the pre-configured web and database pairs during the entire experiment that lasts for one hour. We control the number of concurrent clients from the same geographic location, therefore workload intensity, by assigning different starting time and transaction time to each client.

We repeat the process five times and use the data from first four runs to train the regression model and obtain model parameter (α, β) for each client location. We then use the front-end requests from the fifth run as regression model input and calculate the predicted database query rates and reconstruct ground truth of database queries for each client location.

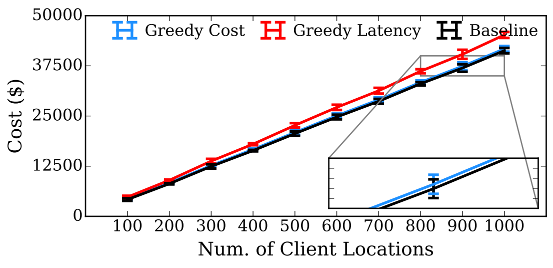
We plot the *Predicted* and *Actual* results for Pennsylvania clients with different workload mixes and prediction intervals in Figure 5.6. We observe that our regression model can make very reasonable predictions for both browsing (read-intensive) and ordering (read-write mix) workload with a mean error of 7.35%. Specifically, as we increase the prediction interval from 30 seconds to 60 seconds, our regression model makes better and smoother decisions.

5.8.2.2 Queueing Model Prediction

Next, we evaluate the queueing model from Section 5.4.1 to estimate database server response times (which in turns yields server capacity). We configure both front-end and back-end servers in Virginia data center and start a number of independent clients based on different server sizes. The independence between clients make sure query arrival to database server satisfies poisson process. And we carefully control the number of clients



(a) 95th percentile network latency.



(b) Operation cost.

Figure 5.8: **Client workload impact on workload mapping algorithms.** As the number of client locations increases, cost-first greedy algorithm can achieve as low as 52.6 ms for 95th percentile network latency while save up to 10.6% in operation cost.

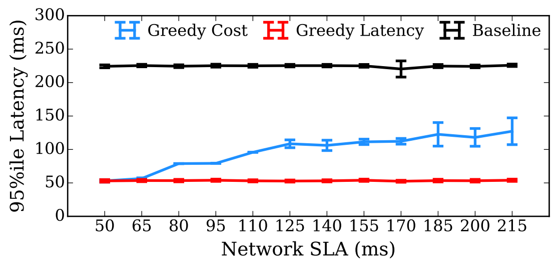
because we don't want to saturate the servers doing the test. For simplicity, we use a total of 100 clients for all server types that are under evaluation.

We collect both query logs and resource utilization logs from database servers for each experiment run that lasts half an hour. We then calculate the database server response time using Equation (5.5) and also obtain actual response time measurement by processing all the logs. We repeat the same process five times for each server type and calculate the average DBScale estimation, the mean measurement value, and the 95th confidence interval across across all five runs.

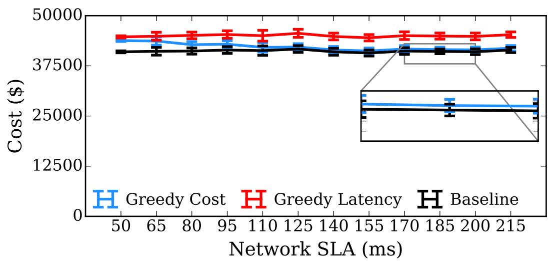
We plot these comparisons for different server sizes in Figure 5.7. We see that empirically measured response times lie within the 95% confidence intervals of the model predictions, indicating a good prediction. Only in case of 2xlarge EC2 servers, where the empirical value is outside the 95% CI, we see a prediction error of 19%. In all cases, the model predictions are overestimates of the response times, indicating that the computed capacity will be conservative from a provisioning perspective.

5.8.2.3 Geo-distributed Workload Mapping Decisions.

Last, we evaluate our two greedy algorithms' performance in tail network latency and daily operation cost by comparing them with a baseline algorithm. The baseline algorithm



(a) 95th percentile network latency.



(b) Operation cost.

Figure 5.9: **Network SLA impact on workload mapping algorithms.** Because cost-first algorithm adjusts workload mapping decision by using Network SLA as a constraint, we observe the 95th percentile network latency increases accordingly. The cost differences between two algorithms are stable around 7.8% after network SLA is set to be larger than 125 ms.

simply selects the cheapest cloud location and maps all client workload to that *single* location. Therefore, this baseline yields the lowest operation cost and a high tail network latency—but it does not provide any guarantee in satisfying T^N_{SLA}.

Our simulation is based on the current Amazon's distributed clouds that consist of twelve global cloud locations [11]. We collect empirical network latency traces by measuring the network distances between all PlanetLab nodes and Amazon clouds. This yields a latency matrix of size $(100, 12)$. We choose one particular server *4x large* server with optimized I/O and determine its capacity empirically. We use the above mentioned data as a basis for constructing simulation input.

Specifically, for each simulation run, we configure the corresponding T^N_{SLA} and the number of client locations (as a proxy for client workload). We construct the set of client locations by uniformly selecting from PlanetLab nodes with replacement. We then generate a normalized workload (compared to λ^c) associated with each client location by drawing a value from a uniform distribution with range $[0, 1]$. We vary simulation configurations and repeat each configuration for ten times and collect the network latency distribution and operation cost as defined in Section 5.5.

In Figure 5.8, we study how our two greedy algorithms behave with an increasing client workload and a fixed T^N_{SLA} of 200 ms. Both greedy algorithms produce up to 172 ms reduction in 95^{th} percentile network latency. Latency-first greedy algorithm achieves optimal tail latency up to 80 ms smaller than those of cost-first greedy algorithm. In addition, cost-first greedy algorithm achieves almost identical operation costs as with the baseline algorithm and an up to 10.6% saving when compared to latency-first greedy algorithm. This is mainly because cost-first algorithm can try to utilize eligible cloud location that is cheapest.

In Figure 5.9, we compare the performance of all three algorithms under different network SLA specification for assigning workload of one thousand client locations. We observe cost-first greedy algorithm behaves similarly to latency-first algorithm with a smaller T^N_{SLA} value, as shown in Figure 5.9a. This is because the number of eligible cloud locations for each client location is determined by T^N_{SLA}—and when T^N_{SLA} is small enough, cost-first algorithm will pick the same cloud location as latency-first algorithm. Therefore, the tail latency performance of two greedy algorithms diverge with more relaxed network SLA values.

In summary, we show that cost-first greedy algorithm leads to higher cost savings when both client workload and network SLA increase while is able to keep 95^{th} percentile network latency within T^N_{SLA} specification. Because cost savings come from the ability to aggregate client workload and having access to more eligible cheaper clouds, we can expect higher savings in a more distributed cloud environment.

Conclusion: We empirically evaluate our regression-based workload prediction model, our queueing-based capacity model and our workload mapping greedy algorithms using current distributed clouds and geographically-distributed clients. We show that our workload prediction only incurs a mean error of 7.35% and our queueing model produces reasonable overestimation when compared to empirical measurement. Further, our simulations demonstrate that our greedy algorithms can effectively make trade-offs between tail network latency and operation costs when compared to the baseline algorithm.

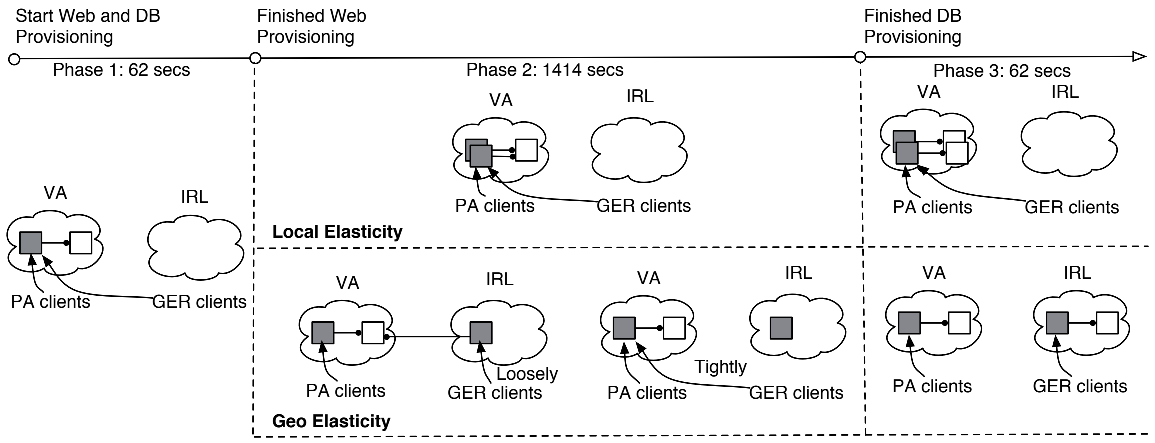
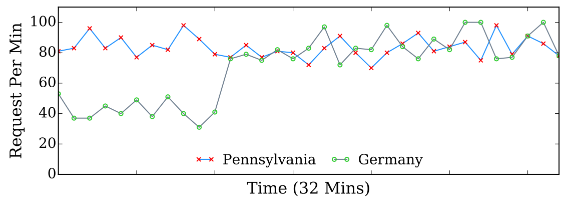


Figure 5.10: **Illustration of elasticity mechanisms and provisioning policies.** We conduct an end-to-end experiment with different phases to demonstrate a policy-driven geo-elasticity is the most effective provisioning approach.



(a) Client request rates.

