Application Workload Prediction and Placement in Cloud Computing Systems

by

Katrina Leigh LaCurts

B.S., Computer Science, University of Maryland (2008)
B.S., Mathematics, University of Maryland (2008)
S.M., Computer Science, Massachusetts Institute of Technology (2010)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy in Computer Science

at the

Massachusetts Institute of Technology

June 2014

©Massachusetts Institute of Technology 2014. All rights reserved.

Author ....
Department of Electrical Engineering and Computer Science
May 16, 2014

Certified by ....
Hari Balakrishnan
Professor
Thesis Supervisor

Accepted by ....
Leslie A. Kolodziejski
Chairman, Department Committee on Graduate Theses

Application Workload Prediction and Placement in Cloud Computing Systems

by

Katrina Leigh LaCurts

Submitted to the Department of Electrical Engineering and Computer Science on May 16, 2014, in partial fulfillment of the requirements for the degree of Doctor of Philosophy in Computer Science

ABSTRACT

Cloud computing has become popular in recent years. Companies such as Amazon and Microsoft host large datacenters of networked machines available for users to rent. These users are varied: from individual researchers to large companies. Their workloads range from short, resource-intensive jobs to long-running user-facing services.

As cloud environments become more heavily used, provisioning the underlying network becomes more important. Previous approaches to deal with this problem involve changing the network infrastructure, for instance by imposing a particular topology [34] or creating a new routing protocol [27]. While these techniques are interesting and successful in their own right, we ask a different question: How can we improve cloud networks *without* changing the network itself? This question is motivated by two desires: first, that customers be able to improve their application's performance without necessarily involving the provider, and second, that our techniques be immediately applicable to today's cloud networks.

This dissertation presents an end-to-end system, Cicada, which improves application performance on cloud networks. Cicada tackles two problems: how to model and predict an application's workload, and how to place applications on machines in the network.

Cicada can be used by either cloud providers or customers. When used by a cloud provider, Cicada enables the provider to offer certain network performance guarantees to its customers. These guarantees give customers the confidence to use cloud resources when building their own user-facing applications (as there is no longer a risk of the cloud under-provisioning for the customer's network needs), and allow providers to better utilize their network. When used by customers, Cicada enables customers to satisfy their own objectives, such as minimizing the completion time of their application. To do this, Cicada exploits variation in the underlying cloud network to use the fastest paths most frequently. This requires an extension to Cicada, called Choreo, which performs quick, accurate, client-side measurement.

We evaluate Cicada using data we collected from HP Cloud, a deployed network with real users. Cicada's workload prediction algorithm outperforms the existing

state-of-the-art [20] by up to 90%. Its placement method reduces application completion time by an average of 22%–43% (maximum improvement: 79%) when applications arrive in real-time, and doubles inter-rack utilization in certain datacenter topologies. These results and others herein corroborate our thesis that application performance can be improved in cloud networks without making any changes to the network itself.

Dissertation Supervisor: Hari Balakrishnan
Title: Professor

To my parents

* * *

She remembered the night of the ball at White Acre, in 1808, when the Italian astronomer had arranged the guests into a tableau vivant *of the heavens, and had conducted them into a splendid dance. Her father—the sun, the center of it all—had called out across the universe, "Give the girl a* place!*" and had encouraged Alma to run. For the first time in her life, it occurred to her that it must have been he, Henry, who had thrust the torch into her hand that night, entrusting her as a Promethean comet across the lawn, and across the wide open world. Nobody else would have had the authority to entrust a child with fire. Nobody else would have bestowed upon Alma the right to have a* place.

Elizabeth Gilbert, "The Signature of All Things"

Contents

List of Figures 10

Previously Published Material 13

Acknowledgments 14

1 Introduction 17
- 1.1 The Problem 18
- 1.2 Philosophy 19
- 1.3 Contents of Dissertation 19
  - 1.3.1 Challenges 20
  - 1.3.2 Overview of Cicada 21
- 1.4 Contributions and Results 22
- 1.5 Outline of Dissertation 24

2 Background 25
- 2.1 Definitions 25
- 2.2 The Public Cloud Environment 26
  - 2.2.1 Networking and Computing 26
  - 2.2.2 Virtual Machine Migration 27
  - 2.2.3 Customer Interaction 27
- 2.3 Context of This Dissertation 27
  - 2.3.1 Assumptions 28
- 2.4 Cicada's Architecture 29
  - 2.4.1 Traffic Prediction 29
  - 2.4.2 Virtual Machine Placement 30
- 2.5 Dataset 32
  - 2.5.1 The HPCS dataset 33
  - 2.5.2 sFlow Data 33
  - 2.5.3 Dataset Size 34
  - 2.5.4 Dataset Limitations 34
  - 2.5.5 Correcting Potential Over-sampling 35
  - 2.5.6 Dataset Usage 35

2.5.7 Dataset Generality ... 36

3 Predicting Application Traffic Patterns **37**
3.1 Introduction ... 37
3.2 Understanding Existing Traffic Patterns ... 39
3.2.1 Spatial Variability ... 39
3.2.2 Temporal Variability ... 40
3.3 Cicada's Traffic Prediction Method ... 41
3.3.1 Related Work ... 41
3.3.2 Prediction Model ... 42
3.3.3 Cicada's Expert-Tracking Algorithm ... 43
3.3.4 Alternate Prediction Algorithms ... 45
3.4 Comparing Cicada to VOC ... 48
3.4.1 VOC-Model ... 48
3.4.2 VOC-style Predictions ... 49
3.4.3 Inefficiencies of VOC ... 50
3.4.4 VOC Parameter Selection ... 50
3.5 Evaluation ... 51
3.5.1 Quantifying Prediction Accuracy ... 52
3.5.2 Determining Whether Predictions Are Reliable ... 53
3.5.3 Prediction Errors for Cicada's Expert-Tracking Algorithm ... 54
3.5.4 Prediction Errors for Alternate Algorithms ... 57
3.5.5 Speed of Predictions ... 63
3.5.6 Summary ... 64
3.6 Conclusion ... 64

4 Application Placement **66**
4.1 Introduction ... 66
4.2 Application Placement Method ... 68
4.2.1 Problem Statement ... 68
4.2.2 Describing the Current Network State ... 68
4.2.3 Describing the Application's Workload ... 69
4.2.4 Placing Applications Using ILPs ... 70
4.2.5 Placing Applications Using Heuristics ... 73
4.2.6 Handling Multiple Applications ... 74
4.3 Application Placement Evaluation ... 75
4.3.1 Improving Application Completion Time ... 75
4.3.2 Dataset and Experimental Set-up ... 76
4.3.3 Improving Network Utilization ... 81
4.3.4 Scalability ... 83
4.3.5 Elasticity ... 83
4.4 End-to-end Evaluation ... 84
4.4.1 Determining Placement with Predictions ... 84

4.4.2 Frequency of Placement Updates ... 86
4.4.3 Cicada with Multiple Customers ... 89
4.5 Conclusion ... 89

5 Provider-centric Cicada: Bandwidth Guarantees 90
5.1 Introduction ... 90
5.2 Related Work ... 91
5.2.1 Determining Guarantees ... 92
5.2.2 Enforcing Guarantees and Fairness ... 92
5.3 Provider/Customer Interactions ... 93
5.3.1 Architecture Overview ... 93
5.3.2 Assumptions ... 94
5.3.3 Measurement Collection ... 94
5.3.4 Prediction Model ... 95
5.3.5 Recovering from Faulty Predictions ... 95
5.4 Implementation ... 96
5.4.1 Server Agents ... 97
5.4.2 Centralized Controller ... 97
5.4.3 Scalability ... 97
5.5 Conclusion ... 98

6 Customer-centric Cicada: Client-side Measurement 100
6.1 Introduction ... 100
6.2 Related Work ... 101
6.3 Understanding Today's Cloud Networks ... 102
6.4 Measurement Techniques ... 105
6.4.1 Measuring Pairwise Throughput ... 106
6.4.2 Estimating Cross Traffic ... 107
6.4.3 Locating Bottlenecks ... 109
6.4.4 Exploring the Network with Additional VMs ... 111
6.5 Evaluation ... 112
6.5.1 Packet Train Accuracy and Temporal Stability ... 112
6.5.2 Cross Traffic ... 116
6.5.3 Bottleneck Locations and Cross Traffic ... 117
6.5.4 Scalability ... 117
6.5.5 Discussion ... 120
6.6 Conclusion ... 121

7 Conclusion 122
7.1 Critiques ... 122
7.2 The Future ... 124

A ILPs for Expressing Additional Goals in Cicada 126

List of Figures

1-1 A typical datacenter network. 17

2-1 A typical datacenter network. 26
2-2 Cicada's architecture. 29
2-3 Collection of the HPCS dataset. 33

3-1 A simple three-tier architecture. The web tier contains three machines; the application and database tiers each contain one. Communication only flows across adjacent tiers. The resulting traffic matrix reflects this, showing that not all machine-pairs communicate the same amount. 38
3-2 Spatial variation in the HPCS dataset. 39
3-3 Temporal variation in the HPCS dataset. 40
3-4 Average weights produced by Cicada's prediction algorithm. These values represent the average of the final weight values over each application in the HPCS dataset. 44
3-5 An example where VOC is inefficient. Because the virtual machines within the group do not send similar amounts of data, more bandwidth is reserved than is used. 50
3-6 Relative error vs. history 53
3-7 54
3-8 Prediction errors for peak demand. 57
3-9 Prediction errors for average-demand using the EWMA algorithm. 58
3-10 Prediction errors for peak-demand using the EWMA algorithm. 59
3-11 Weights produced by Cicada's expert-tracking algorithm when run on a strongly diurnal application. The algorithm assigns almost all of the weight to the datapoint 24 hours earlier. 60
3-12 Prediction errors for average demand using the linear regression-based algorithm. 61
3-13 Prediction errors for peak demand using the linear regression-based algorithm. 62
3-14 The speed of Cicada's prediction algorithm vs. the amount of history available. As the amount of history grows, so does the time it takes to make a prediction. In all but one case, Cicada takes fewer than 10 milliseconds to make a prediction. 63

4-1 An example topology where the greedy network-aware placement is sub-optimal. Because the greedy placement algorithm first places tasks J_1 and J_2 on the path with rate 10, it must also use the path with rate 1 to place J_1 and J_3. The optimal placement avoids this path by placing J_1 and J_2 on the path with rate 9. 73
4-2 Relative speed-up for applications using Cicada vs. alternate placement algorithms. 77
4-3 Cicada's performance improvement compared to the three alternative placements, divided into large and moderate applications. Cicada's performance improvement is comparable for large applications when compared to moderate applications, indicating that large, network-intensive applications are well-served by Cicada. (This experiment was done after that in Figure 4-2, and so the results are not exactly the same, as the network almost certainly changed between the two experiments.) 79
4-4 Relative speed-up of Cicada's placement vs. the optimal placement. The times when Cicada performs better are likely due to changes in the network. 81
4-5 Inter-rack bandwidth available after placing applications (note that the x axis is roughly log-scale). Solid lines represent Cicada's placement, dotted lines represent VOC's placement. 83
4-6 Relative speed-up for applications using Cicada, along with its predictions, vs. alternate placement algorithms. 84
4-7 Change in completion time between using Cicada's predictions to place an application and using ground truth data to place an application. 85
4-8 Percent change in completion time when measuring the network only once vs. every five minutes. 87
4-9 Percent increase in completion time when running Cicada's placement algorithm only once vs. every hour. 88

5-1 Cicada's architecture for providing bandwidth guarantees. 91
5-2 Cicada's implementation. Rate-limiters run in the hypervisor of every physical machine, and communicate with a centralized controller. 96
5-3 Effect of rate-limiters on CPU utilization. We were able to run up to 4096 rate-limiters without significantly effecting on CPU performance. 98

6-1 Cicada's architecture with the addition of Choreo. 101
6-2 TCP throughput measurements on Amazon EC2 taken roughly one year ago, in May, 2012. Each line represents data from a different availability zone in the US East datacenter. Path throughputs vary from as low as 100 Mbit/s to almost 1 Gbit/s. 103

6-3 Achievable TCP throughput measured in May, 2013 on 1710 paths for EC2 and 360 paths for Rackspace. We see some variation on EC2, but less than in Figure 6-2. Rackspace exhibits almost no spatial variation in throughput. 104
6-4 ns-2 simulation topologies for our cross traffic method. The dashed lines are bottlenecks where cross traffic will interfere. 107
6-5 ns-2 simulation results for our cross traffic method. We are able to quickly and accurately determine the number of background connections, particularly for small values of c, even when the number of background connections changes rapidly. 108
6-6 A typical datacenter network (also shown in Figure 2-1). 110
6-7 Percent error for packet train measurements using different burst lengths and sizes. EC2 displays a consistently low error rate over all configurations, while Rackspace's error rate improves dramatically once the burst length is at least 2000. 113
6-8 Percent error when a measurement from τ minutes ago is used to predict the current path throughput. 115
6-9 Comparison of path length with achievable throughput. Path length is not entirely correlated with throughput, as evidenced by the eight-hop paths with throughput over 2500 Mbit/s. 116
6-10 Time it takes for Choreo to measure an n-VM topology using a burst length of 400 packets, and varying burst sizes (n varies with the x-axis). Choreo's measurements perform faster when done in parallel, though at the cost of accuracy in many cases; see Figure 6.5.4 and the accompanying discussion. 118
6-11 120

Previously Published Material

This dissertation includes material from the following publications:

1. K. LaCurts, S. Deng, A. Goyal, H. Balakrishnan. "Choreo: Network-Aware Task Placement for Cloud Applications", *Proc. of the Internet Measurement Conference*, 2013. (Chapters 4 and 6)

2. K. LaCurts, J. C. Mogul, H. Balakrishnan, Y. Turner. "Optimizing a Virtual Datacenter", *Proc. of HotCloud*, 2014. (Chapter 5)

3. K. LaCurts, S. Deng, H. Balakrishnan. "A Plan for Optimizing Network-Intensive Cloud Applications". MIT-CSAIL-TR-2013-003. (Chapter 4 and Appendix A)

4. K. LaCurts, J. C. Mogul, H. Balakrishnan, Y. Turner. "Cicada: Predictive Guarantees for Cloud Network Bandwidth". MIT-CSAIL-TR-2013-004. (primarily Chapters 3 and 5)

The dataset used in Chapters 4 and 6 is available at `http://choreo.mit.edu`.

Acknowledgments

It is possible that I have spent more time in grad school thinking about the acknowledgment section of my dissertation than about my actual dissertation topic. This section may be lengthy, but it is worthwhile.

* * *

First, my advisor, Hari Balakrishnan. Hari emailed me soon after I had been admitted to MIT, saying that he'd like to work with me. This was quite a risk on his part, as I had only recently switched from theory to systems, and did not even know who he was (my then-advisor, surely exasperated with this situation, described Hari as "one of the most important people in networking in the past decade"). Hari took a lot of other risks on me in grad school. He allowed me to switch research topics frequently, gave me my first real teaching job, let me mentor undergraduates, and let me spend one summer teaching high school students instead of working on my dissertation. These risks on his part helped me finally figure out what I wanted to do with my life, which I think is what every grad student hopes to learn from their advisor. There is no other person I would've rather had guide me through the past six years.

In addition to Hari, my research has benefited from a variety of other mentors. My internship at HP Labs with Jeff Mogul formed the basis of this dissertation. He, along with Hari, helped me formulate the questions that underly everything herein. Jeff never lost enthusiasm for this work, and as a result, neither did I. That internship was also bolstered by collaboration with Yoshio Turner, Sujata Banerjee, Gavin Pratt, and a variety of other folks at HP Labs and HP Cloud.

My first internship in grad school was at the International Computer Science Institute, under the guidance of Vern Paxson. Even though the research I did there does not appear in this dissertation, it is my sincere hope that some small fraction of Vern's detailed and precise approach to network measurement does. Christian Kreibich and Jaci Considine brightened that internship; they remain two of my favorite people.

Suchi Raman gave me the opportunity to intern at Plexxi during my final year of grad school. Seeing people outside of academia excited about my dissertation research, as well as seeing its practical applications, was extremely encouraging.

Beyond internships, Sam Madden has been a welcome addition to my dissertation committee, helping me think of my work in a broader setting. Sam, along with Dina Katabi, has been incredibly supportive as I attempted to finish my dissertation and teach a class in the same semester. Jacob White offered me my dream job after grad school. He and Cynthia Skier were instrumental in my decision to accept it.

* * *

There is a large chance that I would not have made it to grad school at all without the support of two people: Brandi Adams and Bobby Bhattacharjee. Brandi has encouraged (and continues to encourage) me more than almost anyone in my entire life (and has allowed me to sleep on her couch an unreasonable number of times). Similarly, Bobby, the exasperated advisor from the first paragraph, never doubted me for one second, or at least never expressed his doubt. I can only hope to do the same for my students as both of these people did for me.

* * *

When I think of places where I've felt most accepted in my life, my original lab at Maryland is near the top of the list. The students and advisors in that lab took me in as an undergraduate, and I still count all of them among my closest friends. Dave Levin forced me into many late nights in the lab running experiments and watching terrible TV, but also gave me the opportunity to co-author my first paper, taught me the ins-and-outs of the research world, and made sure that I knew I had an entire lab behind me no matter what school any of us were working for. Many years later, he also taught me how to shoot a rifle. Randy Baden gave me the courage to leave Maryland (through transitioning techniques such as the Randy-Cam and bird-watching), and then, starting my second year of grad school, showed me what it actually means to be courageous. Aaron Schulman has the most enthusiastic response for every success in my life, despite my accidentally locking him out of our apartment for hours, at midnight, in a foreign country. Neil Spring, Adam Bender, Cristian Lumezanu, and Rob Sherwood have remained my role models in research (and Lume has remained my favorite Romanian).