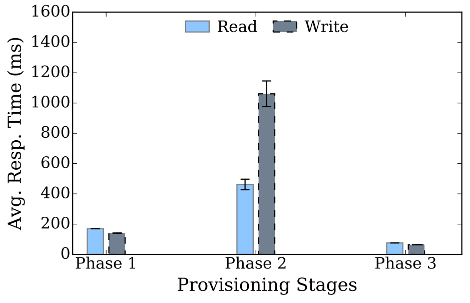
	Local Elasticity	Geo Elasticity
Mean(ms)	169	76
Std. Dev.(ms)	20	10
Conf. Int. (ms)	[167.07, 170.93]	[75.34 , 76.66]

(b) Response time statistics for local and geo elasticity.

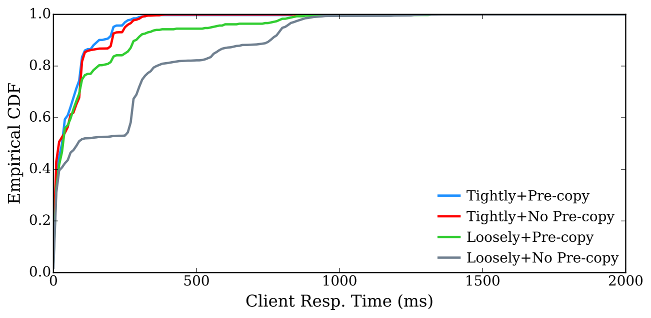
Figure 5.11: **Performance benefits for Geo-elasticity provisioning.** Geo-elasticity provides lower mean response times due to lower client-server network latencies.

5.8.3 Benefits of Database Geo-elasticity

In this section, we design a case study that demonstrates the potential performance improvement with geo-elastic provisioning. In addition, we compare the client performance of running geo-elasticity with four different policies. Figure 5.10 depicts our setup that involves two client locations, Pennsylvania and Germany, and two data center locations, Virginia and Ireland. Dark boxes represent web servers and light boxes represent database servers. For example, in the leftmost column, we have both clients from Pennsylvania and Germany make requests to a web server running inside Virginia's data center, who then fetches data from database running in the same data center. The top time axis shows the



(a) Mean Germany client response time.



(b) CDF of client response time.

Figure 5.12: **Performance benefits of tightly coupled provisioning and pre-copying.** A tightly coupled policy improves the 95^{th} percentile of response time from 810 ms to 250 ms when compared to the loosely coupled policy. Pre-copying further improves 95^{th} percentile of response time to 210 ms.

progress of different provisioning events with both local elasticity and geo-elasticity. The entire provisioning activity is broken down into three different phases, i.e., starting provisioning, finishing provisioning web server, and finishing provisioning database server.

5.8.3.1 Performance Improvement with Geo-elasticity

We first compare the end-to-end performance improvement brought by provisioning in a geo-elastic way when compared to local elasticity. Figure 5.11a shows the average client requests for the entire experiment duration. We deliberately specify a very low server response time SLA—that is, at the end of the first provisioning epoch, our provisioning algorithms will scale up existing server resources. Note that, a choice of low SLA also eliminates potential performance deterioration caused by an overloaded server, making it easy to reason about the performance improvement.

To handle such a workload, local elasticity provisions additional servers locally, within the same Virginia data center location; while geo elasticity can provision capacity at any suitable cloud location, i.e. Ireland cloud. The corresponding provisioning results are shown in Phase 3 of Figure 5.10. We record per request end-to-end response time for clients from both locations.

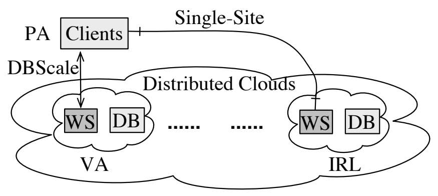
Table 5.11b shows that geo-elasticity reduces average response time from 169 ms to 76 ms, an 55.03% improvement, when compared to local elasticity. The reason underlying the improvement is clients from Germany can now be fully served in the nearby Ireland cloud, instead of in the further Virginia data center. Therefore, all client requests from Germany are at least seeing an 70 ms network round trip time reduction, from 100.3 ms to 29.7 ms.

5.8.3.2 Policy-based Performance Improvement

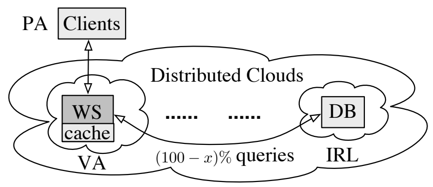
Next, we scrutinize the end-to-end response time variations experienced by Germany clients when performing geo-elasticity with loosely-coupled policy. Figure 5.12a shows the response time spikes that Germany clients experience when provisioning activity is not synchronized between IaaS and DBaaS clouds—loosely-coupled provisioning. During phase two, all client requests from Germany are first sent to the newly provisioned web server in Virginia who then makes query requests to the database server in Ireland. As a result, requests that need to visit back-end servers multiple times to fetch desired data will experience 2x to 6x increase.

To reduce performance impact when provisioning for dependent resources, for example front-end and backend servers, we can either reduce provisioning duration external to clients or synchronize provisioning activities among resources. In the context of this case study, we can dramatically shorten database provisioning time from tens of minutes to a couple of minutes by pre-copying required data in advance; and we can enforce front-end servers to be configured to database servers within the same data center—data centers with acceptable network latency.

We plot client response time CDF obtained using four different policy combinations in Figure 5.12. Tightly-coupled provisioning, with or without pre-copying, outperform loosely-coupled provisioning with an up to 74% improvement for 95^{th} percentile. When pre-copying the database snapshot to the destination Ireland cloud in advance, we only need to copy a delta of 100 MB data during actual database provisioning. As a result, we drasti-

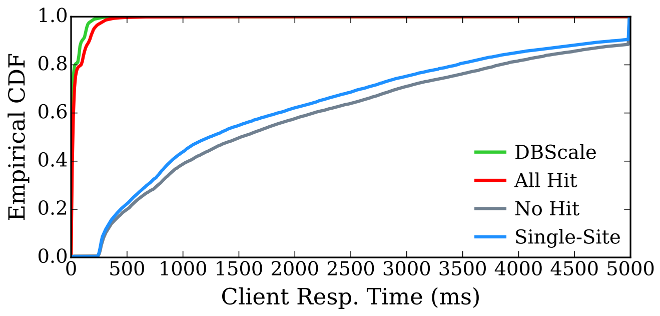


(a) Baseline setups: DBScale and single-site elasticity.

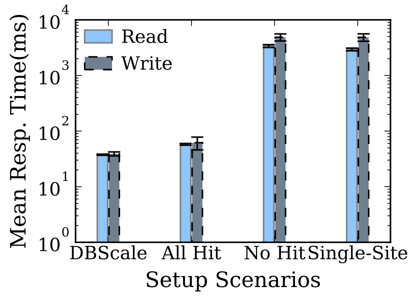


(b) Set up for $x\%$ cache hit rate.

Figure 5.13: **Experimental setup for comparing DBScale to a caching approach.** The front-end tier is replicated and configured with an 1 GB in-memory cache in the caching approach. We use the same web and database server types for all the experiments.



(a) CDF of response time.

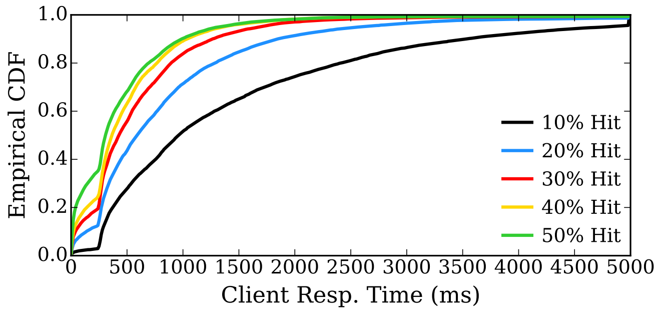


(b) Average response time.

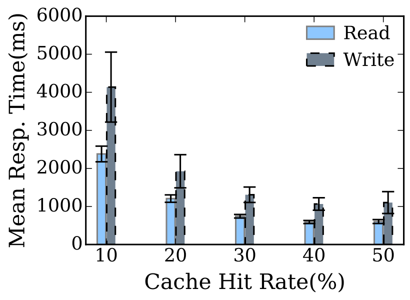
Figure 5.14: **CDF comparison of end-user response time of four different scenarios.** A caching approach with 100% hit rate has comparable performance to DBScale while a 0% hit rate causes performance to be similar to local single-site elasticity.

cally reduce duration of phase two and in turns cut down the number of requests that need to make transcontinental requests from Europe to USA. With pre-copying enabled, loosely-coupled provisioning yields up to 31% improvement for 95^{th} percentile when compared without pre-copying.

Conclusion: Geo-elasticity provisioning effectively reduces the mean end-to-end response time, thus improving performance for all clients. Tight-coupled and pre-copy policies are effective in reducing response time spikes during provisioning.



(a) CDF of response time.



(b) Average response time v.s. hit rate.

Figure 5.15: **CDF comparison of end-user response time with increasing hit rate.** As the hit rate increases from 10% to 50%, the 95^{th} percentile response time improves by 72.18%, from 4780 ms to 1330 ms.

5.8.4 Comparing DBScale to a Caching Approach

In this section, we compare DBScale's performance to that of a caching-based approach. In our caching approach, the database server runs in a single *centralized* location while web servers are replicated in various geographic locations and use Memcached [112], or any other in-memory cache, to store recent query results.

We modified TPC-W so that read requests for data are sent to in-memory cache, who will then query remote database servers during cache miss. We use a small database of 512 MB and allocate 1 GB RAM for the in-memory cache. We warm up the in-memory cache using a modified read-only browsing workload mix. By warming up cache using known workload mix, we are able to control the desired cache hit rates. For example, if we set the cache hit rate to be 10%, each request will trigger a cache miss with a probability of 0.1 and be served directly from cache with 0.9 chance. The setup for this case study is illustrated in Figure 5.13. We run the default browsing workload mix (95% reads and 5% writes) from Pennsylvania clients for different setup, and collect client end-to-end response time for all requests.

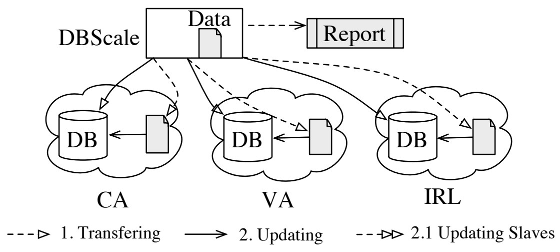
5.8.4.1 In-Memory Cache v.s. DBScale

We plot CDFs of end-user response time for four different scenarios, i.e., DBScale, two extreme caching scenarios and single-site local elasticity, in Figure 5.14. An *All hit* cache represents the best case scenario where all data requests can be satisfied from the local cache, while an *No hit* cache corresponds to the worse case scenario in which data are fetched from remote database across WAN. We show that DBScale has comparable performance with a perfect cache because in both cases, front-end servers are able to fetch data locally, either from a local cache or a local database. In addition, we also demonstrate that both *Single-site* and a complete cold cache perform poorly because all requests for data have to go to the further centralized database, either directly or from within the cache. Besides significant improvement of 95^{th} percentile from 4.9 seconds to 140 ms, DBScale also reduces the mean response time, especially for write requests as shown in Figure 5.14b (log-scale y axis). In summary, DBScale behaves akin to the scenario where all requests are served from local cache, while single-site local elasticity is similar to a complete cache miss.

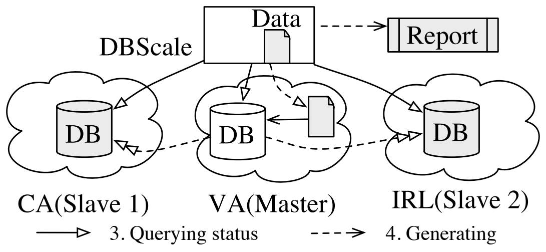
5.8.4.2 Impact of Cache Hit Rate

Next, we study performance impact with an increasing hit rate from 10% to 50%. In Figure 5.15a, we plot the CDF of client response time and show that 95^{th} percentile decreases from 478 ms to 133 ms when more requests are served from local cache. This is because the percentage of requests that avoid WAN latencies decreases as the hit rate increases from 10% to 50%. In addition, as shown in Figure 5.15b, the benefits of caching only accumulate for predominantly read-intensive workloads. Requests that trigger update queries still need to visit the remote database server and therefore experience large network latency, resulting in poor average response time.

Conclusion: we demonstrate that a caching-based approach might provide comparable performance to DBScale, with high hit rate and a low fraction of write workload. How-



(a) Batched updates setup.



(b) Online master-slave setup.

Figure 5.16: **Experimental setup for updating databases in different locations.** Updates are first copied to all the cloud locations or the master database's location. Then we either take the databases offline for batched update or configure a master-slave topology for online synchronization.

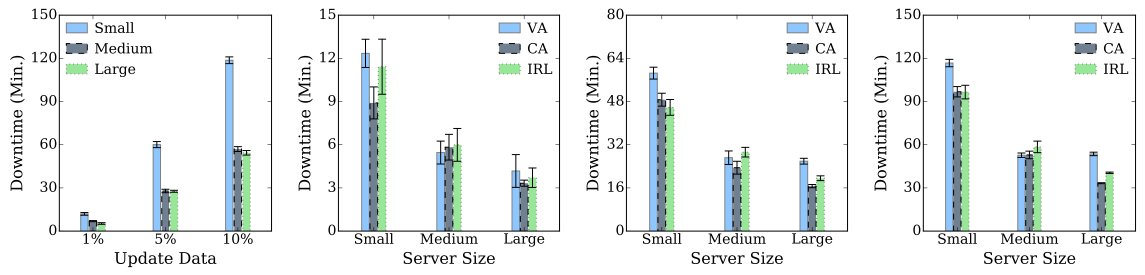
ever, in practice, the actual hit rate depends on a number of factors, such as the skew in query popularity distribution, cache size and replacement algorithms. DBScale does not depend on these factors and could yield good performance always (at the cost of needing consistency maintenance among replicas).

5.8.5 Consistency Maintenance Overheads

In our final set of experiments, we evaluate the overheads of maintaining consistency of database replicas that spread across transcontinental cloud locations. We compare two common approaches, i.e. batched and online updates using MySQL master-slave configuration, for achieving database consistency. The experiment setup is illustrated in Figure 5.16 for both batch and online scenarios. And we use TPC-W web application benchmark that is loaded with 13.43GB database.

5.8.5.1 Batched Updates Overhead

We measure the overhead of applying a varying amount of updates in batch mode during offline maintenance windows; the three database replicas are each hosted separately in Virginia, California and Ireland data centers, as shown in Figure 5.16a. We measure the latency to apply updates at all replicas and restart all servers. Figure 5.17a shows the mean downtime for applying varying amount of updates across five runs along with the 95% con-



(a) Maintenance downtime v.s. data.

(b) Updating 1 % of database.

(c) Updating 5 % of database.

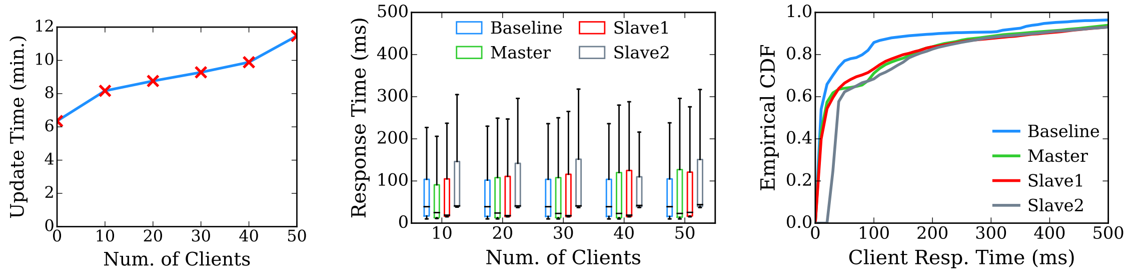
(d) Updating 10% of database.

Figure 5.17: **Batched updates Overheads.** The maintenance downtime (ranging from a few minutes to hours) due to batched updates is impacted by the amount of the data that need to be updated, and the server capacity, i.e. server size and the cloud location.

fidence intervals. The figure shows that it takes 12.52 minutes to update 1% of database data on a small server and as much as 60 minutes to update 10% of the database on a medium server. In general, the downtime is cut in half as we move from a small server to medium or large servers. We observe the capacity differences and slightly different downtimes even for servers of same types in different cloud locations as shown in Figure 5.17d. These results show that, barring under-sized small severs, batched updates can be a feasible option during maintenance windows, which themselves last for a few hours.

5.8.5.2 Online Master-slave Maintenance.

Finally, we study the overheads of using master-slave topology for executing database updates by measuring (i) the impact on maintenance time and (ii) the impact on foreground requests and client response time. As shown in Figure 5.16b, we configure the database in Virginia as the master database and the other two as *Slave 1* and *Slave 2* in California and Ireland respectively. Read queries are sent to the databases in the vicinity while write queries are sent to master database. We record the time to update 1% of database data as well as the end-users response time using this topology; all the database servers run on medium-sized servers.



(a) Maintenance v.s. Clients. (b) Response time distribution. (c) Response time CDF.

Figure 5.18: **Impacts of online master-slave on update time and response time distribution.** As the workload of end-users increase, we observe a corresponding increase in the update latency. Also, response time CDF of both master and slaves behaves similarly to the *no-writes* baseline scenario.

Figure 5.18a shows that it takes 6.35 minutes to update 1% data and as the front-end workload increase, the online update time increase too. Our observation suggests that in order to reduce the length of update time, i.e. the impact duration on end-users, we could adjust the database server size based on end-users' workload during the online maintenance phase. To demonstrate the online maintenance activities' impacts on the end-users' response time, in Figure 5.18b, we compare the client response time distribution of master and slaves compared to baseline *no writes* scenario for different levels of workload intensity. We observe no obvious impact on client response time distribution of master-slave updates approaches at different workload intensity, making it a feasible solution as well. Specifically, in Figure 5.18c, we show that the 95^{th} percentile response time increases from 400 ms to 560 ms for master and to 595 ms for slaves for a 50 clients workload at each location.

Conclusion: we measure performance overhead of two consistency models provided in DBScale in varying scenarios. These measurements can serve as a guideline for configuring consistency for different application needs. We show the batch update time varies according to the amount of new data and server capacity. In master-slave mode, we show *overhead increases with the client workload, with an up to 40% increase at* 95^{th} *percentile response time at master.*

5.9 Related Work

Distributed cloud platforms have became a popular paradigm for hosting web applications. Their pay-as-you-go pricing model and flexible resource allocation make them well-suited for hosting applications with dynamic workloads [15, 23]. When physical resources are shared among multiple VMs, it becomes challenging to accurately model the resource usages for each VM [36, 95, 166] mainly due to interference of co-located VMs [37,88,120,174]. The problem becomes more noticeable for applications with bursty workload characteristics [115]. To overcome this hurdle, recent efforts have attempted to mitigate the impact of interference either by combining the VMs workloads [114] or by employing a novel performance prediction model that is capable of dealing with bursty workloads or even flash crowds [34]. In our work, we combine our empirically measured distributed front-end workload and a regression-based model to predict the spatial and temporal variations in the backend database workload.