It is hard to imagine another group of people with whom I would feel as comfortable as those in my first lab, but my second lab at MIT has come remarkably close. Shuo Deng and I bonded over many nights in the lab working on Choreo; that work would not have been possible without her, as well as Ameesh Goyal, both for technical reasons and moral-support reasons. Anirudh Sivaraman and I never worked on a research project together, but due to synchronized work schedules, he has become quite the cohort, and I am already thinking of how we will remain in touch during the day once I change offices. I am excited to see Tiffany Chen every time she walks in our office, because she is always excited to see me. Raluca Ada Popa is the researcher I aspire to be, despite the fact that I was once technically assigned to mentor *her*. Pratiksha Thaker has listened tirelessly to my complaints about students (not that I

would ever complain about a student).

Outside of our office, Jonathan Perry, Ravi Netravali, Peter Iannucci, and Amy Ousterhout have always cheerfully come along when I insisted that we have a "lab outing". Keith Winstein has at times been more excited about my research than anyone. Sheila Marian and Mary McDavitt have made my road through grad school much, much easier. I do not know how I would've handled the logistical aspects of MIT without Sheila.

* * *

I have a variety of friends outside of grad school to thank. JP Dickerson has been my teammate for almost ten years, and I still struggle to make any life decision without him (in fact, he is currently six timezones away, and I wish he was here to help me write this section). Alan Mislove has bought me countless lunches throughout grad school, as well as a plane ticket once, which remains one of the nicest things anyone has ever done for me.[1] Kristen Eaton helped me remember that life exists outside of grad school, and as far as I'm concerned, every Wednesday is still Anything Can Happen Wednesday.

* * *

Saving the best for last, my parents. I have always been pretty fond of my parents, recognized that I am lucky to have them, and been proud to be influenced by them. However, as I've moved away from home, I've spent a lot of time describing the town where I grew up. In doing so, I realized just how rare it was to have parents who prioritized education, who were willing to let their daughter go on to study math, and who were so open to what has been, compared to many of my classmates, an "alternative" life path. I am so, so, lucky. Everything that I ever dreamed I could do as a child has happened because of them, and I do not make a single decision without hoping that it makes them proud.

* * *

Finally, Greggory Isaacson came into my life at exactly the right time, and the pleasure, the privilege, has been mine.

[1] He also spent many hours helping me select a font for this thesis, and he loves a good footnote.

Chapter 1

Introduction

Cloud computing has risen in popularity in recent years. To that end, it has become somewhat of a nebulous term. Many users use "the cloud" with no understanding of the types of machines they have access to, where those machines are, etc. We begin this thesis by defining cloud computing much more precisely.

Much like distributed computing, in cloud computing applications run over multiple computers connected by a network. This network is often a *datacenter* network, which typically implies a tree-like topology, low latencies, and high line rates (non-tree-like topologies have been proposed in [5] and [90]). Each physical machine in the network is home to many *virtual* machines. Figure 1-1 depicts this environment; we discuss it in more detail in §2.2.1.

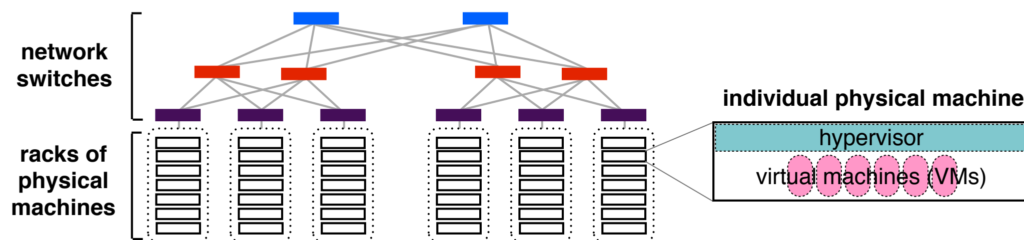


Figure 1-1: A typical datacenter network.

Cloud computing is a unique example of distributed computing due to the type of users involved. Typically, a cloud is managed and owned by one entity; usually a large company such as Amazon or Google. The cloud is used by many diverse customers: from students running code on a handful of virtual machines, to large companies such as Netflix or NASA deploying their services across thousands of cloud machines [17, 18]. These users benefit from the cloud by not having to manage—or even purchase—their own physical resources.

Cloud computing environments can be divided into two types: *private clouds* and *public clouds*. The work in this dissertation applies to both types. In private clouds, the provider and customers are typically associated with the same organization, and

as a result, have some shared goals. Typically, there is also no monetary exchange from the individual customers to the cloud provider.[1] Many companies run their own private clouds, which are used by different—and largely independent—groups within the company; here, the company (or its IT staff) is the provider, and the company's employees are the customers. Similarly, universities often provide this type of service to their faculty and students; here, the provider is the university, the faculty and students are the customers. In contrast, in public clouds, the cloud is owned by one provider and used by many customers who need not be associated with that provider (beyond using their public cloud service). Customers pay the provider to rent machines, usually by the hour. Examples of public clouds include Amazon EC2 [11], Google Compute Engine [33], HP Cloud [42], Rackspace [77], and Windows Azure [95].

1.1 The Problem

As with any popular service, cloud computing is subject to problems, many of which were outlined by researchers at Berkeley in 2009 [16]. Unfortunately, customers feel the brunt of many of these issues. For example, applications do not always run as quickly as customers would like. In the case of public clouds, customers pay per machine, but do not understand their applications well enough to know how many machines or how much inter-machine bandwidth to request. Traffic from other customers can affect how much bandwidth is available for a customer to use, which makes the high link speeds of cloud networks much less enticing (§6.4.2).

These particular problems are exacerbated for *network-intensive* applications. These applications transfer lots of data within the cloud network, and are typically network-bottlenecked rather than CPU- or memory-bottlenecked. Examples include Hadoop jobs [36], analytic database workloads, storage/backup services [8], and scientific or numerical computations.[2] Such applications operate across dozens of machines [86], and are deployed on both public clouds as well as enterprise datacenters.

Our insight is that these problems are caused partly because the customers' needs are unknown to the provider, and often to the customer herself. In particular, the customer usually does not know how much bandwidth her application requires, or has at best a very coarse idea ("No more than 10 GB per hour"). This lack of knowledge can lead to customers over-provisioning and paying for bandwidth that they do not need, or under-provisioning and seeing performance suffer. Moreover, even if the customer *does* know the needs of her application, the provider gives her no way to express those needs beyond making requests for additional machines.

* * *

[1] For consistency, throughout this dissertation we use the term "customers" even when speaking about private clouds, despite the lack of payment.

[2] `http://aws.amazon.com/hpc-applications`

We will show in this thesis that by imbuing the cloud customer with the knowledge of her application's traffic demands, and potentially passing that information on to the provider, application performance can improve in a variety of ways: from minimizing application completion time, to allowing the provider to offer bandwidth guarantees, to improving network utilization. These improvements address common complaints of today's cloud customers.

1.2 Philosophy

Our philosophy in tackling the problem of improving cloud application performance is that one can benefit from knowing what is happening at the *application level*. Rather than rely on solutions that require changes to routing or the topology itself, we design solutions that can be implemented simply by knowing where the application will send traffic. As a result, all of the techniques in this dissertation can be readily deployed *today* on cloud infrastructures, and offer improvement both with and without provider involvement (Chapters 5 and 6, respectively).

As part of this philosophy, a significant portion of this dissertation focuses on network measurement. In order to predict what an application is doing in the future, we must measure what it has done in the past. In order to intelligently place applications, we must know what the underlying network is like; whether it is fast, stable, etc. We believe that network measurement can be done in a lightweight and fast manner, so as not to disrupt currently running applications, and that these measurements provide data that can be used to improve application performance by lowering application completion time, providing predictable guarantees, or improving network utilization.

In particular, we do *not* believe that it is necessary to have complete control over a cloud network to improve application performance. Changes to the network topology or the routing algorithms can be beneficial, but they are not the only option.

We also believe that one can improve application performance and benefit both the customer and the cloud provider. For example, although reducing application completion time results in a lower per-tenant or per-job profit for the provider, it provides a significant competitive advantage, enables more work to be done on the same infrastructure, and encourages the deployment of a greater number of jobs and services.

1.3 Contents of Dissertation

This dissertation focuses on the design and analysis of Cicada, a system designed to improve customers' experience when running applications in the cloud.[3] Cicada solves two problems:

[3]Thanks to Dave Levin, Cicada is an acronym: Cicada Infers Cloud Application Demands Automatically.

1. How to model an application's workload and make predictions about future network demands.
2. How to place an application on the cloud given its predicted workload and the current state of the network.

Both of these problems were motivated by our own measurement studies, and both come with unique challenges.

1.3.1 Challenges

The first problem—workload prediction—is motivated by our own measurement study in §3.2. Using data from HP Cloud Services, we found that many cloud applications exhibit both spatial and temporal variability. That this much variability exists means that most customers cannot be expected to know the demands of their own applications. In order to solve the workload prediction problem, Cicada must overcome the following challenges:

- Cicada must find a way to make predictions that take into account both spatial and temporal variations in an application's workload; most related work in workload prediction only takes into account one of these types of variability. Cicada should be able to make predictions both for average and peak demand (defined precisely in §3.3.2), as different applications have different needs.
- Cicada must be able to detect when its predictions are likely to be incorrect. If Cicada makes a series of incorrect predictions, customers will see decreased performance (or increased cost) and lose faith in the system. It is better for Cicada to classify an application as "unpredictable" than to consistently output incorrect predictions.
- Cicada must be able to make predictions with a relatively small amount of historical data. Though Cicada targets long-running applications, it's unlikely that customers will be willing to wait weeks for Cicada to make its first attempt at improving their application's performance.

The second problem—application placement—is also motivated by our own measurement study (§6.3). In measuring TCP throughput on Amazon EC2, we found that there was significant variation in the TCP throughput on the paths; throughputs varied from 100 Mbit/s to 1 Gbit/s. An application that sends a large amount of traffic over the slower paths will see much poorer performance than a similar application using the faster paths. In order to solve the placement problem, Cicada must overcome the following challenges:

- Placing an application on a cloud network requires a "snapshot" of the network; e.g., knowledge of the TCP throughputs, topology, etc. Cicada must be able to

obtain this snapshot quickly, accurately, and with little overhead. This challenge is of particular concern with Cicada is run by the customer, who does not have easy access to such quantities.

- Once the network has been measured, Cicada must be able to place an application within a fairly short amount of time. If Cicada takes too long to make a placement, the network conditions may have changed; the network snapshot that Cicada used to determine the placement may no longer be valid.

In addition to these individual challenges, we would like Cicada to solve these problems without making changes to the network infrastructure. In that way, our solutions can be readily deployed on today's cloud networks, and even deployed without cooperation from the cloud provider.

1.3.2 Overview of Cicada

In Chapter 3, we describe Cicada's workload prediction algorithm. Cicada is able to make predictions for applications that vary over both space *and* time. It does not require any input from the customer regarding an application's bandwidth needs; Cicada predicts these needs by observing network traffic, removing the need for a customer to understand the intricate details of their application. We show that Cicada can recognize which applications have predictable traffic patterns and make accurate predictions for those applications using only a few hours of data.

In Chapter 4, we describe Cicada's method for placing applications on the cloud. Cicada's placement algorithm exploits variations in cloud networks. Given a snapshot of the current network state, Cicada determines on which machines it is best to place different parts of an application. This placement can be used to satisfy a variety of objectives: for instance, minimizing application completion time, enforcing fault tolerance, and minimizing monetary cost. We also find that by using these techniques, the inter-rack network utilization in certain common datacenter topologies can be more than doubled without under-allocating bandwidth to applications.

* * *

The prediction and placement components of Cicada may be used by either the cloud provider or the cloud customer. Chapter 5 presents an provider-centric application of Cicada. Here, Cicada provides *predictive guarantees* for applications. Predictive guarantees specify the amount of bandwidth an application will use in some future time frame, and where that bandwidth will be needed. These guarantees can be used by the cloud provider to offer bandwidth guarantees to customers, which can improve predictability of application performance and lower costs [20, 96]. We discuss additional details about this environment, including how the customer and provider might interact to agree on the terms of the guarantee, and how network access control might be performed.

Chapter 6 presents a customer-centric application of Cicada. There, Cicada requires *no* interaction from the provider, except the ability of a tenant to obtain a set of machines and place tasks on them; everything else is done entirely on the client's side. Since Cicada requires a snapshot of the cloud network—something that is not readily available to customers—we develop Choreo, a network measurement extension to Cicada. In order to quickly measure network connections, Choreo uses packet trains, which we find work well on public cloud networks (though previous work has shown that it does not always work on the Internet [72, 75]). We also describe methods for measuring cross traffic and for locating bottlenecks, which can be a concern in certain cloud networks.

1.4 Contributions and Results

This dissertation makes both philosophical and technical contributions.

Philosophical contributions: Cicada demonstrates that knowledge of *application-level* traffic patterns can improve performance, and that these traffic patterns can be learned *without any input from the customer*. Traffic-pattern knowledge allows providers to offer customers accurate bandwidth guarantees, improve network utilization, and place applications more intelligently in order to minimize completion time (among other objectives). The fact that Cicada can learn these patterns without customer input makes it a realistic and general system, applicable to a wide range of customers and applications.

Cicada does not require changes to application topology, nor does it require a particular datacenter topology to work (though parts of its measurement framework do require *a* datacenter topology, i.e., a hierarchical, tree-like topology; see §6.4). In fact, although provider involvement can be beneficial, Cicada can be run entirely without provider involvement (Chapter 6), making it applicable to today's cloud customers.

Technical contributions: This dissertation makes technical contributions in the areas of traffic prediction and network measurement.

- Cicada's workload prediction algorithm outperforms a predicting algorithm based on the existing state-of-the-art [20] for static placements by up to 90%, in part because it takes into account both spatial *and* temporal variations in an application's traffic. Previous related work has not taken both types of variations into account, or has only made predictions for a very specific type of application [20, 37, 51, 68, 96]; Cicada can make predictions for a large class of cloud applications. It can also predict when a prediction is likely to be incorrect, and prediction parameters for Oktopus' Virtual Oversubscribed Cluster (VOC) model [20] nearly as well as a perfect oracle.

- Cicada's placement method reduces the average completion time of applications

by 8%–14% (maximum improvement: 61%) when applications are placed all at once, and 22%–43% (maximum improvement: 79%) when applications arrive in real-time, compared to alternative placement methods on a variety of workloads. Its placement method can also double inter-rack utilization in common datacenter topologies. Unlike related work [7, 20, 27, 34, 38, 54, 63, 71], Cicada makes these improvements without making changes to the network infrastructure.

- Choreo, the network measurement extension to Cicada, is able to use packet trains to estimate TCP throughput on ninety network paths within a just a few minutes. As part of Choreo, we also develop new methods for estimating cross traffic and locating bottlenecks within the network.

Through this dissertation, we use real measurement data collected from real public clouds to test and verify our design. Cicada uses data collected from HP Cloud Services, Amazon EC2, and Rackspace. All of these cloud services were deployed networks with real traffic from other users that we did not control. As a result, parts of this dissertation make contributions to measurement studies of cloud networks:

- Using network traces from HP Cloud Services, we show that temporal and spatial variability appear in cloud application traffic; this variability has not previously been quantified for cloud networks, and has rarely been exploited in related work prior to Cicada.
- We present measurements from Amazon EC2 and Rackspace indicating how TCP throughput varies on today's cloud networks, and how this quantity has changed over the past few years.

* * *

Looking toward the future, we believe that this dissertation makes significant contributions towards cloud computing that will remain even as the cloud environment evolves. Cicada was designed around three principles: adapt to the network, utilize application-layer information, and require no changes to the underlying infrastructure. We believe that these principles will remain applicable as cloud networks change. For example, because Cicada starts by measuring the cloud network, we believe that it will provide improvement in the face of changes to network management, rate-limits, etc. Because Cicada does not rely on a particular network infrastructure (beyond there being a datacenter topology), it is likely to provide improvement as the infrastructure changes.

Though today's version of Cicada may not be appropriate for cloud networks years from now, we believe that the general approach of predicting application workloads, measuring the network, and adapting to it will remain useful, and that future versions of Cicada could be built under the same principles to adapt to future cloud networks.

1.5 Outline of Dissertation

The remainder of this dissertation is divided into six chapters. Chapter 2 describes the cloud environment in more detail, places Cicada in a historical context, and describes the dataset that we use throughout this dissertation. The main algorithms appear in Chapters 3 and 4. Chapter 3 describes and evaluates Cicada's prediction algorithm, and includes the measurement study that motivated the need for such an algorithm. Chapter 4 describes and evaluates Cicada's placement algorithm.

Chapters 5 and 6 describe extensions to Cicada, which can be used depending on whether the provider or the customer is deploying the system. Chapter 5 describe how a provider can use Cicada to offer bandwidth guarantees to its customers, and discusses the necessary interactions involved. Chapter 6 presents Choreo, a network measurement system that allows a customer to perform all of Cicada's network measurements on its own.

We conclude this thesis in Chapter 7, where we discuss whether Cicada could be extended to non-cloud networks, as well as to future cloud networks. We also address some potential criticisms of Cicada, such as whether providers are actually motivated to use it.

Chapter 2

Background

2.1 Definitions

Before proceeding, we define some terms that will be used throughout this dissertation.

A **cloud** refers to a group of networked machines controlled by a single entity, which are then used by the public; this type of environment is sometimes referred to as Infrastructure-as-a-Service, or IaaS. The entity controlling the machines is the **provider**. A **customer** is an entity using the cloud. There are two types of clouds: **private** and **public**. In private clouds, the provider and customers are associated with the same organization; for example, in the case of a company providing a cloud for its employees. These clouds are used by diverse teams within the organization. In contrast, customers of public clouds have no affiliation with the provider, beyond using their public cloud service. Examples of public clouds include Amazon EC2 [11], Rackspace [77], and Windows Azure [95].