Queueing-based models have been used extensively to model cloud-based applications [118, 162, 164], but most have focused on front-end servers . To parameterize those proposed models, it often requires to perform empirical measurements on real system with predefined workload. However, due to potential costs of intrusive measurements and the volatility of workload mix, an alternative regression model [173] was proposed to approximate the CPU demand with different transaction mixes so as to effectively model complex live systems with very few parameters. In our work, we focus on dynamic provisioning in distributed database cloud and model the database server as a two-node queueing network with feedback to track both CPU and I/O utilization.

As more database management tasks [48, 136] are offloaded to the cloud, researchers have begun to focus on adaptive and dynamic provisioning of database servers based on

SLA [35, 117, 128, 143, 168]. These efforts on database provisioning include using models and tools to predict resource utilization and performance for OLTP databases [47, 117], cloning techniques to spawn database replicas [35, 124], live migration techniques to horizontally scale up database server [52], middleware approach to coordinate cloud-hosted applications and databases without violating SLA [143] and utilizing distributed cloud platforms for performance-aware data replication [6, 105, 128, 134]. Our focus here is on geo-elasticity, which is less well studied, and we propose the DBScale framework to handle geo-elasticity for cloud hosted databases.

5.10 DBScale Summary

We proposed a new dynamic provisioning algorithm, called geo-elasticity, for DBaaS clouds to handle both temporal and spatial workload dynamics. Our work is motived by the emergence of distributed clouds, the popularity of geographically distributed applications, and the paradigm for applications to host their backend tiers in DBaaS clouds. To achieve geo-elasticity, we presented a regression-based prediction model that infers geographical workload distribution for database tier, and a two-node open queueing network model that estimates database capacity. Further, we proposed the geo-elastic algorithm that combines both models and two greedy workload assignment algorithms for provisioning database servers in distributed clouds.

We implemented a prototype called DBScale as a middleware based on Amazon distributed clouds and conducted comprehensive evaluations to quantify DBScale's performance. Specifically, we performed both end-to-end experiments as well as benchmark experiments to demonstrate the efficacy of our models, algorithms and DBScale as a whole. Our results showed up to a 66% improvement in response time when compared to local elasticity approaches.

CHAPTER 6
CONCLUSION

6.1 Thesis Summary

This thesis looked at new challenges associated with resource management for different cloud-based applications in distributed clouds. I have proposed a set of techniques that dynamically provision cloud resources to meet performance expectations while keeping a reasonable cost budget. Our approaches include both new model-driven techniques that dissect application behaviors, server capacities and their relationships, and system-oriented mechanisms that enable new resource management abstractions through both high-level and low-level engineering. Our contributions are demonstrated through three key systems that make performance and cost trade-offs automatically and dynamically.

First, I proposed an intelligent system to optimize placement and performance of applications in distributed clouds. I implemented a prototype on top of a nested hypervisor to bypass limited management capabilities exposed by native cloud hypervisors. Our prototype, equipped with a black-box fingerprinting technique, can effectively discriminate between latency-sensitive and insensitive desktop VMs and judiciously move only those that will benefit the most from the migration. Our evaluation demonstrates an up to 90% improvement of VNC's refresh rate for desktop VMs with video activity after migration.

Next, I developed a new provisioning technique called geo-elasticity to dynamically provision server resources at *any* geographic location whenever needed. Geo-elasticity is essential for taking full advantage of server resources that are distributed in today's cloud platform and is beneficial for improving end-to-end client response time of geographically distributed applications. This proposed work treats VMs as black-box and leverages a queueing-based model for estimating virtualized server's capacity without prior knowledge of applications running within. Our evaluation, using Amazon's distributed clouds, shows up to 40% improvement in the 95th percentile response time when compared to traditional elasticity techniques.

Last, I extended the geo-elasticity provisioning technique to database clouds. In this work, I relaxed the previous black-box assumption and presented an application-specific performance model by leveraging our prior knowledge of CPU and I/O intensive database workload. We introduced a regression-based workload prediction algorithm to handle workload obscuration caused by front end servers. Our evaluation shows an up to 55% improvement in mean response time when compared to local elasticity. In addition, we proposed greedy algorithms that help reducing provisioning costs by up to 10.6% for reasonable workload intensity.

In summary, this thesis has explored how to efficiently manage resources in distributed clouds for better performance and lower cost. We proposed new approaches to dynamically provision resources for different cloud-based applications. These approaches were implemented into three key systems that leverage existing models and advancement in virtualization techniques. With the help of our approaches, we can perform dynamic provisioning more efficiently for applications with geographically distributed clients within heterogeneous distributed cloud platforms.

6.2 Future Work

In VMShadow, we focus on using migration techniques that are made possible by para virtualization and nested virtualization to improve end-to-end user performance. Those VM-machine based approaches are ideal in the context of providing performance isolation but inevitably impose management overhead that is proportional to VM size. This overhead, in part, comes from the need to transfer data that is not essential for reconstructing application states. It will be interesting to explore different system-level techniques such as cloning execution or OS-level virtualization technique as light-weight alternatives for managing resources in distributed clouds.

I proposed to provide the ability to provision server resources in any cloud locations—geo-elasticity—for both IaaS and DBaaS distributed clouds. However, the overhead to perform geo-elasticity is inherently high because data has to be transferred across WAN. In other words, the benefits of geo-elasticity are highly dependent on meticulous planning. This planning could be about provisioning frequency or how to manage snapshot data. Future work is required to come up with provisioning policies that are tailored to provisioning in a distributed settings. In addition, it will be useful to extend benefits of geo-elasticity to the reactive provisioning scenario where separate provisioning entities can react to workload changes agilely.

Distributed clouds have become a popular platform for monitoring, aggregating and processing a large amount of data. Data processing applications have different performance goal compared to previously targeted interactive applications. These "big data" applications are less sensitive to network latency and might not benefit at all from moving data to a cloud location that is closest to human operators. Future work is needed to explore the fundamental differences between data processing and interactive applications, and to search for novel approaches to managing resources for data processing applications.

APPENDIX 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 undergoing heavy traffic, that is server utilization ρ is approaching 1 with $\rho = \bar{x}\lambda$ based on Little's Law .

To do so, we first use Kingman's theorem for G/G/1 queue under heavy traffic to represent $\alpha_T(95)$. Kingman's theorem states that the waiting time W is an exponential distribution with mean $E[W]$ as shown in Equation A.1 and A.2.

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

$$\begin{aligned} F_W(w) &= P(W \le w) \\ &= 1 - \exp^{-\frac{1}{E[W]}w} \end{aligned} \tag{A.2}$$

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:

$$P(T \leq \alpha_T(95)) = 0.95 \tag{A.3}$$

$$T = W + X \tag{A.4}$$

Note that all x from service time distribution X has to satisfy Equation A.4, we then have $T \approx W + \bar{x}$. Combining Equation A.2 to A.4, we have an equation of $\alpha_T(95)$ in terms of $E[W]$ and $\bar{x}$ in Equation A.5.

$$1 - \exp^{-\frac{1}{E[W]}(\alpha_T(95) - \bar{x})} = 0.95 \tag{A.5}$$

Applying ln operation to both side, we can express $\alpha_T(95)$ with all known parameters as shown in Equation A.6.

$$\alpha_T(95) = \frac{3\lambda(\sigma_a^2 + \sigma_b^2) + 2(1-\rho)\bar{x}}{2(1-\rho)} \tag{A.6}$$

To satisfy SLA y, we just need to have $\alpha_T(95) < y$. By substituting Equation A.6 to the SLA constrain, we obtain the upper bound of request rate λ as:

$$\lambda < \left[\bar{x} + \frac{3({\sigma_a}^2 + {\sigma_b}^2)}{2(y - \bar{x})}\right]^{-1} \tag{A.7}$$

APPENDIX B

GEO-ELASTIC PROVISIONING WITH QUADRATIC PROGRAMMING

Both χ^{lat} and χ^{cos}, obtained through our greedy algorithms in Section 5.5.1, are binary matrices. That is, χ_{ij} is binary—either all or none of workload from client location i is assigned to cloud location j. In this section, we formulate the workload assignment problem using quadratic programming that allows us to assign client workload from the same location i to multiple cloud locations. This provides us flexibility to *split and pool* client workload and potentially reduce the total number of servers needed.

In a high level, this quadratic formulation reduces cost by finding cheapest cloud locations and aggregating client workload into as fewer servers as possible. In other words, for a subset client workload scenarios, our *cost-first* greedy algorithm will produce the assignments with the same costs.

$$\min \quad \sum_{j=1}^{n} \lceil \omega_j \rceil C_j(\omega_j) \tag{B.1}$$

subject to:

$$0 \le \chi_{ij} \le 1, \quad \forall i \in L^c, \forall j \in L^k \tag{B.2}$$

$$\sum_{j=1}^{n} \chi_{ij} = 1, \quad \forall i \in L^c \tag{B.3}$$

$$\alpha_{95}(T_c^N) \le T_{SLA}^N \tag{B.4}$$

$$\omega_j \le R_j, \quad \forall j \in L^k \tag{B.5}$$

Recall that $C_j(\omega_j)$ (5.11) is a function of ω_j (5.10) and represents the total cost to serve ω_j workload at cloud location j. The objective function (B.1) tries to minimize the total cost for all workload $\boldsymbol{\omega}$, where $\lceil \omega_j \rceil$ is the number of servers needed at cloud location j. Constraints (B.2) and (B.3) makes sure that all client workload is assigned, and constraint (B.4) and (B.5) ensure that network SLA and available resource is not violated.

By solving the above QP formulation, we obtain the assignment matrix $\chi^{QP}_{m\times n}$. Combining with λ^N (5.9), we get

$$\boldsymbol{\omega}^{QP} = \boldsymbol{\lambda}^{\boldsymbol{N}} \boldsymbol{\chi}^{QP}. \tag{B.6}$$

The total cost associated is then $C^{QP} = \sum_{j=1}^{n} \lceil \omega_j^{QP} \rceil C_j(\omega_j^{QP})$. Last, we use $T^N_{QP} = \{(A_{ij}, \lceil \chi_{ij}\lambda_i \rceil) \mid \forall i \in L^c, \forall j \in L^k\}$ to approximate the true network latency distribution. Note T^N_{QP} includes at most mn more data samples compared to actual measurement. But given a reasonable workload $\boldsymbol{\lambda}$, the extra samples will not affect statistics we are interested, i.e. mean and 95^{th} percentile.

BIBLIOGRAPHY

[1] Haversine formula. `https://en.wikipedia.org/wiki/Haversine_formula`.

[2] Host Identity Protocol (HIP). `http://tools.ietf.org/html/rfc5201`.

[3] Identifier-Locator Network Protocol (ILNP). `http://tools.ietf.org/html/rfc6740.txt`.

[4] Locator/ID Separation Protocol (LISP). `http://www.lisp4.net/`.

[5] Abadi, Daniel. Consistency tradeoffs in modern distributed database system design: Cap is only part of the story. *Computer 45*, 2 (Feb. 2012), 37–42.

[6] Agarwal, Sharad, Dunagan, John, Jain, Navendu, Saroiu, Stefan, Wolman, Alec, and Bhogan, Harbinder. Volley: Automated data placement for geo-distributed cloud services. In *NSDI* (2010), pp. 17–32.

[7] Ager, Bernhard, Mhlbauer, Wolfgang, Smaragdakis, Georgios, and Uhlig, Steve. Comparing dns resolvers in the wild. In *IMC '10: Proceedings of the 2010 Internet measurement conference* (Melbourne, Australia, Nov 2010).

[8] Aggarwal, Bhavish, Akella, Aditya, Anand, Ashok, Balachandran, Athula, Chitnis, Pushkar, Muthukrishnan, Chitra, Ramjee, Ramachandran, and Varghese, George. Endre: an end-system redundancy elimination service for enterprises. In *Proceedings of USENIX NSDI* (2010).

[9] Alicherry, Mansoor, and Lakshman, T. V. Network aware resource allocation in distributed clouds. In *INFOCOM* (2012).

[10] AWS Auto Scaling. `https://aws.amazon.com/autoscaling/`, 2013.

[11] Amazon Global Infrastructure. `http://aws.amazon.com/about-aws/global-infrastructure/`.

[12] Amazon Online Shopping. `https://www.amazon.com/`.

[13]

[14] Amir, Yair, Danilov, Claudiu, Miskin-Amir, Michal, Stanton, Jonathan, and Tutu, Ciprian. On the performance of consistent wide-area database replication. Tech. rep., 2003.

[15] Arlitt, Martin F., and Williamson, Carey L. Internet web servers: Workload characterization and performance implications. *IEEE/ACM Trans. Netw.* (1997).

[16] Armbrust, Michael, Fox, Armando, Griffith, Rean, Joseph, Anthony D., Katz, Randy, Konwinski, Andy, Lee, Gunho, Patterson, David, Rabkin, Ariel, Stoica, Ion, and Zaharia, Matei. A view of cloud computing. *Commun. ACM 53*, 4 (Apr. 2010), 50–58.

[17] Ballani, Hitesh, Costa, Paolo, Karagiannis, Thomas, and Rowstron, Ant. Towards predictable datacenter networks. In *Proceedings of the ACM SIGCOMM* (2011).

[18] Baratto, Ricardo A, Kim, Leonard N, and Nieh, Jason. Thinc: a virtual display architecture for thin-client computing. In *ACM SIGOPS Operating Systems Review* (2005), vol. 39, ACM, pp. 277–290.

[19] Barham, Paul, Dragovic, Boris, Fraser, Keir, Hand, Steven, Harris, Tim, Ho, Alex, Neugebauer, Rolf, Pratt, Ian, and Warfield, Andrew. Xen and the art of virtualization. In *ACM SIGOPS Operating Systems Review* (2003), vol. 37, ACM, pp. 164–177.

[20] Barker, Sean, Chi, Yun, Moon, Hyun Jin, Hacigümüş, Hakan, and Shenoy, Prashant. ”Cut me some slack”: latency-aware live migration for databases. In *Proceedings of Conference on Extending Database Technology* (2012).

[21] Ben-Yehuda, Muli, Day, Michael D., Dubitzky, Zvi, Factor, Michael, Har'El, Nadav, Gordon, Abel, Liguori, Anthony, Wasserman, Orit, and Yassour, Ben-Ami. The turtles project: Design and implementation of nested virtualization. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation* (Berkeley, CA, USA, 2010), OSDI'10, USENIX Association, pp. 423–436.

[22] Bi, Jing, Zhu, Zhiliang, Tian, Ruixiong, and Wang, Qingbo. Dynamic provisioning modeling for virtualized multi-tier applications in cloud data center. In *2010 IEEE 3rd International Conference on Cloud Computing* (2010), IEEE, pp. 370–377.

[23] Birke, Robert, Chen, Lydia Y., and Smirni, Evgenia. Usage patterns in multi-tenant data centers: A temporal perspective. In *ICAC* (2012).

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

[25] Bonvin, Nicolas, Papaioannou, Thanasis G, and Aberer, Karl. Autonomic sla-driven provisioning for cloud applications. In *Proceedings of the 2011 11th IEEE/ACM international symposium on cluster, cloud and grid computing* (2011), IEEE Computer Society, pp. 434–443.

[26] Box, George Edward Pelham, and Jenkins, Gwilym. *Time Series Analysis, Forecasting and Control*. 1990.

[27] Boxma, O. J., van der Mei, R. D., Resing, J. A.C., and van Wingerden, K. M. C. Sojourn time approximations in a two-node queueing network. In *ITC* (2005).

[28] Bradford, Robert, Kotsovinos, Evangelos, Feldmann, Anja, and Schiöberg, Harald. Live wide-area migration of virtual machines including local persistent state. In *Proceedings of ACM SIGPLAN/SIGOPS conference on Virtual Execution Environments (VEE)* (2007).

[29] Breitgand, David, Kutiel, Gilad, and Raz, Danny. Cost-aware live migration of services in the cloud. In *Proceedings of Annual Haifa Experimental Systems Conference* (2010).

[30] Bronson, Nathan, Amsden, Zach, Cabrera, George, Chakka, Prasad, Dimov, Peter, Ding, Hui, Ferris, Jack, Giardullo, Anthony, Kulkarni, Sachin, Li, Harry, et al. Tao: Facebooks distributed data store for the social graph. In *Presented as part of the 2013 USENIX Annual Technical Conference (USENIX ATC 13)* (2013), pp. 49–60.

[31] Buyya, Rajkumar, Yeo, Chee Shin, Venugopal, Srikumar, Broberg, James, and Brandic, Ivona. Cloud computing and emerging it platforms: Vision, hype, and reality for delivering computing as the 5th utility. *Future Generation computer systems 25*, 6 (2009), 599–616.

[32] Calheiros, Rodrigo N, Ranjan, Rajiv, and Buyya, Rajkumar. Virtual machine provisioning based on analytical performance and qos in cloud computing environments. In *2011 International Conference on Parallel Processing* (2011), IEEE, pp. 295–304.

[33] Calyam, P., Rajagopalan, S., Selvadhurai, A., Mohan, S., Venkataraman, A., Berryman, A., and Ramnath, R. Leveraging openflow for resource placement of virtual desktop cloud applications. In *2013 IFIP/IEEE International Symposium on Integrated Network Management (IM 2013)* (May 2013), pp. 311–319.

[34] Casale, G., Mi, Ningfang, Cherkasova, L., and Smirni, E. Dealing with burstiness in multi-tier applications: Models and their parameterization. In *TSE* (2012).

[35] Cecchet, Emmanuel, Singh, Rahul, Sharma, Upendra, and Shenoy, Prashant. Dolly: Virtualization-driven database provisioning for the cloud. In *VEE* (2011).

[36] Cherkasova, Ludmila, and Gardner, Rob. Measuring cpu overhead for i/o processing in the xen virtual machine monitor. In *ATEC* (2005).

[37] Chiang, Ron C, and Huang, H Howie. Tracon: interference-aware scheduling for data-intensive applications in virtualized environments. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis* (2011), ACM, p. 47.

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

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

[40] Coffmann, E. G., Gary, M. R., and Johnson, D. S. Approximation algorithms for bin-packing-an updated survey. *Algorithm Design for Computer System Design* (1984), 49–106.

[41] Content Delivery Network. `http://www.akamai.com/html/resources/content-distribution-network.html`, 2013.

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

[43] Cooper, Brian F, Silberstein, Adam, Tam, Erwin, Ramakrishnan, Raghu, and Sears, Russell. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM symposium on Cloud computing* (2010), ACM, pp. 143–154.