In private clouds, there is typically no monetary exchange between the individual customers and the providers, as the customer and providers are employed by the same organization. In public clouds, however, customers pay the provider to rent machines, usually by the hour. The work in this dissertation applies to both private and public clouds; for consistency, we use the terms "provider" and "customer" throughout, despite the lack of payment in private clouds.

As is common in industry [70], we make an intentional difference between customer and **tenant**. We define a tenant as a collection of virtual machines that communicate with each other. This definition implies that a single customer may be responsible for multiple tenants.

We use the term **application** to refer to a particular type of software (e.g., Hadoop [36]). When a tenant runs an instance of that software, we refer to it as the tenant's **workload**. A tenant may run multiple applications at once, although it is not uncommon to have a tenant running a single application.

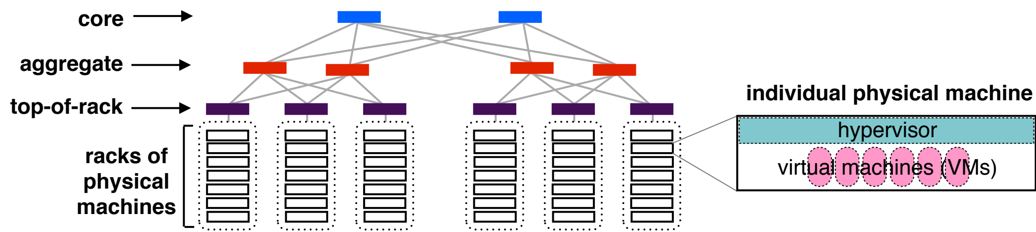


Figure 2-1: A typical datacenter network.

2.2 The Public Cloud Environment

In addition to being under the control of a single administrator, today's public clouds are typically characterized by three features: the network topology and speeds/latencies, the compute environment, and how customers interact with the cloud.

2.2.1 Networking and Computing

The underlying network in most public clouds is a type of datacenter network. These networks are hierarchical, as shown in Figure 2-1. They begin with racks of physical machines, connected by top-of-rack switches. Traffic that exits the rack goes to the next layer—the aggregate layer—and possibly through the core layer. Higher layers generally have fewer switches than lower layers, and the connections between machines at higher layers have higher line rates (see [27] for more details). Higher layers require higher rates as they will potentially transport more traffic than the lower layers. For example, a top-of-rack switch is only responsible for traffic in and out of one rack of machines. Aggregate switches are responsible for traffic in and out of an entire cluster of racks, and core switches are responsible for traffic in and out of multiple clusters of racks.

The layers in a datacenter network can be connected by one of many tree-like topologies. For instance, fat-tree networks [57], or specific types of Clos networks [6]. Unlike pure trees, these networks allow for multiple paths between a source and destination; hence, there can be multiple "best paths". Public clouds often use Equal-Cost Multi-Path routing (ECMP) to determine which path to take [2]. To avoid packet reordering, ECMP usually results in a particular flow remaining on the same path for its lifetime, unless there are failures [41].

In addition to this particular type of topology, datacenter networks often have high line rates, very low round-trip times, and are built with a fairly homogeneous set of equipment [8]. For example, as of 2013, Facebook's datacenter network was connected with 10G, 40G, 80G, and 160G links [32]; as of this writing, typical datacenter latencies are on the order of hundreds or even tens of microseconds [9, 10]. Higher layers are often *oversubscribed*, implying that the maximum amount of traffic that could be

routed through a switch is more than network topology is capable of carrying. This is rarely a problem, as the entire network capacity is rarely used concurrently.

In public clouds, each physical machine employees virtualization to host multiple virtual machines. It is these virtual machines on which customers run applications.

2.2.2 Virtual Machine Migration

Occasionally, virtual machines in a cloud network will need to be *migrated* from one physical machine to another. Migration is particularly relevant to Cicada, as part of Cicada's architecture placing virtual machines and potentially updating their placement (Chapter 4).

In this dissertation, our model is one where virtual machines reside on physical servers, and any large dataset is available to all VMs via a separate storage facility, such as Amazon S3 [13]. Migrating a virtual machine, then, requires migrating its virtual memory and any local data. Typically, virtual machines can be migrated via "live migration", which causes only a few seconds of downtime for the VM [92]. §4.4.2 discusses the impact of migration on Cicada.

2.2.3 Customer Interaction

A customer's interaction with a public cloud starts by requesting a set of virtual machines, often termed **instances**. Currently, the customer can select the computational, storage, and memory capabilities of these machines, but has little to no control over the networking capabilities between their VMs. At best, they can select more computationally powerful machines, which typically results in a higher maximum achieved-throughput, but there is no guarantee about how much bandwidth the customer will receive.

Once the customer has requested these instances, the cloud provider will launch the VMs; once they are up and running, the customer is able to log in to them. The customer may be under the illusion that they have been given their own separate physical machines, and that the connections between those machines are unaffected by any other traffic. In reality, the connections between a customer's virtual machines can be affected by many things in the cloud provider's network; for instance, cross traffic from other users, and shared bottlenecks on the network. We elaborate on these issues in Chapter 6.

2.3 Context of This Dissertation

In 2009, researchers Berkeley set out ten challenges for cloud computing, two of which—data transfer bottlenecks and performance unpredictability—dealt specifically with network performance [16, 15]. Much work in this area has focused on approaches to these problems that require a certain amount of control over the network: either the ability to use a specific topology [34], or the ability to make routing decisions [27].

They may also assume that the cloud customers arrive knowing quite a bit about their application. For example, the techniques in Greenberg et al. [34] require a particular datacenter topology to improve network performance; Ballani et al. [20] assume that the customers know the network needs of their applications before running them on the network.

Cicada offers a complementary approach. We aim to improve application performance by using information about the underlying network and traffic patterns, but *without* making changes to the network. We are particularly concerned with how individual applications perform, and not with more network-specific metrics such as the presence (or lack) of hotspots.

2.3.1 Assumptions

Cicada relies on a few assumptions about the datacenter network and the applications running on it. Cicada targets a particular type of application: long-running, network-intensive applications that require more than a few virtual machines. Due to the nature of Cicada's design, in particular its prediction algorithm, applications that transfer very little data, run for only a few minutes, or use only two or three VMs will experience little improvement; see §3.5 for more details. Moreover, Cicada has been evaluated on a dataset that we believe represents "typical" cloud applications (see §2.5.7); we cannot speak to its performance on more general applications.

Cicada's placement component relies on the assumption that a customer can launch virtual machines in the cloud network, and choose what parts of their applications to place on which of those virtual machines. Alternatively, if Cicada is run entirely by the provider, it relies on the assumption that the provider can place virtual machines anywhere in its own network. Both of these assumptions are true of virtually all cloud networks today.

Most of Cicada's functionality does not rely on any assumptions about the datacenter network itself. However, parts of Choreo (Chapter 6), Cicada's network measurement extension, requires two: that the underlying topology be some type of datacenter topology (i.e., a hierarchical, tree-like topology), and that there be relatively high line rates (in order to measure the network quickly). Furthermore, customers using Cicada will see the most benefit from its placement algorithm when there is a large disparity in the achieved TCP throughputs on paths, as we see in today's cloud networks (§6.3).

In Chapter 7, we discuss how Cicada might perform if some of these assumptions were removed (for instance, if Cicada were used on non-cloud networks, or on a more general set of applications).

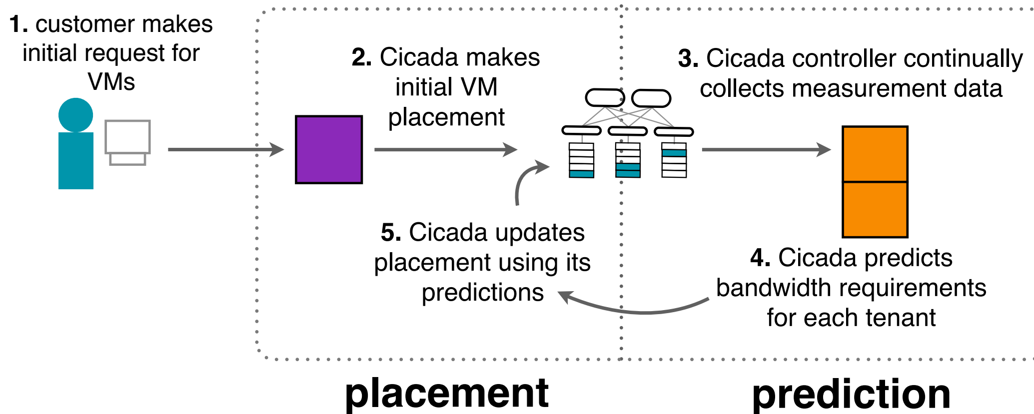


Figure 2-2: Cicada's architecture.

2.4 Cicada's Architecture

Cicada's high-level architecture is detailed in Figure 2-2. It divides into two sections: prediction and placement.

2.4.1 Traffic Prediction

Cicada predicts application traffic patterns in order to help providers and customers better estimate an application's network needs. It makes predictions for both average demands—the total amount of data an application will need to transfer over some future time interval—as well as peak demands—the maximum amount of data an application will need to transfer over a small time interval; see §3.3.2 for a more precise definition of mean and peak demands. In the context of today's cloud networks, this goal puts Cicada a step beyond research that assumes customers already know these patterns beforehand [20, 56].

To put Cicada's predictive approach in an historical context, it resembles some types of ISP traffic engineering, which ISPs deploy in practice. Traffic engineering uses estimates of traffic to map flows or aggregates to paths in a network, and attempts to predict the amount of traffic that links in a network will carry. However, Cicada tackles a slightly different problem than many traffic engineering schemes. These schemes try to estimate true traffic matrices from noisy measurements [85, 97], such as link-load data. Cicada, on the other hand knows the exact traffic matrices observed in past epochs; its problem is to predict a *future* traffic matrix.

There is some work on traffic engineering schemes that attempt to predict future demands. On ISP networks, COPE [94] describes a strategy that predicts the best *routing* over a space of traffic predictions, made up of the convex hull of previous traffic matrices. It is not a traffic-prediction scheme per se, but we draw inspiration

from this work in Chapter 3.

Cicada also uses its predictions for a different purpose than traffic engineering: to place virtual machines or tasks (Chapter 4), or to offer bandwidth guarantees to tenants (Chapter 5).

We discuss prior work related specific to Cicada's prediction algorithm in Chapter 3.

2.4.2 Virtual Machine Placement

Cicada's placement component deals with how to place an application's components—its *tasks*—on the cloud network. This problem is closely related to the problem of virtual machine placement (in fact, in the case where there is one task per virtual machine, the problems are identical).

The problem of virtual machine placement is as follows: given a network of M machines, each machine having a CPU constraint representing the amount of available CPU (this problem could easily be extended to include CPU and RAM), and each pair of machines having a network constraint representing the measured TCP throughput between the machines (we choose to concentrate on TCP traffic since most datacenter traffic uses TCP [8]).

A tenant wants to run a set of k applications on this network; they may all be known up front, or they may come in sequence. Each application A_i uses J_i virtual machines (VMs), which need to be placed on the M physical machines in the network. More than one VM is allowed on each machine, so long as the CPU constraints are still satisfied. Each VM has a CPU demand, representing the amount of CPU needed by that VM to complete its computations (this demand is typically given by the customer, and may be expressed, for example, as the number of cores a VM requires). Each pair of VMs has a network demand, representing the amount of data that will be transferred between them (demands can be asymmetric; $i \to j$ may have a different value than $j \to i$). The goal is to place the k applications on the M machines so as to satisfy some objective—for example, to minimize the application completion time—while also satisfying the per-machine CPU constraints

Previous papers on network virtualization and optimization for cloud computing systems and datacenters address one of two problems: how to choose paths between fixed machines, or how to choose machines given a fixed network. Cicada's traffic placement mechanism falls into the second category, as work in the first generally assumes a level of control over the network that we do not assume in this dissertation.

How to Choose Paths

Given a fixed network, there are many ways to choose paths in the network to route from one machine to another. Many datacenters choose paths with load balancing in mind. A popular method mentioned earlier is Equal-Cost Multi-Path forwarding [27], where a flow's path is selected based on a hash of various header fields. VL2 [34]

uses Valiant Load Balancing on a Clos network to select routes. Hedera [7] designed a dynamic flow scheduling system, which can move flows in a network to efficiently utilize all network resources. Other approaches to load balancing in datacenters include software-defined networking, as in [38].

Other schemes, such as NetShare [54], focus on choosing routes for performance isolation, ensuring that network resources are shared among tenants based on some weight function. All of these systems differ from Cicada in that they have control, at a minimum, of the routing protocol in the network. Cicada aims to improve performance *without* changing routing protocols.

How to Choose Machines

Now we turn our attention to the following problem: Given a large network of machines, how do we pick which machines to run a particular workload on? Here we assume that the routes between machines are fixed. Oktopus [20] uses a virtual topology to place tenant VMs such that they meet bandwidth and oversubscription requirements. This system must be deployed by providers, but offers more stable workload completion times (the goal of Oktopus is to improve the accuracy of predicted completion times, rather than to focus on finding an optimal placement). Unlike Oktopus, Cicada need not be deployed by providers. However, if it is, Cicada can be used to provide bandwidth guarantees that are more accurate than those derived from Oktopus, decreasing the median error by 90% (see §3.5 and Chapter 5).

Researchers have also designed an algorithm to place VMs on physical machines with the goal of minimizing the number of physical machines used while also meeting the VMs' requirements for CPU, memory, and bandwidth [63]. In this proposal, it is not clear whether virtual machines are allowed to transfer data to multiple machines, nor is it clear how it would operate in general; the only simulation is done on an extremely small topology with a highly structured workload. Purlieus [71] reduces MapReduce job execution times by placing Map/Reduce nodes in a locality-aware manner. This method requires information about the network topology, which may not be available to Cicada. Even in the case where network topology information is available, Cicada can provide improvement for a much more general case of applications, not just MapReduce jobs.

The largest difference between Cicada and these works is that we present network-aware placement schemes that run with *no changes to the network*, assuming no detailed, a priori knowledge of the topology is assumed, and not introducing a new routing protocol. In fact, as we show in Chapter 6, Cicada may even be run entirely by customers, without provider assistance. In addition to making no changes to the network, Cicada infers application demands automatically, without any input from the customer.

Our idea of network-aware placement is in contrast to workload placement systems that only take CPU constraints into account [3, 76]. These systems generally attempt to load balance across machines or minimize the number of physical machines used.

Additionally, Cicada operates in real-time; other work [51] measures traffic for weeks before making a placement.

2.5 Dataset

In order to understand and evaluate both of the main problems in this dissertation—workload prediction and application placement—we need access to network measurements from cloud networks. We must understand how the traffic patterns of cloud application workloads vary in order to make predictions about them, and we must understand how cloud networks vary in order to place applications on them. Furthermore, to properly evaluate Cicada, we need to test its prediction and placement on real applications under real network conditions.

Despite the importance of datacenter networks, the research community has had scant access to measurements, particularly of VM-to-VM traffic matrices in IaaS clouds. We are interested in obtaining this type of data for two reasons:

1. Cicada's prediction method makes predictions for individual VM pairs (or even individual "task" pairs; see §3.3.2), in order to exploit spatial and temporal variability between VMs. For this reason, we need VM-to-VM traffic matrices; aggregated data will not suffice.

2. We're interested in VM-to-VM traffic matrices from IaaS clouds as we speculate that these type of clouds may contain a different mix of long-running, network-intensive applications than other cloud or datacenter environments, in part due to the large number of *independent* customers that use these clouds, and the typical cloud use cases (e.g., MapReduce jobs, scientific computing, etc.). The applications running of IaaS clouds may have characteristics that a more general set of applications does not.

   It's possible that this assumption is incorrect; maybe cloud applications exhibit the same workload characteristics as some more general class of applications. Even so, we would still need a dataset of IaaS-cloud applications in order to prove that.

These measurements can be difficult for researchers to obtain due to the challenges of gathering large-scale measurements, the privacy and security risks created by these data sets, and the proprietary value that providers place on understanding what goes on in their networks. We know of no published measurement studies on IaaS traffic matrices (except perhaps [21], discussed below).

Prior studies on datacenter networks have detected temporal and spatial variability. Benson et al. [22] analyzed link-level SNMP logs from nineteen datacenters, finding "temporal and spatial variations in link loads and losses." These, we believe, were not cloud (IaaS) networks *per se* (they may have been SaaS/PaaS datacenters),

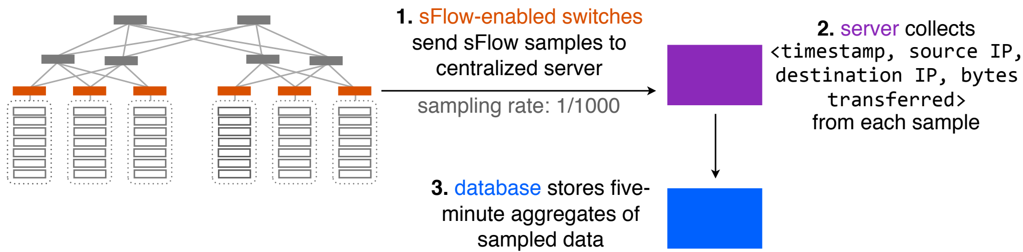


Figure 2-3: Collection of the HPCS dataset.

although their applications may be similar to those of cloud tenants. Benson et al. [21] gathered SNMP statistics for ten datacenters and packet traces from a few switches in four datacenters. They describe several of these as "cloud data centers," but it is unclear whether they are actually IaaS networks. They report that "diurnal patterns [in link utilization] exist in all [ten] data centers," and that "time-of-day and day-of-week variation exists in many of the data centers," especially in the cores of these networks.

Greenberg et al. [34] report on SNMP and NetFlow data from a "large cloud service provider." We believe this, too, is not an IaaS provider. They report a distinct lack of short-term predictability for traffic matrices, but do not say whether this datacenter experiences diurnal variations. Kandula et al. [49] found considerable short-term spatial variation in a 1500-server data-mining cluster, but did not investigate whether this variation is predictable. Bodík et al. [24] also found spatial variation in inter-service traffic in a datacenter, as a result of the fact that machines responsible for different services did not typically communicate with one another.

2.5.1 The HPCS dataset

Throughout this dissertation, we use a dataset collected from HP Cloud Services. We collected sFlow [88] data from HP Cloud Services, which we refer to as the HPCS dataset. Figure 2-3 shows our dataset collection process, which we detail below.

2.5.2 sFlow Data

Collecting sFlow data requires having sFlow-enabled switches in a network, as the data is gathered from the switches themselves. To achieve scalability, sFlow uses packet sampling; each switch sends the sampled data to a specified host. The data comes as a series of *sFlow datagrams*. These datagrams contain information pertaining to the type of sampling done, as well as one or more samples. The samples contain standard packet trace fields, such as source IP, source MAC, etc.

In our implementation, our sFlow-enabled switches randomly sampled one out of every 1000 packets, and sent the sFlow datagrams to a single centralized server. For each sample, the server recorded the following:

- Timestamp of the sample
- Source IP address
- Destination IP address
- Number of bytes transferred from the source to the destination in this sample

For privacy reasons, we kept no information pertaining to the packet payload.

2.5.3 Dataset Size

Our dataset consists of about six months of samples from 71 Top-of-Rack (ToR) switches. Each ToR switch connects to either 48 servers via 1GbE NICs, or sixteen servers via 10GbE NICs. In total, the data represents about 1360 servers, spanning several availability zones (portions of a datacenter that are isolated from failures in other parts of the same datacenter). This dataset differs from those in previous work—such as [21, 22, 34]—in that it captures VM-to-VM traffic patterns. It does not include any information about what types of applications were running (e.g., MapReduce jobs, webservers, etc.), as that information is not available from the packet headers that we captured.

Due to the amount of storage we had available, we aggregate the sFlow data over five-minute intervals, to one datapoint of the form ⟨*timestamp, source IP, destination IP, number of bytes transferred from source to destination*⟩ every five minutes. We keep only the data where both the source and destination are private IPs of virtual machines, and thus all our datapoints represent traffic *within* the datacenter. Making predictions about traffic traveling outside the datacenter or between datacenters may be a more challenging task, in part due to the additional cross traffic and lower line speeds available in parts of the Internet. We do not address this challenge in this work. For privacy reasons, after this processing, we store a deterministic hash of the IP, rather than the IP itself.

Under the agreement by which we obtained this data, we are unable to reveal information such as the total number of tenants, the number of VMs per tenant, or the growth rates for these values.

2.5.4 Dataset Limitations

Because sFlow samples packets, it cannot provide perfect information regarding how many bytes were sent between a source and a destination; it may entirely miss flows from sources that send very little data. However, we do not believe that missing these small flows has any effect on Cicada's overall performance. After all, if sources A and B communicate rarely, they will not suffer from being placed on a particularly slow path.

Note that because sFlow samples packets, not flows, if A and B send more than one thousand packets between them, it is highly likely that at least one sample will be

generated; it does not matter if A and B send that traffic as many small flows or a few large flows.

During data collection, the physical network was generally over-provisioned. Hence, our measurements reflect the actual offered loads at the virtual interfaces of the tenant VMs. Some of the smaller VMs were output-rate-limited at their virtual interfaces by HPCS; we do not know these rate-limits.

Because the HPCS dataset samples come from the ToR switches, they do not include any traffic between pairs of VMs when both are on the same server. We believe that we still get samples from most VM pairs, because, in the measured configuration (OpenStack Diablo), VMs are placed on servers uniformly at random. Assuming such placement on n servers, the expected number of unordered pairs of k VMs sharing n servers is $\frac{1}{n}\binom{k}{2}$. There are $\binom{k}{2}$ possible unordered pairs among k VMs, so the probability that any VM-pair shares the same server is $\frac{1}{n}\binom{k}{2}/\binom{k}{2} = \frac{1}{n}$. With $n = 1360$, the probability that we will miss any given VM-pair's traffic is less than $.001\%$.

For privacy reasons, our dataset does not include information associating VMs with tenants or applications. Instead, we define tenants by using the connected components of the full traffic matrix, which is in line with the OpenStack definition of tenants.

2.5.5 Correcting Potential Over-sampling

Some of the paths in the HPCS dataset are over-sampled. If A and B are VMs on the same ToR switch S_1, traffic on $A \rightsquigarrow B$ will only be sampled at S_1. But if VM C resides on a different ToR S_2, traffic on $A \rightsquigarrow C$ will be sampled at both S_1 and S_2, twice as often as $A \rightsquigarrow B$.

We correct this problem by noting which switches we see for each VM-pair (sFlow samples are tagged with the ToR's own address). If we see two switches on a path, we know that this flow has been oversampled, so we scale those measurements down by a factor of two.

2.5.6 Dataset Usage

We use the HPCS dataset throughout this dissertation. First, in Chapter 3, we use it to motivate the need for a prediction algorithm that captures spatial and temporal variability, by showing that the cloud applications captured in the HPCS dataset exhibit both types of variability. Then, we use it to evaluate Cicada's prediction algorithm, showing that on the applications in our dataset, Cicada outperforms the existing state-of-the-art. Finally, we use the HPCS dataset to evaluate Cicada's placement algorithm, showing that it improves application completion time on the applications represented in this dataset.

One feature which is *not* captured in the HPCS dataset is a distribution of achieved TCP throughputs on today's cloud networks. We collect an additional dataset in

Chapter 6 to determine this quantity, which we use in evaluating Choreo, an extension to Cicada which allows a customer to perform their own network measurements.

2.5.7 Dataset Generality

We believe that our dataset describes a fairly general set of cloud applications. We collected traces from real users on an actual IaaS-cloud; this is in contrast to work that uses simulated data [63] or data from different types of datacenter networks [22, 24, 34, 49].

However, at the time of data collection, HP Cloud Services was likely not being used as heavily as more well-established public clouds (e.g., Amazon EC2). For that reason, we believe that our dataset may not capture as many user-facing applications as a dataset from a different cloud would. In an attempt to correct for this possible omission, we simulate various user-facing applications in §3.5.4, showing that Cicada also performs well on those.

Chapter 3

Predicting Application Traffic Patterns

3.1 Introduction

The first of Cicada's two major components, as shown in Figure 2-2, is its traffic prediction module. Cicada makes predictions about how much traffic a tenant's virtual machines will send to each other, and exploits predictable inhomogeneity in traffic demands to create options for better task placement. Most existing systems for improving application performance in the cloud assume that tenants already know their application's requirements [20, 56]. These systems require tenants to explicitly specify bandwidth requirements, or to request a specific set of network resources.

But do cloud customers really know their network requirements? Application bandwidth demands can be complex and time-varying, and not all application owners accurately know their bandwidth demands. A tenant's lack of accurate knowledge about its *future bandwidth demands* can lead to over- or under-provisioning. We show in this dissertation that increasing the level of detail known about these demands can lead to more accurate bandwidth guarantees (Chapter 5) and better placements (Chapter 4).

In this chapter, using the HPCS dataset (§2.5), we demonstrate that tenant bandwidth demands can be time-varying and spatially inhomogeneous. We show, however, that they can also be predicted, based on automated inference from their previous history. This result suggests that it is difficult for tenants to determine their applications' bandwidth requirements without a sophisticated prediction mechanism such as Cicada.

Cicada's traffic predictions support time-varying and space-varying demands. Prior approaches to prediction typically offer predictions that are static in at least one of those respects, but these approaches do not capture general cloud applications. For example, we expect *temporal variation* for user-facing applications with diurnally-varying workloads, and *spatial variation* in VM-to-VM traffic for applications such as three-tier services, as shown in Figure 3-1. Additionally, Cicada can make both long-term (on the order of hours) and short-term (on the order of minutes) predictions.

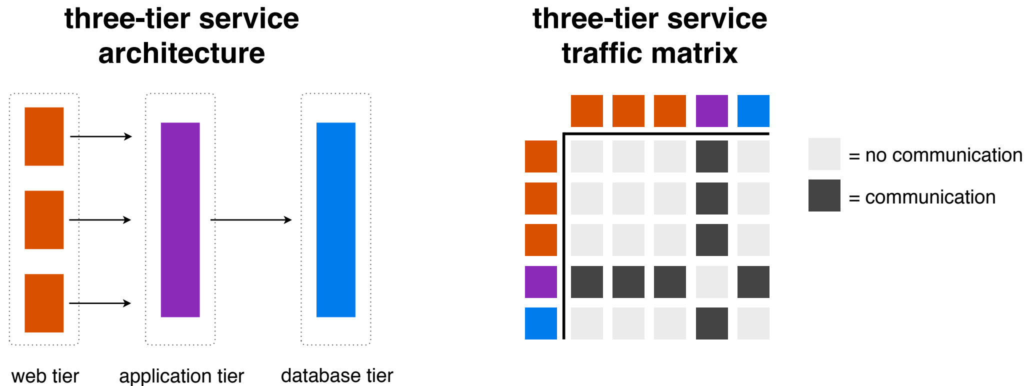


Figure 3-1: A simple three-tier architecture. The web tier contains three machines; the application and database tiers each contain one. Communication only flows across adjacent tiers. The resulting traffic matrix reflects this, showing that not all machine-pairs communicate the same amount.

Recent research has addressed these issues in part. For example, Oktopus [20] supports a limited form of spatial variation; Proteus [96] supports a limited form of temporal-variation prediction. But no prior work, to our knowledge, has offered a comprehensive framework for cloud customers and providers to make predictions about applications with time-varying and space-varying traffic demands.

Our primary contributions in this chapter are Cicada's prediction algorithm, and a trace-based analyses of the motivation for, and utility of, our approach. We answer the following questions:

1. *Do real applications exhibit traffic variability?* We show that they do, using network traces from HP Cloud Services, and thus justify the need for predictions based on temporal and spatial variability in application traffic. Although such variability is not surprising, it has not previously been quantified for cloud networks. Cicada's prediction algorithm is designed with this type of variability in mind.

2. *How does Cicada predict network demands?* We describe Cicada's prediction algorithm. The algorithm treats certain previously observed traffic matrices as "experts" [40] and computes a weighted linear combination of the experts as the prediction. The weights are computed automatically using online learning.

3. *How well does Cicada predict network demands?* Using our network traces, we show that the median prediction errors of the algorithm are 90% lower than a prediction algorithm based on Oktopus' Virtual Oversubscribed Cluster (VOC) [20] (this model is a hierarchical hose model; see §3.4 for a detailed description). Moreover, our algorithm can often predict when the prediction is

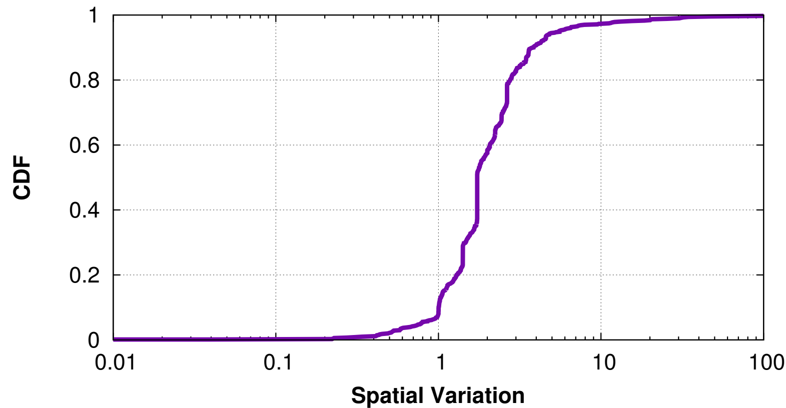


Figure 3-2: Spatial variation in the HPCS dataset.

likely to be wrong. It also predicts parameters for the VOC model nearly as well as a perfect oracle.

In Chapter 4, we show how Cicada uses these predictions to place virtual machines in the cloud (via its placement module, also shown in Figure 2-2). In Chapter 5, we show how a provider can turn these predictions into bandwidth guarantees for its customers.

3.2 Understanding Existing Traffic Patterns

As mentioned earlier, Cicada is designed to make predictions for applications that exhibit both spatial and temporal variability. Before building Cicada, however, we must determine whether cloud applications exhibit both of these types of variability. If they do, existing approaches (e.g., VOC-style allocations [20], or static, all-to-all traffic matrices) will not suffice; there is no existing work that captures both spatial and temporal variability of general cloud applications.

In this section, using the dataset described in §2.5, we analyze the spatial and temporal variability of its tenants' traffic.

3.2.1 Spatial Variability

To quantify spatial variability, we compare the observed tenants to an ideal, static, all-to-all tenant. This tenant is ideal as it is the easiest to make predictions for: every intra-tenant connection sends the same amount of data at a constant rate. Let F_{ij} be the fraction of this tenant's traffic sent from VM_i to VM_j. For the "ideal" all-to-all tenant, the distribution of these F values has a standard deviation of zero, because every VM sends the same amount of data to every other VM. As another example, consider a bimodal tenant with some pairs that never converse and others that converse equally.

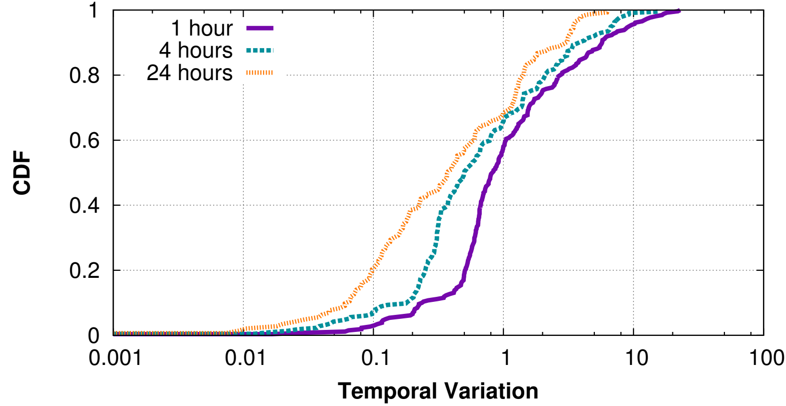


Figure 3-3: Temporal variation in the HPCS dataset.

Let $k = n/2$. Each VM communicates with $1/k$ VMs, and sends $1/k$ of its traffic, hence half of the F values are $1/k$; the other half are zero.

For each tenant in the HPCS dataset, we calculate the distribution of its F values, and plot the coefficient of variation (standard deviation over mean) in Figure 3-2. The median cov value is 1.732, which suggests nontrivial overall spatial variation (for contrast, note that the cov of our ideal tenant is zero, and the cov of our bimodal tenant will be one[1]).

Some VMs communicate much more than others: in this data, one tenant with $cov > 10$ has eight VMs, with a few pairs that send modest amounts of data, and all other pairs sending little to no data. These results suggest that a uniform, all-to-all model is insufficient for making traffic predictions; given the high cov values in Figure 3-2, a less strict but not entirely general model such as VOC [20] may not be sufficient either (see §3.4 for a detailed description of this model).

3.2.2 Temporal Variability

To quantify temporal variability, we first pick a time interval H. For each consecutive non-overlapping interval of H hours, we calculate the sum of the total number of bytes sent between each pair p of VMs. This gives us a distribution T_p of these totals. We then compute the coefficient of variation for this distribution, cov_p. The temporal variation for a tenant is the weighted sum of these values, where the weight for cov_p is the total amount of data sent between the pair p. This scaling reflects the notion that tenants where only one small flow changes over time are less temporally-variable than those where one large flow changes over time.

For each tenant in the HPCS dataset, we calculate its temporal variation value, and plot the CDF in Figure 3-3. Like the spatial variation graph, Figure 3-3 shows that most

[1]For the bimodal tenant, $\mu = 1/2k$ and $\sigma = \sqrt{1/n \sum_{i=1}^{n} (1/2k)^2} = 1/2k$. So $cov = 1$.

tenants have high temporal variability. This variability decreases as we increase the time interval H, but we see variability at all time scales. Tenants with high temporal variation are typically ones that transfer little to no data for long stretches of time, interspersed with short bursts of activity.

The results so far indicate that typical cloud tenants may not fit a rigid model. In particular, Figure 3-3 shows that most tenants exhibit significant temporal variation. Thus, static models cannot accurately represent the traffic patterns of the tenants in the HPCS dataset.

3.3 Cicada's Traffic Prediction Method

3.3.1 Related Work

Much work has been done on estimating traffic matrices from noisy measurements such as link-load data [85, 97]. In these scenarios, approaches such as Kalman filters and Hidden Markov Models—which try to estimate true values from noisy samples—are appropriate. Cicada, however, knows the exact traffic matrices observed in past epochs (an epoch is H-hours long; typically one hour); its problem is to predict a *future* traffic matrix.

To the best of our knowledge, there is little work in this area, especially in the context of cloud computing. On ISP networks, COPE [94] describes a strategy that predicts the best *routing* over a space of traffic predictions, made up of the convex hull of previous traffic matrices. It is not a traffic prediction scheme per se, but we draw inspiration from this work, and base our prediction algorithm around computing a "best" convex combination of previous traffic matrices as our future estimate.