[44] Crago, Steve, Dunn, Kyle, Eads, Patrick, Hochstein, Lorin, Kang, Dong-In, Kang, Mikyung, Modium, Devendra, Singh, Karandeep, Suh, Jinwoo, and Walters, John Paul. Heterogeneous cloud computing. In *2011 IEEE International Conference on Cluster Computing* (2011), IEEE, pp. 378–385.

[45] Cuervo, Eduardo, Balasubramanian, Aruna, Cho, Dae-ki, Wolman, Alec, Saroiu, Stefan, Chandra, Ranveer, and Bahl, Paramvir. Maui: making smartphones last longer with code offload. In *Proceedings of the 8th international conference on Mobile systems, applications, and services* (2010), ACM, pp. 49–62.

[46] Cully, Brendan, Lefebvre, Geoffrey, Meyer, Dutch, Feeley, Mike, Hutchinson, Norm, and Warfield, Andrew. Remus: high availability via asynchronous virtual machine replication. In *Proceedings of USENIX NSDI* (2008).

[47] Curino, Carlo, Jones, Evan P.C., Madden, Samuel, and Balakrishnan, Hari. Workload-aware database monitoring and consolidation. In *SIGMOD* (2011).

[48] Curino, Carlo, Jones, Evan PC, Popa, Raluca Ada, Malviya, Nirmesh, Wu, Eugene, Madden, Sam, Balakrishnan, Hari, and Zeldovich, Nickolai. Relational cloud: A database-as-a-service for the cloud.

[49] Das, Sudipto, Nishimura, Shoji, Agrawal, Divyakant, and El Abbadi, Amr. 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.

[50] Dutreilh, Xavier, Moreau, Aurélien, Malenfant, Jacques, Rivierre, Nicolas, and Truck, Isis. From data center resource allocation to control theory and back. In *2010 IEEE 3rd International Conference on Cloud Computing* (2010), IEEE, pp. 410–417.

[51] Elmore, Aaron J., Das, Sudipto, Agrawal, Divyakant, and El Abbadi, Amr. Zephyr: Live migration in shared nothing databases for elastic cloud platforms. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data* (New York, NY, USA, 2011), ACM, pp. 301–312.

[52] Elmore, Aaron J., Das, Sudipto, Agrawal, Divyakant, and El Abbadi, Amr. Zephyr: Live migration in shared nothing databases for elastic cloud platforms. In *SIGMOD* (2011).

[53] Emeakaroha, Vincent C, Netto, Marco AS, Calheiros, Rodrigo N, Brandic, Ivona, Buyya, Rajkumar, and De Rose, César AF. Towards autonomic detection of sla violations in cloud infrastructures. *Future Generation Computer Systems 28*, 7 (2012), 1017–1029.

[54] Expedia travel. `https://www.expedia.com/`.

[55] Fabric documentation. `http://www.fabfile.org/`, 2013.

[56] Facebook newsroom. `http://newsroom.fb.com/company-info/`.

[57] Fernandez, Hector, Pierre, Guillaume, and Kielmann, Thilo. Autoscaling web applications in heterogeneous cloud infrastructures. In *Cloud Engineering (IC2E), 2014 IEEE International Conference on* (2014), IEEE, pp. 195–204.

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

[59] Ford, Bryan, Srisuresh, Pyda, and Kegel, Dan. Peer-to-peer communication across network address translators. In *Proceedings of USENIX Annual Technical Conference* (2005).

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

[61] Garg, Saurabh Kumar, Toosi, Adel Nadjaran, Gopalaiyengar, Srinivasa K, and Buyya, Rajkumar. Sla-based virtual machine management for heterogeneous workloads in a cloud datacenter. *Journal of Network and Computer Applications 45* (2014), 108–120.

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

[63] Google App Engine. `https://appengine.google.com/`.

[64] Google cloud sql. `https://cloud.google.com/sql/docs`.

[65] Guo, Chuanxiong, Lu, Guohan, Wang, Helen J., Yang, Shuang, Kong, Chao, Sun, Peng, Wu, Wenfei, and Zhang, Yongguang. Secondnet: a data center network virtualization architecture with bandwidth guarantees. In *Proceedings of ACM CoNEXT* (2010).

[66] Guo, Chuanxiong, Yuan, Lihua, Xiang, Dong, Dang, Yingnong, Huang, Ray, Maltz, Dave, Liu, Zhaoyi, Wang, Vin, Pang, Bin, Chen, Hua, Lin, Zhi-Wei, and Kurien, Varugis. Pingmesh: A large-scale system for data center network latency measurement and analysis. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication* (New York, NY, USA, 2015), SIGCOMM '15, ACM, pp. 139–152.

[67] Guo, Tian, Gopalakrishnan, Vijay, Ramakrishnan, KK, Shenoy, Prashant, Venkataramani, Arun, and Lee, Seungjoon. Vmshadow: Optimizing the performance of virtual desktops in distributed clouds. In *Proceedings of the 4th annual Symposium on Cloud Computing* (2013), ACM, p. 42.

[68] Guo, Tian, Gopalakrishnan, Vijay, Ramakrishnan, KK, Shenoy, Prashant, Venkataramani, Arun, and Lee, Seungjoon. Vmshadow: optimizing the performance of latency-sensitive virtual desktops in distributed clouds. In *Proceedings of the 5th ACM Multimedia Systems Conference* (2014), ACM, pp. 103–114.

[69] Guo, Tian, Sharma, Upendra, Shenoy, Prashant, Wood, Timothy, and Sahu, Sambit. Cost-aware cloud bursting for enterprise applications. *ACM Transactions on Internet Technology (TOIT) 13*, 3 (2014), 10.

[70] Guo, Tian, Sharma, Upendra, Wood, Timothy, Sahu, Sambit, and Shenoy, Prashant. Seagull: intelligent cloud bursting for enterprise applications. In *Proceedings of USENIX Annual Technical Conference* (2012).

[71] Guo, Tian, and Shenoy, Prashant. Model-driven geo-elasticity in database clouds. In *Autonomic Computing (ICAC), 2015 IEEE International Conference on* (2015).

[72] Guo, Tian, Shenoy, Prashant, and Hacigümüş, Hakan. Geoscale: Providing geo-elasticity in distributed clouds. Tech. Rep. UM-CS-2015-009, School of Computer Science, Univ. of Massachusetts at Amherst, April 2015.

[73] Guo, Tian, Shenoy, Prashant, and Hacigümüş, Hakan. Geoscale: Providing geo-elasticity in distributed clouds. In *International Conference on Cloud Engineering (IC2E)* (2016), IEEE.

[74] Gupta, D., Cherkasova, L., Gardner, R., and Vahdat, A. Enforcing performance isolation across virtual machines in xen. In *Proceedings of the ACM/IFIP/USENIX 7th International Middleware Conference (Middleware'2006), Melbourne, Australia* (November 2006).

[75] Han, Rui, Guo, Li, Ghanem, Moustafa M, and Guo, Yike. Lightweight resource scaling for cloud applications. In *Cluster, Cloud and Grid Computing (CCGrid), 2012 12th IEEE/ACM International Symposium on* (2012), IEEE, pp. 644–651.

[76] Harney, Eric, Goasguen, Sebastien, Martin, Jim, Murphy, Mike, and Westall, Mike. The efficacy of live virtual machine migrations over the internet. In *Proceedings of VTDC* (2007).

[77] He, Keqiang, Fisher, Alexis, Wang, Liang, Gember, Aaron, Akella, Aditya, and Ristenpart, Thomas. Next stop, the cloud: Understanding modern web service deployment in ec2 and azure. In *IMC* (2013).

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

[79] Hiltunen, Matti, Joshi, Kaustubh, Schlichting, Richard, Yamada, Nishio, and Moritsu, Toshiyuki. CloudTops: Latency aware placement of Virtual Desktops institution Distributed Cloud Infrastructures.

[80] Hirofuchi, Takahiro, Ogawa, Hirotaka, Nakada, Hidemoto, Itoh, Satoshi, and Sekiguchi, Satoshi. A live storage migration mechanism over wan for relocatable virtual machine services on clouds. In *Proceedings of the 2009 9th IEEE/ACM International Symposium on Cluster Computing and the Grid* (Washington, DC, USA, 2009), CCGRID '09, IEEE Computer Society, pp. 460–465.

[81] Huang, Qun, and Lee, Patrick PC. An experimental study of cascading performance interference in a virtualized environment. *ACM SIGMETRICS Performance Evaluation Review 40*, 4 (2013), 43–52.

[82] Hung, Chien-Chun, Golubchik, Leana, and Yu, Minlan. Scheduling jobs across geo-distributed datacenters. In *Proceedings of the Sixth ACM Symposium on Cloud Computing* (2015), ACM, pp. 111–124.

[83] Ibrahim, Khaled Z., Hofmeyr, Steven, Iancu, Costin, and Roman, Eric. Optimized pre-copy live migration for memory intensive applications. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis* (New York, NY, USA, 2011), SC '11, ACM, pp. 40:1–40:11.

[84] Ilya, Grigorik. Latency: The new web performance bottleneck. `http://www.igvita.com/2012/07/19/latency-the-new-web-performance-bottleneck/`, 2012.

[85] Iqbal, Waheed, Dailey, Matthew N, Carrera, David, and Janecek, Paul. Adaptive resource provisioning for read intensive multi-tier applications in the cloud. *Future Generation Computer Systems 27*, 6 (2011), 871–879.

[86] Islam, Sadeka, Keung, Jacky, Lee, Kevin, and Liu, Anna. Empirical prediction models for adaptive resource provisioning in the cloud. *Future Generation Computer Systems 28*, 1 (2012), 155–162.