The Proteus system [96] profiles specific MapReduce jobs at a fine time scale, to exploit the predictable phased behavior of these jobs. It supports a "temporally interleaved virtual cluster" model, in which multiple MapReduce jobs are scheduled so that their network-intensive phases do not interfere with each other. Proteus assumes uniform all-to-all hose-model bandwidth requirements during network-intensive phases, although each such phase can run at a different predicted bandwidth (see §3.3.2 for a description of hose-model requirements). Unlike Cicada, it does not generalize to a broad range of enterprise applications. Similarly, [1] was also developed to profile Hadoop jobs, and does not generalize to the applications with which we are concerned.

Other recent work has focused on making traffic predictions to produce short-term (ten-minute) guarantees for video streaming applications [68]. This work uses a factor model, which models traffic demands as being driven by some number of uncorrelated underlying factors. The factors are found by using principal-component analysis on the video traffic, and the components are modeled using a seasonal ARIMA model. Although this work considers VM-to-VM predictions, it is not clear that the approach generalizes to long-term (more than ten minutes into the future) predictions, or to applications beyond video streaming.

3.3.2 Prediction Model

As stated, Cicada's prediction take into account both spatial and temporal variations in bandwidth demand. Cicada has the ability to predict both average traffic demands as well as peak traffic demands. Average demand predictions predict the mean bandwidth that an application will need over a period of H hours. Peak demand predictions predict the maximum bandwidth expected during any averaging interval δ during a given time interval H. If $\delta = H$, a peak prediction is equivalent to the average prediction. For many applications, we expect $\delta \ll H$.

Cicada's prediction algorithm takes, as input, a time series of previously observed traffic matrices, $M_1, \ldots, M_n$. These matrices are collected at equally-spaced intervals at times in the past. Each matrix represents the aggregated data for one H-hour interval, and data is collected continuously. That is, assuming that data collection begins at time t_0, matrix M_1 represents the aggregated data from t_0 to $t_0 + H$; matrix M_2 represents the aggregated data from $t_0 + H$ to $t_0 + 2 \cdot H$; matrix M_i represents the aggregated data from $t_0 + (i-1) \cdot H$ to $t_0 + i \cdot$.

The rows and columns of the matrix represent an application's *tasks*. We use "tasks" as an intuitive term: a task may map to a collection of processes in an application, for instance a map or reduce task during a MapReduce job. Alternatively, one could consider the group of processes that runs on a single virtual machine to be one task, in which case our prediction algorithm is making VM-to-VM predictions, rather than more fine-grained task-to-task predictions. This is the case in our evaluation (§3.5).

An entry in row i and column j specifies the number of bytes that task (or VM) i sent to task (or VM) j in the corresponding epoch (for predicting average demand) or the maximum observed over a δ-length interval (for peak demand). Cicada's algorithm produces $\hat{M}_{n+1}$, the prediction for epoch $n+1$.

Pipe and Hose Models

As described, Cicada's predictions conform to a *pipe model*. A pipe-model prediction is one where a separate prediction is made between each task pair; for k tasks, there are $k \cdot (k-1)$ predictions. A pipe model contrasts to a *hose model*, where a prediction is made for each task to the set of all other tasks; for k tasks, there are k predictions. These terms were originally applied by Duffield et al. to resource management in VPNs [30].

Cicada can output hose model predictions in addition to pipe-model predictions simply by changing the input matrices $M_1, \ldots, M_n$. To output hose-model predictions, we use input matrices that represent the hose bandwidth. In this case, each matrix is a $1 \times n$ matrix, and entry i corresponds to the total number of bytes sent from task i.

3.3.3 Cicada's Expert-Tracking Algorithm

Cicada's algorithm uses Herbster and Warmuth's "tracking the best expert" idea [40], which has been successfully adapted before in wireless power-saving and energy reduction contexts [29, 64]. To predict the traffic matrix for epoch $n+1$, we use all previously observed traffic matrices, $M_1, \ldots, M_n$ (later, we show that matrices from the distant past can be pruned away without affecting accuracy). Algorithm 1 gives the pseudocode for this algorithm.

Each of these previously observed matrices acts as an "expert," recommending that M_i is the best predictor, $\hat{M}_{n+1}$, for epoch $n+1$. The algorithm computes $\hat{M}_{n+1}$ as a weighted linear combination of these matrices:

$$\hat{M}_{n+1} \quad = \quad \sum_{i=1}^{n} w_i(n) \cdot M_i$$

where $w_i(n)$ denotes the weight given to M_i when making a prediction for epoch $n+1$, with $\sum_{i=1}^{n} w_i(n) = 1$.

The algorithm learns the weights *online*; there is no notion of training vs. testing data. At each step, it updates the weights according to the following rule:

$$w_i(n+1) = \frac{1}{Z_{n+1}} \cdot w_i(n) \cdot e^{-L(i,n)}$$

where $L(i,n)$ denotes the *loss of expert i in epoch n* and Z_{n+1} normalizes the distribution so that the weights sum to unity. In keeping with [40], we use the term loss to mean the discrepancy (or error) between expert i and the true outcome.

We use the relative Euclidean ℓ^2-norm between M_i and M_n as the loss function L. Treating both these matrices as vectors, $\vec{M_i}$ and $\vec{M_n}$ respectively, the relative ℓ^2-norm error is

$$L(i,n) \quad = \quad E_{\ell^2} = \frac{\|\vec{M_i} - \vec{M_n}\|}{\|\vec{M_n}\|}.$$

That is, the norm of the individual errors over the Euclidean norm of the observed data (the square-root of the sum of the squares of the components of the vector).

At each step, a new weight must be added to the set of weights; there are $n-1$ weights used to predict $\hat{M}_n$, but n used to predict $\hat{M}_{n+1}$. To do this, we insert a small weight to the beginning of the weight vector (see line 4 of Algorithm 1). This insertion has the effect of shifting all of the other weights: w_0 becomes w_1, w_1 becomes w_2, etc. To understand why this process is appropriate, consider an alternate negative indexing for the weights: $w_n = w_{-1}; w_{n-1} = w_{-2}; \ldots; w_1 = w_{-n}$. With this indexing, w_{-1} represents the weight given to the most recent matrix, w_{-2} represents the weight given to the second-most recent matrix, etc. Our insertion adds a new weight for the nth-most recent matrix. This matrix was not present when predicting $\hat{M}_n$ (there were

Algorithm 1 Cicada's expert-tracking algorithm

Input: $M_1, \ldots, M_n$, the previous observed matrices

1: $W = \{w_i\}$, the series of $n-1$ current weights (used to calculate $\hat{M}_n$)
2: **for** $w_i \in W$ **do**
3: $\quad w_i = w_i \cdot e^{-L(i,n)}$
4: $W = \{min(w_i) \cdot .5\} + W$
5: $Z = \sum_i^n w_i$
6: **for** $w_i \in W$ **do**
7: $\quad w_i = w_i / Z$
8: $\hat{M}_{n+1} = \sum_{i=1}^n w_i \cdot M_i$

Output: $\hat{M}_{n+1}$, the prediction matrix for all task pairs

only $n+1$ previous matrices at that time), and so deserves a small weight as its "expert opinion" has not yet been tested.

Note that this algorithm can predict both average demand as well as peak demand. The only difference is the input matrices: for average demand, the matrices $M_1, \ldots, M_n$ represent the total amount of data in each epoch, while for peak demand, they represent the maximum over a δ-length interval.

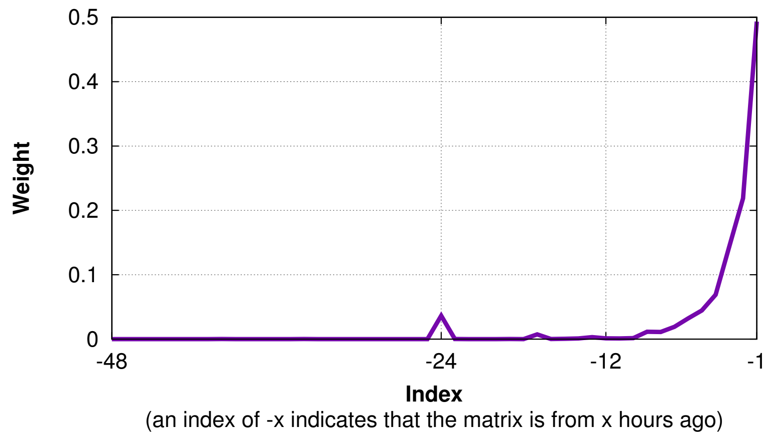


Figure 3-4: Average weights produced by Cicada's prediction algorithm. These values represent the average of the final weight values over each application in the HPCS dataset.

Intuition

Our intuition when approaching the problem of predicting traffic is that the traffic matrix M_n should depend most heavily on the most recent traffic matrices (M_{n-1}, M_{n-2}, etc.), as well as on traffic matrices from similar time periods on previous days (M_{n-24}, M_{n-48}, etc.).

If our intuition were correct, Cicada's prediction algorithm will naturally result in higher weights for these matrices. After making the predictions for each application in our dataset, we took the weights for each application, and calculated the average weight values over our entire dataset. These average weights, using the negative indexing discussed above, are plotted in Figure 3-4 (the x axis is limited to the two most recent days of data). The twelve most recent hours of traffic are weighted heavily, and there is also a spike at 24 hours earlier. Weights are vanishingly small prior to 24 hours earlier. In particular, we looked for a spike at the 7-day offset, expecting that some user-facing applications have weekly variations, but found none. This result indicates that, at least in our dataset, one does not need weeks' worth of data to make reliable predictions; a much smaller amount of data suffices.

3.3.4 Alternate Prediction Algorithms

We tested Cicada's prediction algorithm against two other algorithms: a machine learning algorithm based on linear regression, and a simple exponentially-weighted moving average (EWMA). Though neither worked as well as Cicada's expert-tracking algorithm, we describe both of them here to illuminate why the expert-tracking algorithm *does* work well.

Though Cicada's prediction scheme is similar in spirit to ISP traffic engineering, we do not explicitly compare it to any traffic engineering work. The closest related work, COPE [94], actually solves a different problem than Cicada: to predict the best *routing* over a particular *space* of traffic predictions; namely, the convex hull of previous observed traffic matrices. In COPE, there is no notion of which traffic matrix in the space is the best prediction, but rather a notion of which routing would perform the best over this space.

Linear Regression-based Algorithm

Algorithm 2 provides pseudocode of the first alternate prediction algorithm. At a high level, a prediction between tasks i and j is made by finding "relevant" historical data between i and j, finding the function f that best maps a previous epoch's data to the next epoch's, and using f to make a prediction for the current time. In the case of average-demand predictions, this prediction is the number of bytes it expects will be transferred between these two tasks in the next time period; in the case of peak-demand predictions, it is an estimate of the peak bandwidth in the next hour.

Algorithm 2 An alternate, linear regression-based prediction algorithm, for one time epoch and mean-demand predictions

1: **for** $\langle i, j \rangle \in$ task pairs **do**
2: $F_c = \langle$current time of day, current hour$\rangle$.
3: H_c = historical data on path $i \rightsquigarrow j$ under conditions similar to those in F_c.
4: f = result of linear regression on the data in H_c. f is the best linear mapping from the number of bytes in one epoch to the number of bytes in the next epoch, of the data in H_c.
5: b = number of bytes sent from i to j in the past hour
6: $M_{ij} = f(b)$

Output: M, the prediction matrix for all task pairs

Finding feature vectors: F is the set of features that describe the network conditions. We used F = {hour of day, day of week}. Line 2 of Algorithm 2 gets the values of F at the current time.

Finding similar conditions: Line 3 of Algorithm 2 finds the most relevant historical data to predict the amount of traffic that i will send to j in the next epoch. But what data is "relevant"? Given F_C, we consider the relevant historical data to be data from this task pair, under similar conditions as in F_C; i.e., similar in time-of-day and day-of-week.

To determine similarity, we first group all data between i and j by hour. For each hour h', we get a distribution of points, all collected from i to j within the hour h' (but perhaps on different days of the week). We are interested in the distribution d of data from the tuple $\langle i, j, h \rangle$, and how it compares to each of the other distributions d'. For each d', we run a Mann-Whitney U-test between d' and d (similar to the method in [59] for comparing road-traffic delay distributions). If the p-value of this test is above a particular threshold,[2] we consider the two distributions to be similar. We return data from every $\langle i, j, hour \rangle$ tuple such that its distribution d' was similar to d.

Finding f: Lines 4–6 of Algorithm 2 determine a function f and use it to make a prediction for tasks i and j. This function captures the notion that the number of bytes that i will transfer to j in the next epoch should be related to two things: the number of bytes b that i transferred to j in the previous epoch, and the current network conditions (in some cases, it could also be related to the number of bytes that i received from other tasks; we do not model this). H_C contains data under similar network conditions as the current ones, but does not necessarily contain data from task pairs that transferred b bytes in the previous epoch. For this reason, instead of making a decision tree out of the data in H_C as in [59], we use linear regression to calculate the linear function f that best maps the number of bytes a task pair

[2] In our implementation, we use a threshold value of $p = .75$.

transferred in the previous epoch to the number of bytes the same pair transferred in the next epoch. Once we have calculated f, the algorithm can make a prediction for $\langle i, j \rangle$. Since f is calculated on a per-task-pair basis, it accounts for spatial variation.

Extending this algorithm to predict peak demand: To extend this algorithm to predict peak demand, we have f map from the maximum number of bytes transferred in a δ-second period in the previous epoch to a prediction for the next epoch. We found that this approach worked better than an approach where f maps the standard deviation of a pair's bandwidth in one epoch to the next, and uses $\mu + k * \sigma$ as an estimate for the peak bandwidth (where k is a small positive constant).

Exponentially-weighted Moving Average Algorithm

We also tested Cicada's prediction algorithm against a simple exponentially-weight moving average (EWMA) algorithm. Here,

$$\begin{aligned}\hat{M}_{n+1} &= \alpha \cdot M_n + (1-\alpha) \cdot \hat{M}_n \\ &= \alpha \cdot M_n + \alpha(1-\alpha) \cdot M_{n-1} + \alpha(1-\alpha)^2 \cdot M_{n-2} + \ldots + \alpha(1-\alpha)^{n-1} \cdot M_1\end{aligned}$$

In the terminology of the expert-tracking algorithm, this is equivalent to

$$\hat{M}_{n+1} = \sum_{i=1}^{n} w_i(n) \cdot M_i,$$

where

$$w_i(n) = \alpha(1-\alpha)^{n-i}$$

We used $\alpha = .488$, which corresponds to the α value that best fits the graph in Figure 3-4.

Comparison

In Section §3.5.4, we present a detailed evaluation of Cicada's expert-tracking algorithm against the two alternative prediction algorithms. Here, we give a high-level comparison of the three.

All three algorithms capture the intuition that the previous epoch of traffic dictates, in large part, the next epoch of traffic. As shown in Figure 3-4, the expert-tracking algorithm assigns the largest weight to matrix M_n when predicting $\hat{M}_{n+1}$. By definition, the EWMA algorithm assigns the highest weight to M_n. The linear-regression algorithm predictions $\hat{M}_{n+1}$ as a function of M_n.

Expert-tracking vs. EWMA: The expert-tracking and EWMA algorithm both assign weights to previous matrices, but the EWMA algorithm is less flexible. It only

allows for a particular decreasing relationship between $w_i(n)$ and $w_{i-1}(n)$; the expert-tracking algorithm lets the weights be independent, modulo the constraint that they all sum to 1. This independence permits the spike seen in Figure 3-4.

Expert-tracking vs. linear-regression: Via its notion of "similar conditions", the linear-regression algorithm allows for the possibility that other epochs besides the most recent may substantially influence the prediction of $\hat{M}_{n+1}$. A key difference is that the expert-tracking algorithm updates its weights based on prediction errors, and so tries to correct past mistakes. The linear-regression algorithm does no such thing; its notion of similarity is defined a priori, and is not updated.

3.4 Comparing Cicada to VOC

Evaluating any prediction algorithm can be challenging; as we must expect that predictions will be imperfect at times, it does not make sense to compare our predictions only against an oracle. Instead, we also compare Cicada against the current state-of-the-art in terms of predicting cloud traffic demands.

Though there has been little to no work on making predictions for cloud traffic (see §2.4.1), we believe that the VOC model [20] represents the state of the art in spatially-varying cloud-bandwidth reservations; hence, our trace-based evaluation of Cicada, uses a VOC-style system as the baseline. In [20], the authors use VOCs to make bandwidth reservations for tenants, allowing different levels of bandwidth between different groups of VMs. Although their system was *not* developed for making bandwidth predictions, we can interpret its bandwidth reservations as predictions of the amount of bandwidth that will be used between different virtual machines.

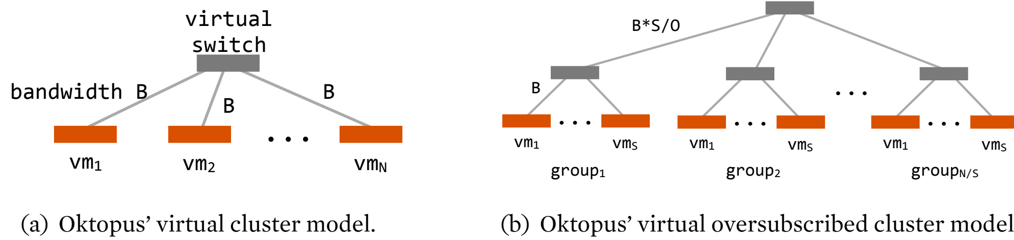
We compare a VOC-style system to Cicada, finding that Cicada can accurately predict the parameters for a VOC-style system.

3.4.1 VOC-Model

Oktopus allows tenants to make bandwidth reservations among a set of virtual machines. Their implementation utilizes "virtual networks". These virtual networks present various virtual network abstractions to the tenants, but allow the provider to use any topology. For example, consider Figure 3-5(a), where the N virtual machines are connected via a "virtual switch". From a tenant's point of view, the N machines have a maximum sending rate of $N \cdot B$ in general, but a maximum rate of B if they are all sending to the same virtual machine; just as if they were all connected to one switch with B-rate links. From the provider's point of view, however, the virtual machines need not be physically connected in this way; the provider may use whatever topology he likes to enforce this abstraction.

To make the most basic reservation, the tenant starts by supplying two parameters:

1. N, the number of virtual machines required



(a) Oktopus' virtual cluster model.

(b) Oktopus' virtual oversubscribed cluster model.

2. B, the amount of bandwidth to be reserved between a virtual machine and the "virtual switch" connecting it to the other machines

Figure 3-5(a) displays this type of reservation. The authors note that this abstraction does not reflect many application traffic patterns (citing [37] and [61]), and that instead, a "virtual oversubscribed cluster" (VOC) model is more appropriate. This model allows groups of virtual machines, with oversubscribed communication between them, as depicted in Figure 3-5(b). The VOC model requires two additional parameters:

1. S, the size of each group of virtual machines
2. O, the oversubscription factor (between any pair of groups, VOC provides bandwidth B/O)

VOC is designed to reflect not only what the authors believe to be a typical application structure—lots of communication within small groups of virtual machines, but not a lot of communication across groups—but also a typical physical datacenter topology, where Top-of-Rack (ToR) switches have high capacity, but the aggregation layer that connects the ToRs is oversubscribed.

3.4.2 VOC-style Predictions

Oktopus itself is designed to place the VMs on the network such that their VOC reservations are met; it is not designed to make future predictions about applications. However, we can easily extend the VOC model to output predictions.

To do so, we interpret the B and B/O bandwidths as hose-model predictions (see §3.3.2): VOC predicts that a VM in a particular group will use bandwidth B to send to VMs within its cluster, and groups will use bandwidth B/O to send to other groups. These predictions are not pipe-model predictions, as is our preferred model for Cicada, but we are still able to compare the accuracy of the two systems on a tenant-by-tenant basis.

The VOC model is fairly rigid, enforcing groups of the same size, and the same bandwidth reservations throughout. In §3.4.4, we extend its model, to allow for a fairer comparison to Cicada.

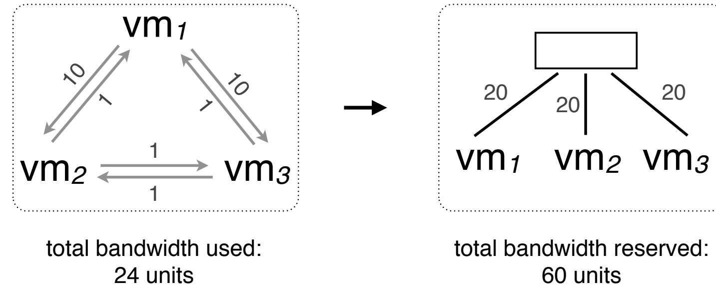


Figure 3-5: An example where VOC is inefficient. Because the virtual machines within the group do not send similar amounts of data, more bandwidth is reserved than is used.

3.4.3 Inefficiencies of VOC

The Oktopus paper [20] showed that the VOC model gives providers more flexibility than a clique abstraction, which provides a static, uniform bandwidth between all pairs of VMs. However, we give an example to illustrate how even the VOC model can limit provider flexibility for certain applications, due to over-allocating bandwidth both within groups and across groups.

Consider a group of three VMs as in Figure 3-5. Suppose that VM_1 sends 20 units total to VM_2 and VM_3. Because of this, B must be at least 20 for each VM in the group. However, if VM_2 and VM_3 send fewer than 20 units total, this value of B will over-allocate bandwidth. In practical terms, if each VM is on a distinct server, the VOC model requires allocating 20 units of each server's NIC output bandwidth to this tenant, even though VM_2 and VM_3 only need two NIC-output units. A similar pattern can exist across groups, where one group requires more total bandwidth than the others.

VOC's over-allocation of bandwidth in these scenarios stems from an assumption that VMs within a group behave similarly (sending the same amount of data to the rest of the group), as well as a corresponding assumption across groups.

3.4.4 VOC Parameter Selection

In order to compare Cicada and VOC, we need to know what the appropriate values for B, N, S, and O are. Oktopus assumes that the tenants will provide these input parameters. Except in the case of N—the total number of virtual machines—we believe this is a large assumption. Cicada's prediction module was motivated by the fact that customers often do not know their application's bandwidth demands; although VOC presents a coarser model, we do not expect customers to know the appropriate VOC parameters either.

To that end, we designed a heuristic, detailed below, to determine the appropriate

input parameters given a particular application. Additionally, we allow for the possibility of variably-sized groups, an extension which is mentioned in [20] (§3.2) but not detailed. Our heuristic outputs the following:

- B, the amount of bandwidth available within a cluster
- O, the oversubscription factor
- $\{G_i\}$, a set of groups. Rather than requiring N/S groups of size S, we allow for a variable number of groups, each of which can be a different size. We believe that this will allow VOC to adapt to application patterns that do not fit into its strict original form.

Our heuristic works by starting with an initial configuration of groups, with each VM is in its own group. The heuristic merges the groups that have the highest bandwidth between them for a new configuration, and iterates on this configuration in the same manner, until the new configuration has more *wasted bandwidth* than the previous configuration. We define "wasted bandwidth" as bandwidth that VOC reserved for the tenant, but was not used by the application (we can determine what bandwidth was used by an application based on its measurement data).

Finding B and O for a particular configuration is easy, since Cicada collects measurement data. Our heuristic selects the B and O that minimize wasted bandwidth while never under-allocating a path. B, then, is the maximum of any hose within one group, and O is the maximum of any hose across groups, given the previous historical data.

3.5 Evaluation

We evaluated Cicada's prediction method on the HPCS dataset, described in §2.5. We tested two hypotheses: first, that Cicada can determine when one of its predictions is reliable or not, and second, that Cicada can accurately predict a tenant's future bandwidth requirements. Overall, we found that the reliability of Cicada's predictions was correlated with the size of a tenant and the frequency of under-predictions, and that Cicada's predictions were accurate for both average and peak traffic, and that Cicada's predictions were more accurate than predictions based on VOC, decreasing the relative per-tenant error by 90% in both the average-bandwidth case and the peak-bandwidth case.

As described in §3.3, Cicada's prediction algorithm can make fine-grained predictions for an application's tasks. Due to the nature of our dataset (§2.5), in this evaluation, we consider all processes running on one virtual machine to constitute one task. For clarity, we use the term "virtual machine" (or VM) throughout this evaluation, rather than "task".

Algorithm 3 VOC parameter selection.

```
1: groups_p = list of groups, initialized to one group per VM
2: G_v = groups_p // last valid configuration
3: while True do
4:    G' = mergeLargestClusters(groups_p)
5:    if G' = groups_p then
6:       break
7:    B, O, w = getParamsAndWastedBandwidth(G)
8:    if O < 1 then
9:       // indicates an invalid configuration
10:      groups_p = G'
11:      continue
12:   B_v, O_v, w_v = getParamsAndWastedBandwidth(G_v)
13:   if w > w_v then
14:      break
15:   else
16:      groups_p = G_v = G'
17: B, O = findParameters(G_v)
Output: B, O, G_v
```

We observed very little evidence of VM flexing in our dataset (where the number of a tenant's VMs changes over time). Flexing would typically manifest as over-prediction errors (making predictions for VM pairs that used to exist, but no longer do because of flexing). To eliminate this mistake, we eliminated any predictions where the ground truth data was zero, but found that it did not appreciably change the results. For this reason, we do not present separate results in this section, and simply report the results over all of the data.

3.5.1 Quantifying Prediction Accuracy

To quantify the accuracy of a prediction, we compare the predicted values to the ground truth values, using the relative ℓ^2-norm error. For two vectors $\vec{M}_i$ and $\vec{M}_n$, the relative ℓ^2-norm error is

$$L(i,n) \quad = \quad E_{\ell^2} = \frac{\|\vec{M}_i - \vec{M}_n\|}{\|\vec{M}_n\|}.$$

That is, the norm of the individual errors over the Euclidean norm of the observed data (the square-root of the sum of the squares of the components of the vector). To obtain $\vec{M}_i$, $\vec{M}_n$, we simply treat our application demand matrices as vectors. This error was also described in §3.3.3.

For Cicada, the prediction and ground-truth vectors are of length $N^2 - N$, because Cicada makes predictions between each pair of distinct VMs. In a VOC-style system,

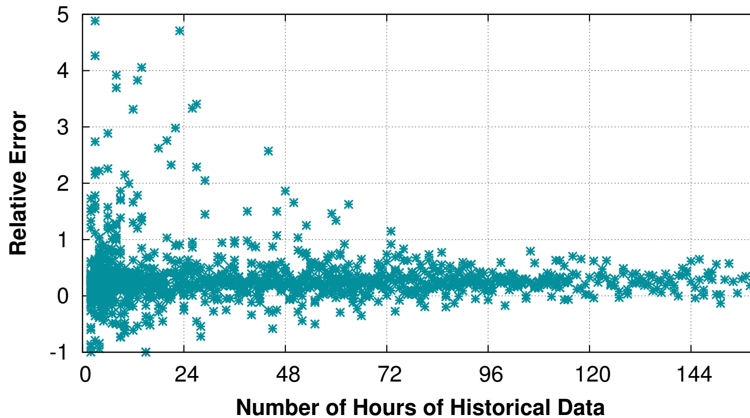


Figure 3-6: Relative error vs. history

there are fewer predictions: one prediction for each of the N VMs to the other VMs in its group, and one prediction for each group to the other groups. Regardless of the number of total predictions, we get one error value per tenant, per time interval.

In addition to the relative ℓ^2-norm error, we also present the per-tenant relative error. This metric is simply the sum of the prediction errors divided by the total amount of ground-truth data. Unlike the relative ℓ^2-norm error, this metric discriminates between over-prediction and under-prediction, since the latter can be more disruptive to application performance. Because it does not use vector norms, the per-tenant relative error makes it easier for us to see if either system has substantially under-predicted for a tenant. However, it is possible that under- and over-prediction errors for different VM pairs could cancel out in the relative error; this type of cancellation does not happen in the relative ℓ^2-norm error.

3.5.2 Determining Whether Predictions Are Reliable

In our dataset, we found that Cicada could not reliably make a correct prediction for tenants with few VMs. In all of the results that follow, we eliminate tenants with fewer than five VMs. This elimination is in line with the fact that Cicada is meant for tenants with network-heavy workloads. It is possible that on other datasets, the precise number of tenants below which Cicada cannot reliably make predictions will differ. Additionally, for tenants that started out with a series of under-predictions, Cicada's prediction algorithm rarely recovered. For this reason, we also consider tenants with more than fifteen under-predictions to be unreliable, and do not continue making predictions for them (we do, however, include results from all predictions for that tenant up to that point).

Interestingly, we found that Cicada could often make accurate predictions even with very little historical data. Figure 3-6 shows the relative per-tenant error as a function of the amount of historical data. Though there is a clear correlation between

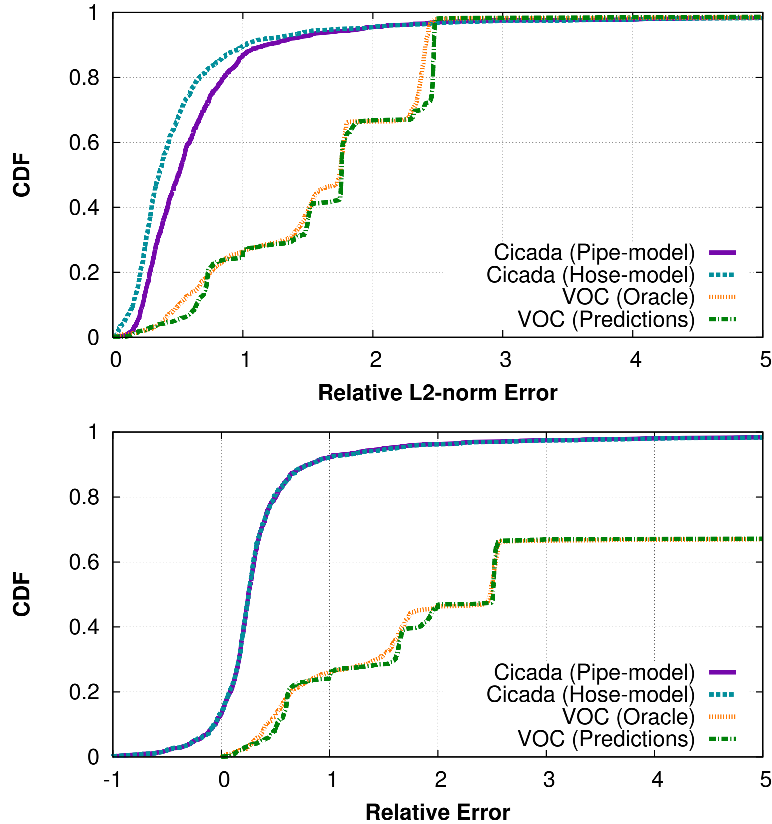


Figure 3-7: Prediction errors for average demand. For average demand, straightforward math shows that Cicada's hose-model and pipe-model relative errors will always be identical; see below.

relative error and the amount of historical data, it is not necessary to have many hours of data in order to make accurate predictions.

3.5.3 Prediction Errors for Cicada's Expert-Tracking Algorithm

To compare the accuracy of Cicada and a VOC-style model, we make predictions for one-hour intervals. We allow both types of predictions to change over time; that is, the VOC configuration for a particular hour need not be the same as the configuration in the next hour. This is an extension from the original Oktopus paper [20], and improves VOC's performance in our comparison.

We evaluate the VOC-style model using both predicted parameters and "oracle-generated" parameters (i.e., with perfect hindsight). For the oracle parameters, we determine the VOC clusters for a prediction interval using the ground-truth data from that interval. This method allows us to select the absolute best values for B and O for

that interval, as well as the best configuration. Thus, any "error" in the VOC-oracle results comes from the constraints of the model itself, rather than from an error in the predictions.

Average-demand Predictions

Figure 3-7 shows the results for average-demand prediction. Straightforward math shows that the relative errors for the pipe- and hose-model predictions are exactly the same:

> Let $\hat{p}(i \rightsquigarrow j)$ be the pipe-model predicted demand between VM_i and VM_j, and let $p(i \rightsquigarrow j)$ be the ground-truth demand between VM_i and VM_j. The relative error for all of the VMs in the network is
>
> $$\sum_{i \in VMs} \sum_{j \in VMs} \frac{\hat{p}(i,j) - p(i,j)}{p(i,j)}$$
>
> Now, let $\hat{h}(i)$ be Cicada's hose-model predicted demand out of i, and let $h(i)$ be the ground-truth demand out of i. Cicada's hose-model predictions (see §3.3.2 are equivalent to
>
> $$\begin{aligned} \hat{h}(i) &= \sum_{j \in VMs} \hat{p}(i,j) \\ h(i) &= \sum_{j \in VMs} p(i,j). \end{aligned}$$
>
> The relative error for the hose-model, then, is
>
> $$\begin{aligned} \sum_{i \in VMs} \frac{\hat{h}(i) - h(i)}{h(i)} &= \sum_{i \in VMs} \left(\frac{\sum_{j \in VMs} \hat{p}(i,j) - \sum_{j \in VMs} p(i,j)}{\sum_{j \in VMs} p(i,j)} \right) \\ &= \sum_{i \in VMs} \sum_{j \in VMs} \frac{\hat{p}(i,j) - p(i,j)}{p(i,j)}, \end{aligned}$$
>
> which is equivalent to the pipe-model relative error. This same calculation does not apply for peak errors. Here, a hose-model prediction is a sum of pipe-model predictions; with peak error, a hose-model prediction is a *maximum* of pipe-model predictions.

Both Cicada models have lower error than either VOC model; Cicada's pipe model decreases the error by 90% compared to the VOC oracle model (comparison against the predictive VOC model, as well as between VOC and Cicada's hose model, yields a similar improvement). The ℓ_2-norm error decreases by 71%. The errors of the VOC model using predictions closely track those from the VOC-oracle model, indicating

that Cicada's prediction algorithm can generate accurate predictions for a system using VOC. This result indicates that it is the lack of expressiveness in VOC's model that prevents it from making as accurate predictions as Cicada.

In terms of per-tenant relative error, Cicada occasionally under-predicts, whereas neither VOC model does. Under-prediction could be worse than over-prediction, as it means that an application's performance could be reduced. The effect of under-provisioning can be lessened by scaling predictions by an additive or multiplicative factor, though this risks over-prediction. In the results presented here, we have scaled the predictions by $1.25\times$. In addition to lessening the effects of under-provisioning, this scaling also allows the bank-of-experts algorithm to make a prediction that is *greater* than any of the previous matrices. We were unable to determine a systematic way to remove the remaining under-predictions, but speculate that altering the loss function to penalize under-prediction more heavily than over-predictions may help; we leave this to future work.

Peak-demand Predictions

Because our data collection samples samples over five-minute intervals, for our peak-demand evaluation, we use $\delta \geq 300$ seconds. Real applications might require guarantees with $\delta < 1$ second; an implementation of Cicada can support these small δ values (§5.4).

Figure 3-8 compares peak predictions for $\delta = 5$ minutes (also scaled by $1.25x$ as above). As with the average-demand predictions, Cicada's prediction errors are generally lower, decreasing the median error again by 90% from the VOC oracle model (the median ℓ_2-norm error decreases by 80%). As before, Cicada does under-predict more frequently than either VOC model, but overall, the results for peak-demand predictions show that Cicada performs well even for non-average-case traffic.