[87] Jin, Hai, Deng, Li, Wu, Song, Shi, Xuanhua, and Pan, Xiaodong. Live virtual machine migration with adaptive, memory compression. In *CLUSTER'09* (2009), pp. 1–10.

[88] Kambadur, Melanie, Moseley, Tipp, Hank, Rick, and Kim, Martha A. Measuring interference between live datacenter applications. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis* (2012), IEEE Computer Society Press, p. 51.

[89] Katz-Bassett, Ethan, John, John P., Krishnamurthy, Arvind, Wetherall, David, Anderson, Thomas, and Chawathe, Yatin. Towards ip geolocation using delay and topology measurements. In *Proceedings of the 6th ACM SIGCOMM conference on Internet measurement* (New York, NY, USA, 2006), IMC '06, ACM, pp. 71–84.

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

[91] Kivity, Avi, Kamay, Yaniv, Laor, Dor, Lublin, Uri, and Liguori, Anthony. kvm: the linux virtual machine monitor. In *Proceedings of the Linux symposium* (2007), vol. 1, pp. 225–230.

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

[93] Kortuem, Gerd, Kawsar, Fahim, Sundramoorthy, Vasughi, and Fitton, Daniel. Smart objects as building blocks for the internet of things. *IEEE Internet Computing 14*, 1 (2010), 44–51.

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

[95] Kundu, S., Rangaswami, R., Dutta, K., and Zhao, Ming. Application performance modeling in a virtualized environment. In *HPCA* (2010).

[96] Labrinidis, Alexandros, and Jagadish, Hosagrahar V. Challenges and opportunities with big data. *Proceedings of the VLDB Endowment 5*, 12 (2012), 2032–2033.

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

[98] Lagar-Cavilla, H. Andrés, Tolia, Niraj, De Lara, Eyal, Satyanarayanan, M., and O'Hallaron, David. Interactive resource-intensive applications made easy. In *Proceedings of the 8th ACM/IFIP/USENIX International Conference on Middleware* (Berlin, Heidelberg, 2007), MIDDLEWARE2007, Springer-Verlag, pp. 143–163.

[99] Lai, Albert M., and Nieh, Jason. On the performance of wide-area thin-client computing. *ACM Trans. Comput. Syst. 24*, 2 (May 2006), 175–209.

[100] Li, Ang, Yang, Xiaowei, Kandula, Srikanth, and Zhang, Ming. Cloudcmp: comparing public cloud providers. In *Proceedings of the 10th ACM SIGCOMM conference on Internet measurement* (2010), ACM, pp. 1–14.

[101] Li, Qiang, Hao, Qinfen, Xiao, Limin, and Li, Zhoujun. Adaptive management of virtualized resources in cloud computing using feedback control. In *2009 First International Conference on Information Science and Engineering* (2009), IEEE, pp. 99–102.

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

[103] Linux container project. `https://linuxcontainers.org/`.

[104] Little, John D. C. A proof for the queuing formula: $l = \lambda w$. *Operations Research 9*, 3 (1961), 383–387.

[105] Liu, Guoxin, Shen, Haiying, and Chandler, Harrison. Selective data replication for online social networks with distributed datacenters. In *Network Protocols (ICNP), 2013 21st IEEE International Conference on* (2013), IEEE, pp. 1–10.

[106] Mainetti, Luca, Patrono, Luigi, and Vilei, Antonio. Evolution of wireless sensor networks towards the internet of things: A survey. In *Software, Telecommunications and Computer Networks (SoftCOM), 2011 19th International Conference on* (2011), IEEE, pp. 1–6.

[107] Malik, Haroon, Hemmati, Hadi, and Hassan, Ahmed E. Automatic detection of performance deviations in the load testing of large scale systems. In *2013 35th International Conference on Software Engineering (ICSE)* (2013), IEEE, pp. 1012–1021.

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

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

[110] Matsunaga, Andréa, and Fortes, José AB. On the use of machine learning to predict the time and resources consumed by applications. In *Proceedings of the 2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing* (2010), IEEE Computer Society, pp. 495–504.

[111] Maximind geoip service. `https://www.maxmind.com/en/home`.

[112] Memcached. `http://memcached.org/`.

[113] Meng, Xiaoqiao, Isci, Canturk, Kephart, Jeffrey, Zhang, Li, Bouillet, Eric, and Pendarakis, Dimitrios. Efficient resource provisioning in compute clouds via vm multiplexing. In *Proceedings of the 7th international conference on Autonomic computing* (2010), ACM, pp. 11–20.

[114] Meng, Xiaoqiao, Isci, Canturk, Kephart, Jeffrey, Zhang, Li, Bouillet, Eric, and Pendarakis, Dimitrios. Efficient resource provisioning in compute clouds via vm multiplexing. In *ICAC* (2010).

[115] Mi, Ningfang, Casale, Giuliano, Cherkasova, Ludmila, and Smirni, Evgenia. Burstiness in multi-tier applications: Symptoms, causes, and new models. In *Middleware* (2008).

[116] Michlmayr, Anton, Rosenberg, Florian, Leitner, Philipp, and Dustdar, Schahram. Comprehensive qos monitoring of web services and event-based sla violation detection. In *Proceedings of the 4th international workshop on middleware for service oriented computing* (2009), ACM, pp. 1–6.

[117] Mozafari, Barzan, Curino, Carlo, and Madden, Samuel. Dbseer: Resource and performance prediction for building a next generation database cloud. In *CIDR* (2013).

[118] N. Bennani, Mohamed, and A. Menasce, Daniel. Resource allocation for autonomic data centers using analytic performance models. In *ICAC* (2005).

[119] Nan, Xiaoming, He, Yifeng, and Guan, Ling. Optimal resource allocation for multimedia cloud based on queuing model. In *Multimedia signal processing (MMSP), 2011 IEEE 13th international workshop on* (2011), IEEE, pp. 1–6.

[120] Nathuji, Ripal, Kansal, Aman, and Ghaffarkhah, Alireza. Q-clouds: Managing performance interference effects for qos-aware clouds. In *EuroSys* (2010).

[121] Nazir, Atif, Raza, Saqib, and Chuah, Chen-Nee. Unveiling facebook: a measurement study of social network based applications. In *Proceedings of the 8th ACM SIGCOMM conference on Internet measurement* (2008), ACM, pp. 43–56.

[122] Nelson, Michael, Lim, Beng-Hong, and Hutchins, Greg. Fast transparent migration for virtual machines. In *Proceedings of the annual conference on USENIX Annual Technical Conference* (2005).

[123] Ng, T.S.E., and Zhang, Hui. Predicting internet network distance with coordinates-based approaches. In *INFOCOM* (2002), pp. 170–179.

[124] Nguyen, Hiep, Shen, Zhiming, Gu, Xiaohui, Subbiah, Sethuraman, and Wilkes, John. Agile: Elastic distributed resource scaling for infrastructure-as-a-service. In *Proc. of the USENIX International Conference on Automated Computing (ICAC13). San Jose, CA* (2013).

[125] Nielsen, Jakob. Response times: The 3 important limits. `http://www.nngroup.com/articles/response-times-3-important-limits/`.

[126] Nordström, Erik, Shue, David, Gopalan, Prem, Kiefer, Robert, Arye, Matvey, Ko, Steven Y., Rexford, Jennifer, and Freedman, Michael J. Serval: an end-host stack for service-centric networking. In *Proceedings of USENIX NSDI* (2012).

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

[128] P N, Shankaranarayanan, Sivakumar, Ashiwan, Rao, Sanjay, and Tawarmalani, Mohit. Performance sensitive replication in geo-distributed cloud datastores. In *DSN* (2014).

[129] Padala, Pradeep, Hou, Kai-Yuan, Shin, Kang G., Zhu, Xiaoyun, Uysal, Mustafa, Wang, Zhikui, Singhal, Sharad, and Merchant, Arif. Automated control of multiple virtualized resources. In *Proceedings of the 4th ACM European conference on Computer systems (EuroSys)* (New York, NY, USA, 2009), ACM, pp. 13–26.

[130] Patiño Martinez, Marta, Jiménez-Peris, Ricardo, Kemme, Bettina, and Alonso, Gustavo. Middle-r: Consistent database replication at the middleware level. *ACM Trans. Comput. Syst. 23*, 4 (2005), 375–423.

[131] Pegus II, Patrick, Varghese, Benoy, Guo, Tian, Irwin, David, Shenoy, Prashant, Mahanti, Anirban, Culbert, James, Goodhue, John, and Hill, Chris. Analyzing the efficiency of a green university data center. In *Proceedings of ACM International Conference on Performance Engineering (ICPE)* (2016).

[132] Percona Xtrabackup. `http://www.percona.com/doc/percona-xtrabackup/2.2/`.

[133] Piao, Jing Tai, and Yan, Jun. A network-aware virtual machine placement and migration approach in cloud computing. In *Proceedings of Grid and Cooperative Computing (GCC 2010)* (2010), pp. 87–92.

[134] Ping, Fan, Li, Xiaohu, McConnell, Christopher, Vabbalareddy, Rohini, and Hwang, Jeong-Hyon. Towards optimal data replication across data centers. In *Distributed Computing Systems Workshops (ICDCSW), 2011 31st International Conference on* (2011), IEEE, pp. 66–71.

[135] Pinheiro, Eduardo, Weber, Wolf-Dietrich, and Barroso, Luiz André. Failure trends in a large disk drive population. In *Proceedings of the 5th USENIX conference on File and Storage Technologies* (Berkeley, CA, USA, 2007), USENIX Association, pp. 2–2.