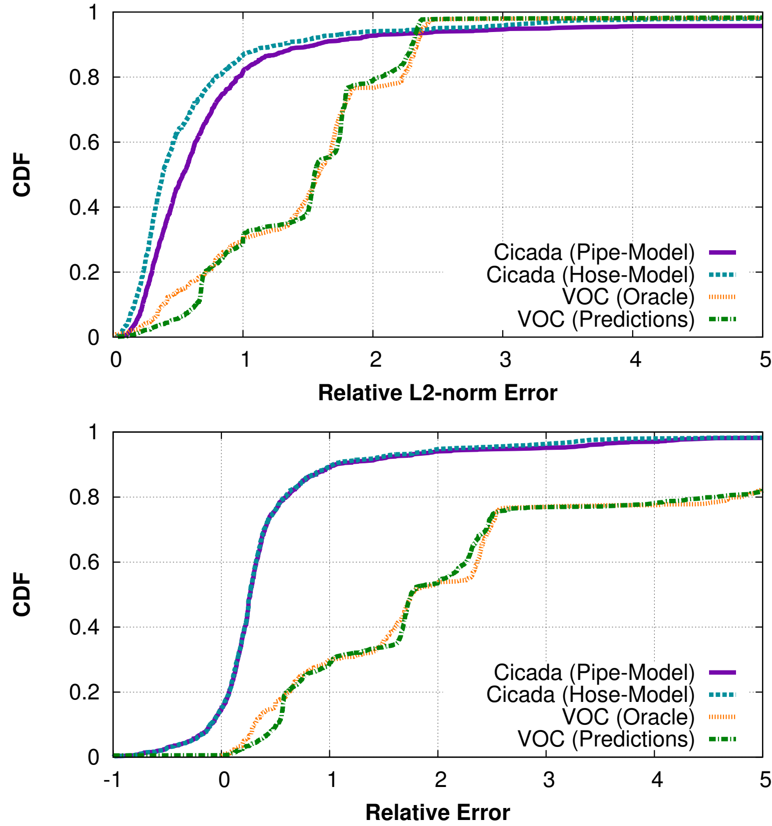


Figure 3-8: Prediction errors for peak demand.

3.5.4 Prediction Errors for Alternate Algorithms

EWMA-based Algorithm

On our dataset, the EWMA algorithm (detailed in §3.3.4, with results in Figures 3-9 and 3-10) performed comparably to the experts-tracking algorithm once we increased each prediction by a small multiplicative constant (1.25) to decrease the under-prediction (median errors for average demand: .24 (relative) and .47 (ℓ^2); for peak demand: .25 (relative) and .50 (ℓ^2). This result was somewhat surprising, as Cicada's expert-tracking algorithm is more sophisticated than an EWMA. However, given that the resulting weights of the experts-tracking algorithm (Figure 3-4) are (to first order) exponentially decreasing, it makes sense that the EWMA algorithm should perform similarly on our data. Still, there are two weaknesses of the EWMA algorithm compared to the expert-tracking algorithm:

Tuning α: The EWMA requires a single parameter, α, to determine how much weight to give to new samples. In our evaluations, we chose α based on the results of the

expert-tracking algorithm. How hard is it to select α in general?

To answer this question, we evaluated the EWMA algorithm with randomly sampled values for α. On our dataset, the EWMA algorithm is fairly robust, but not perfect; its relative and ℓ^2 errors increased with $\alpha <\approx .1$ and $\alpha >\approx .9$. For other datasets, these values may change. In all cases, there is some tuning involved in using EWMA-based algorithm.

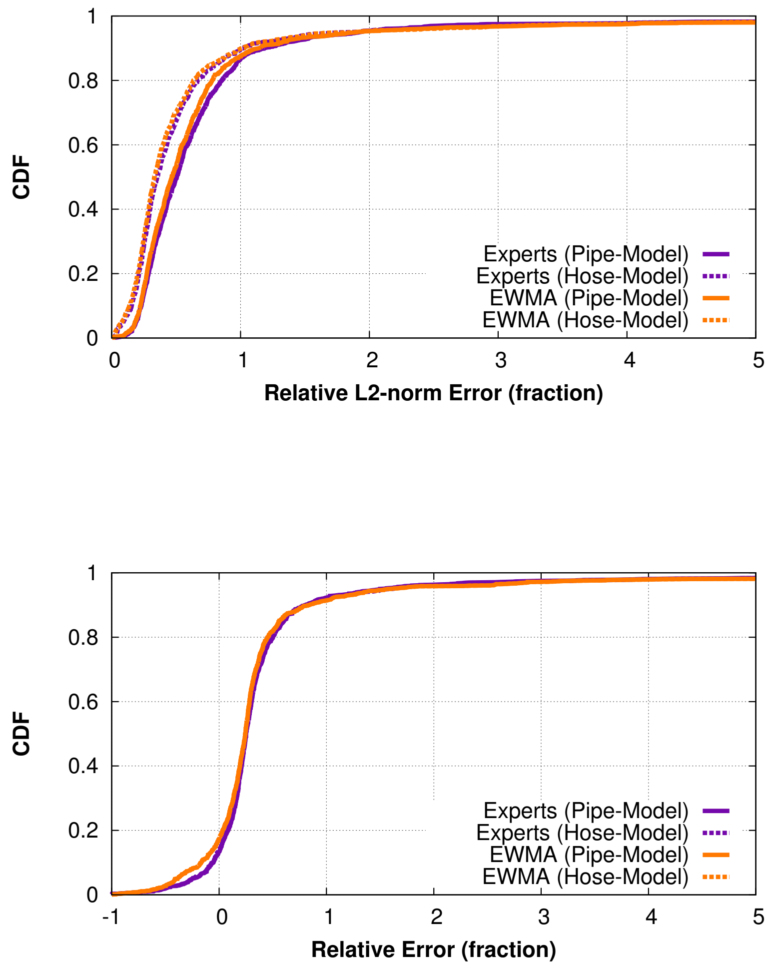


Figure 3-9: Prediction errors for average-demand using the EWMA algorithm.

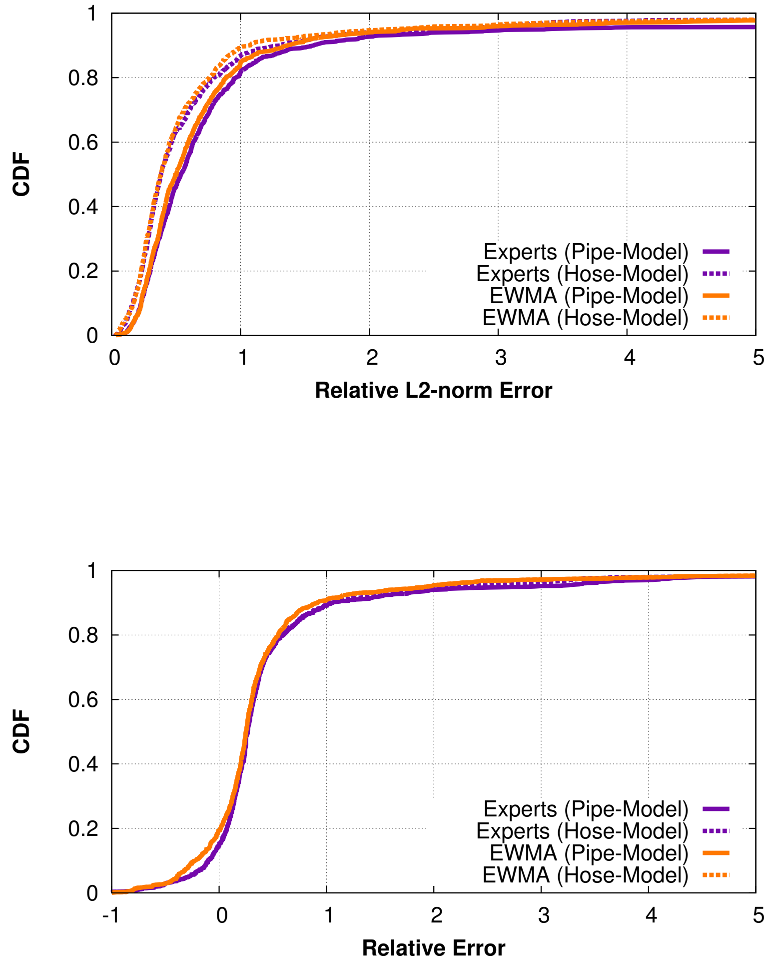


Figure 3-10: Prediction errors for peak-demand using the EWMA algorithm.

Handling Diurnal Applications: One application that an EWMA will not perform as well on is a heavily diurnal application. Here, the most recent is typically not the best predictor; rather, the matrix from 24 hours ago is.

As mentioned in §2.5.7, we believe that our dataset may not have a large number of user-facing applications, which are a type of applications that exhibit strong diurnality. To test Cicada's performance on these types of applications, we simulated 14 days of data with a strong diurnal pattern, where the traffic demands followed a sinusoidal curve throughout the day (with random jitter added). We tested average demand

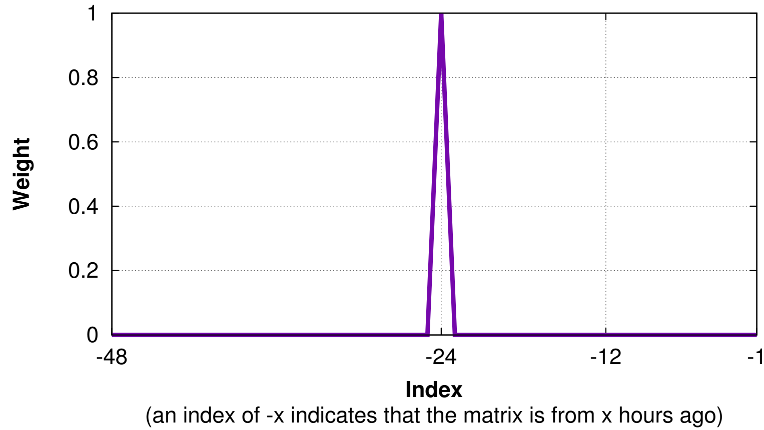


Figure 3-11: Weights produced by Cicada's expert-tracking algorithm when run on a strongly diurnal application. The algorithm assigns almost all of the weight to the datapoint 24 hours earlier.

predictions.

In this experiment, the median ℓ^2 and relative errors of the EWMA algorithm remained at roughly the same values as before (.46 and .23, respectively), whereas the median ℓ^2 error of the expert-tracking error was much better (.29 compared to .46; the relative error remained the same). This result remained consistent across a variety of different sinusoids.

The reason the expert-tracking algorithm performs better in this case is shown in Figure 3-11. Here, we show the weights of the expert-tracking algorithm after testing it on a diurnal application. As expected, the algorithm assigns almost all of the weight to the datapoint 24 hours earlier. The EWMA algorithm cannot adapt to this traffic pattern.

Linear-Regression Algorithm

Using the linear regression-based algorithm (detailed in §3.3.4; results in Figure 3-12), under-prediction for average demand occurred most frequently in tenants with little relevant historical data ($|H_C| < 6000$). The amount of under-prediction could be decreased by multiplying each prediction for such tenants by a small constant $k \leq 1.2$, without causing any tenants to be substantially over-predicted. Using larger constants leads to even less under-prediction, but increases the chance of over-prediction in some cases.

The prediction error for the average demand case is comparable to the error in

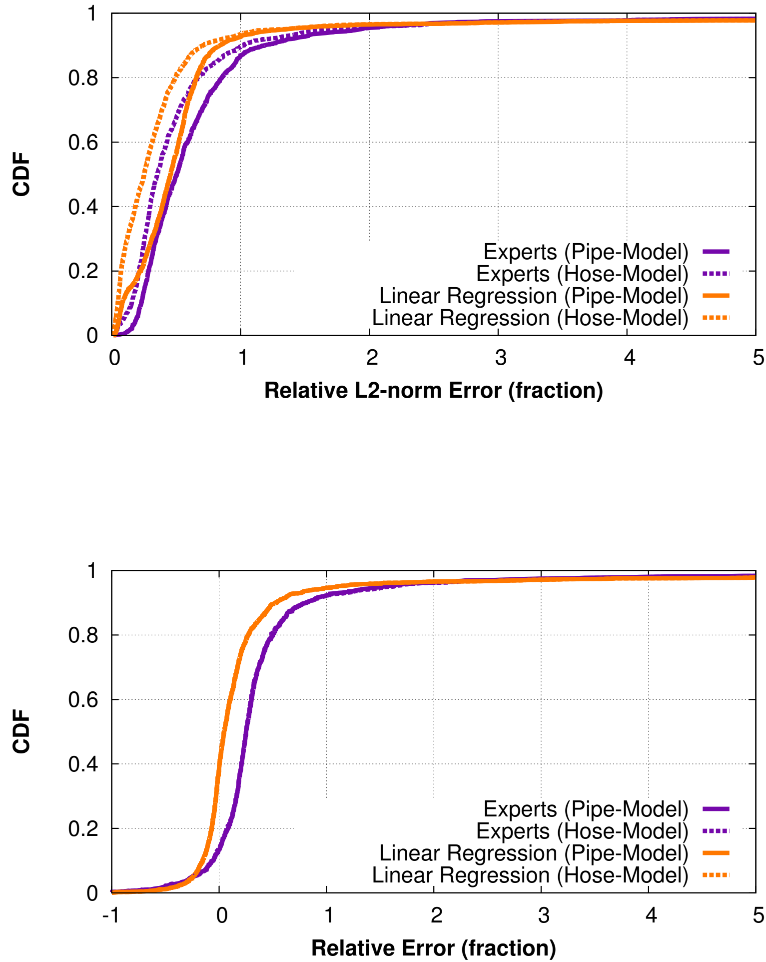
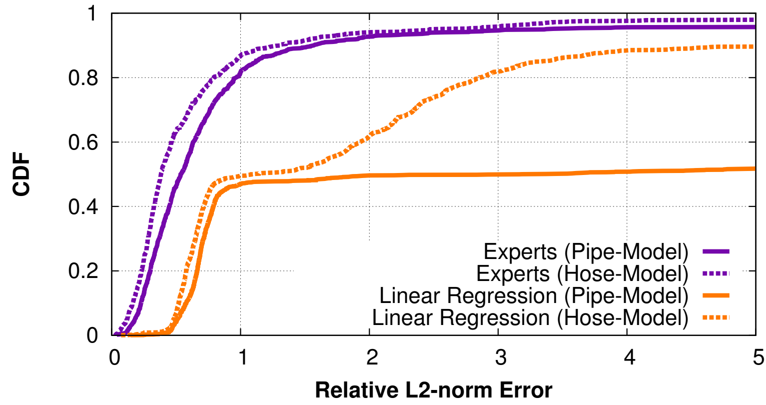


Figure 3-12: Prediction errors for average demand using the linear regression-based algorithm.

Cicada's experts-tracking algorithm (Figure 3-7).

For peak demand predictions (Figure 3-13), we found that without reducing the maximum predictions by a multiplicative constant they had a tendency to severely over-predict the peak demand; a scale factor of 0.2 improved accuracy. Increasing each prediction by a small additive constant improved the under-prediction errors, much like multiplying by a constant did in the average demand case. However, Cicada's peak predictions with the linear regression algorithm are quite bad. We discuss



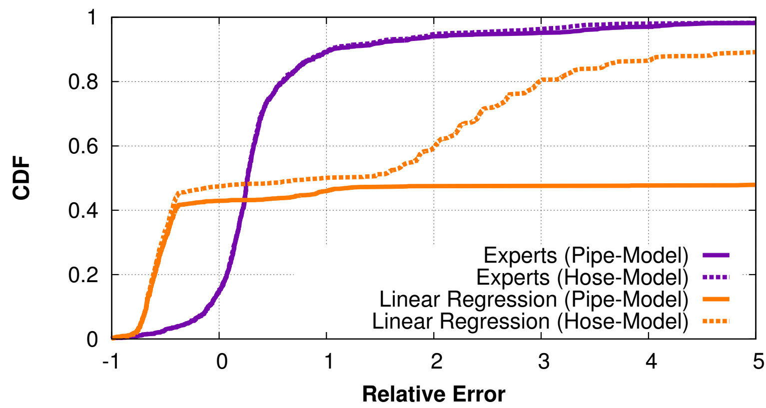


Figure 3-13: Prediction errors for peak demand using the linear regression-based algorithm.

the reason for this below, in addition to pointing out a second weakness of the linear-regression algorithm.

Correcting Errors: The expert-tracking algorithm updates its weights in response to errors in previous prediction. The linear-regression algorithm does no such thing; it has an a priori notion of "similarity", and this is not updated. We believe this to be a possible reason why the peak predictions are bad: the a priori notion of similarity

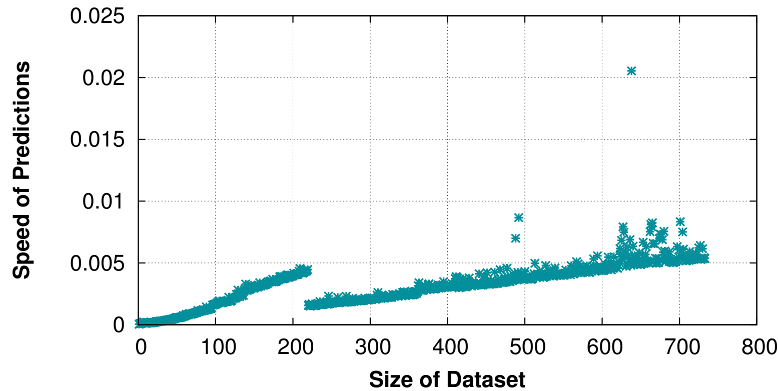


Figure 3-14: The speed of Cicada's prediction algorithm vs. the amount of history available. As the amount of history grows, so does the time it takes to make a prediction. In all but one case, Cicada takes fewer than 10 milliseconds to make a prediction.

that the linear-regression algorithm has does not work well for predicting peak-demand, and the algorithm has no way to correct its mistakes. It is possible that the performance of this algorithm would improve with online-updating of the similarity constraints, but this brings us to a second weakness of the linear-regression algorithm.