[136] Popa, Raluca Ada, Redfield, Catherine, Zeldovich, Nickolai, and Balakrishnan, Hari. Cryptdb: protecting confidentiality with encrypted query processing. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles* (2011), ACM, pp. 85–100.

[137] Project voldemort: A distributed database. `http://www.project-voldemort.com/voldemort/`.

[138] Pu, Qifan, Ananthanarayanan, Ganesh, Bodik, Peter, Kandula, Srikanth, Akella, Aditya, Bahl, Paramvir, and Stoica, Ion. Low latency geo-distributed data analytics. *ACM SIGCOMM Computer Communication Review 45*, 4 (2015), 421–434.

[139] Pujol, Josep M., Erramilli, Vijay, Siganos, Georgos, Yang, Xiaoyuan, Laoutaris, Nikos, Chhabra, Parminder, and Rodriguez, Pablo. The little engine(s) that could: Scaling online social networks. In *Proceedings of the ACM SIGCOMM 2010 conference on SIGCOMM* (New York, NY, USA, 2010), ACM, pp. 375–386.

[140] Qian, H., and Andresen, D. Jade: An efficient energy-aware computation offloading system with heterogeneous network interface bonding for ad-hoc networked mobile devices. In *Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD), 2014 15th IEEE/ACIS International Conference on* (June 2014), pp. 1–8.

[141] Rajagopalan, Shriram, Williams, Dan, Jamjoom, Hani, and Warfield, Andrew. Split/merge: System support for elastic execution in virtual middleboxes. In *Proceedings of the 10th USENIX conference on Networked Systems Design and Implementation (NSDI)* (Berkeley, CA, USA, 2013), USENIX Association, pp. 227–240.

[142] Roy, Nilabja, Dubey, Abhishek, and Gokhale, Aniruddha. Efficient autoscaling in the cloud using predictive models for workload forecasting. In *Cloud Computing (CLOUD), 2011 IEEE International Conference on* (2011), IEEE, pp. 500–507.

[143] Sakr, S., and Liu, A. Sla-based and consumer-centric dynamic provisioning for cloud databases. In *CLOUD* (2012).

[144] Salesforce products. `http://www.salesforce.com/products/`.

[145] Satyanarayanan, Mahadev, Bahl, P., Caceres, R., and Davies, N. The case for vm-based cloudlets in mobile computing. *Pervasive Computing, IEEE 8*, 4 (2009), 14–23.

[146] Satyanarayanan, Mahadev, Bahl, Paramvir, Caceres, Ramón, and Davies, Nigel. The case for vm-based cloudlets in mobile computing. *IEEE pervasive Computing 8*, 4 (2009), 14–23.

[147] Shafiq, M Zubair, Ji, Lusheng, Liu, Alex X, Pang, Jeffrey, and Wang, Jia. Characterizing geospatial dynamics of application usage in a 3g cellular data network. In *INFOCOM, 2012 Proceedings IEEE* (2012), IEEE, pp. 1341–1349.

[148] Sharma, Abhishek B., Bhagwan, Ranjita, Choudhury, Monojit, Golubchik, Leana, Govindan, Ramesh, and Voelker, Geoffrey M. Automatic request categorization in internet services. *SIGMETRICS Perform. Eval. Rev. 36*, 2 (2008), 16–25.

[149] Sharma, Prateek, Guo, Tian, He, Xin, Irwin, David, and Shenoy, Prashant. Flint: batch-interactive data-intensive processing on transient servers. In *Proceedings of the Eleventh European Conference on Computer Systems* (2016), ACM, p. 12.

[150] Sharma, Prateek, Lee, Stephen, Guo, Tian, Irwin, David, and Shenoy, Prashant. Spotcheck: Designing a derivative iaas cloud on the spot market. In *Proceedings of the Tenth European Conference on Computer Systems* (2015), ACM, p. 16.

[151] Sharma, Upendra, Shenoy, Prashant, Sahu, Sambit, and Shaikh, Anees. A cost-aware elasticity provisioning system for the cloud. In *Distributed Computing Systems (ICDCS), 2011 31st International Conference on* (2011), IEEE, pp. 559–570.

[152] Shen, Zhiming, Subbiah, Sethuraman, Gu, Xiaohui, and Wilkes, John. Cloudscale: Elastic resource scaling for multi-tenant cloud systems. In *Proceedings of the 2nd ACM symposium on Cloud computing (SoCC)* (New York, NY, USA, 2011), ACM, pp. 5:1–5:14.

[153] Singh, Rahul, Sharma, Upendra, Cecchet, Emmanuel, and Shenoy, Prashant. Autonomic mix-aware provisioning for non-stationary data center workloads. In *Proceedings of the 7th international conference on Autonomic computing (ICAC)* (New York, NY, USA, 2010), ACM, pp. 21–30.

[154] Singla, Ankit, Chandrasekaran, Balakrishnan, Godfrey, P. Brighten, and Maggs, Bruce. The internet at the speed of light. In *Proceedings of the 13th ACM Workshop on Hot Topics in Networks* (New York, NY, USA, 2014), HotNets-XIII, ACM, pp. 1:1–1:7.

[155] Sovran, Yair, Power, Russell, Aguilera, Marcos K., and Li, Jinyang. Transactional storage for geo-replicated systems. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles* (New York, NY, USA, 2011), SOSP '11, ACM, pp. 385–400.

[156] Steiner, Moritz, Gaglianello, Bob Gaglianello, Gurbani, Vijay, Hilt, Volker, Roome, W.D., Scharf, Michael, and Voith, Thomas. Network-aware service placement in a distributed cloud environment. In *Proceedings of the ACM SIGCOMM* (2012).

[157] Stream Control Transmission Protocol. `http://tools.ietf.org/html/draft-natarajan-http-over-sctp-00`, 2013.

[158] Subramanya, Supreeth, Guo, Tian, Sharma, Prateek, Irwin, David, and Shenoy, Prashant. Spoton: A batch computing service for the spot market. In *Proceedings of the Sixth ACM Symposium on Cloud Computing* (2015), ACM, pp. 329–341.

[159] Talia, Domenico. Toward cloud-based big-data analytics. *IEEE Computer Science* (2013), 98–101.

[160] Techtonik, Anatoly. Python vnc viewer: A simple vnc viewer. `http://code.google.com/p/python-vnc-viewer/`.

[161] The objectweb tpc-w implementation. `http://jmob.ow2.org/tpcw.html`.

[162] Urgaonkar, B., Shenoy, P., Chandra, A., and Goyal, P. Dynamic provisioning of multi-tier internet applications. In *ICAC* (2005).

[163] Vaquero, Luis M, Rodero-Merino, Luis, Caceres, Juan, and Lindner, Maik. A break in the clouds: towards a cloud definition. *ACM SIGCOMM Computer Communication Review 39*, 1 (2008), 50–55.

[164] Villela, Daniel, Pradhan, Prashant, and Rubenstein, Dan. Provisioning servers in the application tier for e-commerce systems. *TOIT* (2007).

[165] Williams, Dan, Jamjoom, Hani, and Weatherspoon, Hakim. The xen-blanket: virtualize once, run everywhere. In *Proceedings of ACM EuroSys* (2012).

[166] Wood, Timothy, Cherkasova, Ludmila, Ozonat, Kivanc, and Shenoy, Prashant. Profiling and modeling resource usage of virtualized applications. In *Middleware* (2008).

[167] Wood, Timothy, Ramakrishnan, K. K., Shenoy, Prashant, and Van der Merwe, Jacobus. CloudNet : Dynamic Pooling of Cloud Resources by Live WAN Migration of Virtual Machines. In *Proceedings of ACM SIGPLAN/SIGOPS conference on Virtual Execution Environments (VEE)* (Mar. 2011).

[168] Xiong, P., Chi, Y., Zhu, S., Moon, H. J., Pu, C., and Hacigümüş, H. Intelligent management of virtualized resources for database systems in cloud environment. In *2011 IEEE 27th International Conference on Data Engineering* (April 2011), pp. 87–98.

[169] Xu, Qiang, Erman, Jeffrey, Gerber, Alexandre, Mao, Zhuoqing, Pang, Jeffrey, and Venkataraman, Shobha. Identifying diverse usage behaviors of smartphone apps. In *Proceedings of the 2011 ACM SIGCOMM conference on Internet measurement conference* (2011), ACM, pp. 329–344.

[170] Xu, Qiang, Erman, Jeffrey, Gerber, Alexandre, Mao, Zhuoqing, Pang, Jeffrey, and Venkataraman, Shobha. Identifying diverse usage behaviors of smartphone apps. In *IMC* (2011).

[171] y. Chen, K., Xu, Y., Xi, K., and Chao, H. J. Intelligent virtual machine placement for cost efficiency in geo-distributed cloud systems. In *2013 IEEE International Conference on Communications (ICC)* (June 2013), pp. 3498–3503.

[172] Zhang, Qi, Cherkasova, Ludmila, and Smirni, Evgenia. A regression-based analytic model for dynamic resource provisioning of multi-tier applications. In *Fourth International Conference on Autonomic Computing (ICAC'07)* (2007), IEEE, pp. 27–27.

[173] Zhang, Qi, Cherkasova, Ludmila, and Smirni, Evgenia. A regression-based analytic model for dynamic resource provisioning of multi-tier applications. In *ICAC* (2007).

[174] Zhu, Qian, and Tung, Teresa. A performance interference model for managing consolidated workloads in qos-aware clouds. In *Cloud Computing (CLOUD), 2012 IEEE 5th International Conference on* (2012), IEEE, pp. 170–179.