Performance: The linear-regression algorithm performed orders of magnitude more slowly than the expert-tracking (and EWMA-based) algorithm(s), as it requires going through all of the past data to find the similar data. Due to restricted access to the dataset, we were unable to test the linear-regression algorithm using a pruned history (one that used only the most recent 48 hours of data, e.g.), but we note that the expert-tracking and EWMA algorithms had no trouble utilizing the entire past history of an application.

3.5.5 Speed of Predictions

Cicada is able to make a prediction using the expert-tracking algorithm very quickly. The amount of time it takes to make a prediction is affected predominantly by the amount of history used.

Figure 3-14 plots the mean prediction speed against the amount of history used. In all cases, the mean prediction speed is fewer than 10 milliseconds in all but one case (fewer than 25 milliseconds in all cases), and in the majority of cases, fewer than 5 milliseconds. Should tenants run longer than those in our dataset—i.e., have more

than hundreds of hours of history—the history could be pruned. In our dataset, Cicada is able to make accurate predictions for applications using only 48 hours of history (see Figure 3-6).

Due to the restrictions on our dataset, we are unable to publish specific results as to how the size of the tenant affects the prediction speed. However, we found tenant size to have a smaller effect on prediction speed than the amount of history did.

3.5.6 Summary

To summarize, Cicada's expert-tracking prediction algorithm outperforms predictions generated from a VOC model. Cicada's predictions decreased the relative per-tenant error by 90% in both the average-demand case and the peak-demand case compared to the VOC-based algorithm.

We believe Cicada's expert-tracking algorithm performs well for the following reasons:

- It allows for a large amount of spatial- and temporal-variability. Systems such as VOC do not.
- It takes into account an appropriate amount of past history; simply using the most recent datapoint to make a prediction performed poorly (this is equivalent to an EWMA algorithm with $\alpha = 1$, and the EWMA algorithm performed poorly for $\alpha > .9$).
- It corrects past mistakes via updates to the weights based on the relative ℓ^2-norm error.
- It handles a general case of applications, adapting well to diurnal applications, because the weights are not fixed beforehand.

Furthermore, Cicada's algorithm requires no parameter tuning, and makes predictions quickly (no more than ten milliseconds in almost all cases).

Though Cicada's prediction algorithm performs well on our dataset and against alternative algorithms, we make no claim that it is the *best* prediction algorithm. Rather, we have shown that accurate traffic prediction is possible and worthwhile in the context of cloud computing.

3.6 Conclusion

This chapter described the design and evaluation of Cicada's traffic prediction algorithm. As discussed in §3.5, Cicada's expert-tracking prediction algorithm is able to make fine-grained temporally- and spatially-varying predictions and to handle a general case of applications. Using traces from HP Cloud Services (§2.5), we showed

that Cicada accurately predicts tenant bandwidth needs when compared to a variety of alternative algorithms.

Perhaps more importantly than the algorithm itself, we have shown that fine-grained workload prediction is possible for cloud applications. Existing systems assume that the customer understands an application's traffic pattern beforehand; we believe that customers typically have *no* understanding of their application's demands. Cicada now makes it feasible for cloud systems to understand and adapt to the workload patterns of their applications.

To that end, the rest of this dissertation shows how Cicada's predictions can be used to improve application performance. Chapter 4 details an application placement algorithm that minimizes application completion time. Chapter 5 shows how providers can turn Cicada's predictions into bandwidth guarantees for their customers, which can improve customer's performance (by guaranteeing them the bandwidth they need rather than risking an under-provisioned network) and encourage companies that need to meet their own SLAs to utilize cloud services.

Chapter 4

Application Placement

4.1 Introduction

In the previous chapter, we discussed how to predict an application's traffic demands. This chapter focuses on what we can do with an application once we know that information. Specifically, we are interested in placing applications on cloud networks. This placement can be done either by the cloud provider, to place applications anywhere in an availability zone, or by a customer, to place applications on her own virtual machines in the cloud (in Chapter 6, we discuss the additions that Cicada needs for a full customer-run deployment).

* * *

The placement of applications in a public cloud is particularly important with the advent of *network-intensive* applications. The performance of such applications depends not just on computational and disk resources, but also on the network resources between the machines on which they are deployed. Previous research [20, 34] has shown that the datacenter network is the limiting factor for many applications. For example, the VL2 paper reports that "the demand for bandwidth between servers inside a datacenter is growing faster than the demand for bandwidth to external hosts," and that "the network is a bottleneck to computation" with top-of-rack (ToR) switches frequently experiencing uplink utilizations above 80% [34].

Much previous work has focused on better ways to design datacenter networks to avoid hot spots and bottlenecks. In this chapter, we take a different approach and ask the converse question: given a network architecture, what is the impact of a network-aware task placement method on end-to-end performance (e.g., application completion time)? Our hypothesis is that by measuring the inter-node throughputs and bottlenecks in a datacenter network, and by understanding an application's data transfer characteristics (for instance, via the traffic prediction method in Chapter 3), it is possible to improve the performance of a mix of applications.

A concrete context for our work is a tenant with a set of network-intensive cloud applications. To run these applications, a tenant requests a set of virtual machine

(VM) instances from the cloud provider; to contain costs, the tenant is thrifty about how many VMs or instances it requests. When it gets access to the VMs, the tenant now has a decision to make: how should the different applications be placed on these VMs to maximize their performance? A common tenant goal is to minimize the run-time of the application. An alternative context is a cloud provider aiming to place tenant applications across a single availability zone (or even an entire datacenter) so as to improve network utilization.

For network-intensive applications, an ideal solution for placement is to map an application's tasks to the VMs taking into consideration the inter-task network demands as well as the inter-VM network capacities. As in the previous chapter, we use "tasks" as an intuitive term: a task may map to a collection of processes in an application, for instance a map or reduce task during a MapReduce job. As a simple example, suppose an application has three tasks, S, A, and B, where A and B communicate often with S, but not much with each other. If we are given three VMs (in general on different physical machines) and measure the network throughputs between them to be different—say, two of them were higher than the third—then the best solution would be to place S on the VM with the highest network throughput to the other two VMs. By not taking the performance of the underlying network into account, applications can end up sending large amounts of data across slow paths, while faster, more reliable paths remain under-utilized. Our goal is to build on this insight and develop a scalable system that works well on current public clouds.

* * *

This chapter makes two contributions. The first is the design of Cicada's second module for network-aware application placement, depicted earlier in Figure 2-2, which tenants can use to place a mix of applications on a cloud infrastructure. This module utilizes three sub-systems: a component to profile the data transfer characteristics of an application (described in Chapter 3), a low-overhead measurement component to obtain inter-VM network throughputs (described in this chapter, and also Chapter 6), and an algorithm to map application tasks to VMs in order to satisfy a particular objective that is affected by the underlying network.

These sub-systems must overcome four challenges: first, inter-VM throughputs are not constant [22]; second, cloud providers often use a "hose model" to control the maximum output rate from any VM; third, any practical measurement or profiling method must not introduce much extra traffic; and fourth, placing a subset of tasks changes the network throughput available for subsequent tasks. Moreover, an optimal placement method given the achieved network throughputs and application profile is computationally intractable, so any practical approach can only be approximate. To the best of our knowledge, the problem of matching compute tasks to nodes while taking inter-VM network properties into account has not received prior attention.

The second contribution of this chapter is an evaluation of how well Cicada performs compared to other placement methods that do not consider network through-

puts or inter-task communication patterns. We collect network performance data from Amazon's EC2 and Rackspace, and use the application profiles obtained from HP Cloud Services (§2.5), to evaluate Cicada. For the specific goal of minimizing application run-time, we find that Cicada can reduce the average running time of applications by 8%–14% (maximum improvement: 61%) when applications are placed all at once, and 22%–43% (maximum improvement: 79%) when applications arrive in real-time, compared to alternative placement methods on a variety of workloads. These results validate our hypothesis that task mapping using network measurements and application communication profiles are worthwhile in practice.

4.2 Application Placement Method

4.2.1 Problem Statement

To place an application, Cicada requires two inputs: a description of the current network state, and a description of the application's future workload. For now, we focus on placing a single application; the placement of a sequence of applications follows naturally (§4.2.6).

4.2.2 Describing the Current Network State

To describe the existing network of M machines, Cicada uses two matrices:

1. $C_{M\times 1}$, the CPU constraint vector. C_m is the CPU resource available on machine m. In our formulation, $C_m =$ the number of cores on machine m.

2. $R_{M\times M}$, the network constraint matrix. R_{mn} is the throughput that *one* connection on the path from Machine m to n achieves (this connection may exist in the presence of other, background connections).

In cloud networks, C can easily be obtained when the virtual machines are launched (virtually all clouds have an option to launch machines with a user-defined level of computing power). R is more difficult to obtain; it requires Cicada to measure the network. The process of measuring is different depending on whether a customer or a provider is running Cicada.

Provider-centric Measurement

It is not too difficult for a provider to measure throughput on its own cloud network, in part because the provider can use passive measurements. For example, the provider can enable tools such as sFlow [88] or NetFlow [66] on the routers inside the datacenter, which collect statistics about traffic usage. In fact, we collected the dataset that we use in this work using sFlow (§2.5).

Alternatively, the provider can collect measurements through the hypervisors of the virtual machines. This type of measurement can provide more detailed statistics (e.g., sFlow typically subsamples packets; hypervisor-level measurement need not) as well as probe the upper bound on path capacity (passive measurements give a lower bound).

Customer-centric Measurement

If a customer is running Cicada alone (i.e., without any assistance from the provider), its measurement methodology must necessarily be different than the provider's measurement methodology. For one, customers do not have access to the switches in the cloud datacenter, nor to the hypervisors. In fact, a customer can only hope to measure the network throughput between its own virtual machines (for a customer to use Cicada, we assume that once its tenants have access to a set of machines, they can specify which machine(s) they want to run a particular portion of their application on).

In order to get a snapshot of the network, the customer must *actively* measure the network, i.e., introduce traffic into it. Passive measurement will not suffice, in general, because the customer needs to determine the maximum possible TCP throughput on the path, rather than just the observed throughput of their data. Sending a lot of measurement traffic can affect the measurements themselves, slow down systems already running on the network, and in the case of public clouds, possibly incur monetary cost to the customer. Additionally, customers must be able to infer from their measurements the behavior of the other tenants on the cloud network, who are unknown to them. This inference is necessary because the traffic from other tenants can appreciably affect the customer's traffic (§6.4.2).

Doing this type of measurement in a scalable, accurate, and fast manner is a complicated task. We describe a system, Choreo, for doing just this in Chapter 6.

4.2.3 Describing the Application's Workload

In this chapter, we consider an application to be a collection of tasks. To describe the J tasks, Cicada uses two matrices:

3. CR_J, the CPU demand vector. CR_j is the CPU demand for task j, i.e., the number of cores required for task j.

4. $B_{J \times J}$, the network demand matrix. B_{ij} is the amount of data task i needs to transfer to task j.

Notice that to specify the transfer that occurs between any two tasks, Cicada uses the *amount* of data that needs to be transferred, rather than the amount of network bandwidth needed as in [8, 20, 63]. We have chosen this formulation because network-intensive batch applications generally do not need to sustain a transfer at a particular

rate. Assuming VM v_1 has data to send to VM v_2, v_1 will send it as fast as its connection to v_2 allows. Hence, what matters is the amount of data, and not the speed of the transfer in a particular placement. Though existing work [19] for pricing datacenters advocates for specifying bandwidth, we believe specifying the total amount of data is more appropriate for optimizing application completion time.

Cicada obtains these application-workload matrices using the techniques described in the previous chapter (Chapter 3). In the terminology of that chapter, a task corresponds to the set of processes that the application ran on one virtual machine. To describe an entire application in one matrix, Cicada can simply sum up the time series of matrices generated in the previous chapter. That is,

$$B = \sum_{i=1}^{n} M_i$$

Optionally, Cicada can keep the time series of matrices, and update its placement every epoch; we discuss this further in §4.4.3. For now, we will assume that B is a description of the entire behavior of the application.

4.2.4 Placing Applications Using ILPs

To place applications, the four matrices described in the previous sections are used as an input to an integer linear program (ILP). Before describing this program, we need one more matrix:

5. $X_{J \times M}$, the task assignment matrix. $X_{jm} = 1$ if task j is placed on machine m, and 0 otherwise.

In the end, X will tell us which tasks to place on which machine.

Before we describe the ILP itself, we must first decide what objective function we want to optimize. What does it mean to improve application in the cloud? To have the application run as fast as possible? To minimize monetary cost? In fact, Cicada can operate with both of these objective functions, and others; Cicada can place applications to satisfy **any** goal that can be formulated as an optimization problem.

In this section, we describe the ILP for minimizing application completion time. Appendix A gives ILPs for other goals.

ILP for Minimizing Completion Time

We can model the particular problem of minimizing application completion time as a quadratic optimization problem. In our formulation, we assume that there is no unknown cross-traffic in the network. We show in §6.5.2 that this assumption generally holds in the Amazon EC2 and Rackspace networks, and also discuss how we could change our formulation if the assumption were not true.

A placement of the J tasks onto the M machines is an X such that:

$$\sum_{m=1}^{M} X_{jm} = 1, \forall j \in [1, J]$$

That is, each task must be placed on exactly one machine. In addition, the placement of tasks on each machine must obey CPU constraints, i.e.,

$$\sum_{j=1}^{J} CR_j \cdot X_{jm} \leq C_m, \forall m \in [1, M]$$

Given a particular placement of tasks, we need to calculate how long an application running with this placement will take to complete. By definition, this completion time is equal to the time taken to complete the longest-running flow.

Let $f_1, \ldots, f_k$ be a set of flows that share a bottleneck link ℓ with rate R. The flows transmit $b_1 \leq, \ldots, \leq b_k$ bytes of data, respectively. The amount of data traversing link ℓ is then

$$\sum_{i=1}^{i=k} b_i$$

Because there is no cross traffic (by our assumption), the total amount of time these flows take is

$$\sum_{i=1}^{i=k} b_i / R.$$

The total completion time for the workload placement, then, is

$$\max \sum_{i=1}^{i=k} b_i / R$$

over all sets of flows that share a bottleneck link. Cicada's goal is to minimize that time over all possible placements.

To formulate this objective so that it can be solved by quadratic program solvers, we need to express it using matrices. In particular, we need a way to define whether two flows will share a bottleneck link. Let S be an $M^2 \times M^2$ matrix such that $S_{mn,ab} = 1$ if the path $m \leadsto n$ shares a bottleneck link with path $a \leadsto b$. Let D_{mn} be the $M \times M$ matrix expressing the amount of data to be transferred between machines m and n ($D = X^T BX$). Finally, let

$$E_{mn} = \sum_{m'=1,n'=1}^{m'=M,n'=M} D_{m'n'} \times S_{mn,m'n'}.$$

E_{mn} expresses the amount of data traveling on $m \rightsquigarrow n$'s bottleneck link. Note that the rate of that path is, by definition, R_{mn}. Our objective, then, is

$$\min \max_{m,n} {E_{mn}}/{R_{mn}}$$

such that all CPU constraints are still satisfied.

In §6.5, we show that VMs in both EC2 and Rackspace cloud follow a hose model: the outgoing rate for each VM is limited by a certain threshold. We can model this type of rate-limiting with the following:

$$S_{mi,mj} = 1; m \neq i, m \neq j.$$

If the incoming connection was also rate-limited, we would add

$$S_{im,jm} = 1; (m \neq i, m \neq j).$$

What happens if a tenant does not know the physical topology of the network, i.e., does not know the correct values for S? First of all, this case will only happen if Cicada is run entirely by the customer; the cloud provider will surely know the values for S. Second, the customer can in fact infer values for S; we show this in §6.4.3.

Nevertheless, if S is unknown, Cicada assumes that every entry in S is zero, i.e., that there are *no* shared bottlenecks in the network (effectively, this S models a network with unique paths between every pair of machines). If this value is incorrect, i.e., there are certain paths that share bottlenecks, our formulation will likely calculate the workload completion time incorrectly. However, we show in §4.3 that in practice, significant performance gains can still be seen without knowledge of S.

It turns out that our problem can be converted into a linear programming problem. For each quadratic term in above constraints, $X_{im} \cdot X_{jn}, i < j$, we define a new variable, z_{imjn} ($i < j$ forces $X_{im} \cdot X_{jn}$ and $X_{jn} \cdot X_{im}$ to be equal). Now the problem is as follows:

minimize: z

subject to:

$z - \sum_{i=1}^{J} \sum_{j=1}^{J} \frac{B_{ij} \cdot z_{imjn}}{R_{mn}} \geq 0 \, \forall \, m, n \in [1, M]$

$C_m - \sum_{i=1}^{J} CR_i \cdot X_{im} \geq 0 \, \forall \, m \in [1, M]$

$\sum_{m=1}^{M} X_{im} = 1 \, \forall \, i \in [1, J]$

$z_{imjn} - X_{im} \leq 0$, and $z_{imjn} - X_{jn} \leq 0 \, \forall \, i, j \in [1, J]$

$\sum_{n,m} \sum_{j=i+1}^{J} z_{imjn} + \sum_{n,m} \sum_{j=1}^{i-1} z_{jnim} = J - 1 \, \forall \, i \in [1, J]$

$z \geq 0$

binaries:

X_{im}, z_{imjn} ($i < j, i, j \in [1, J], m, n \in [1, M]$)

The constraints serve the following purposes, in order: To force z to represent the maximum amount of data transferred between two machines, to make sure the computation capability of any machine is not exceeded, to make sure each task is placed on exactly one machine, to force $z_{imjn} \leq X_{im} \cdot X_{jn}$, and to force $z_{imjn} \leq X_{im} \cdot X_{jn}$. The resulting program can be solved using a solver such as CPLEX [28].

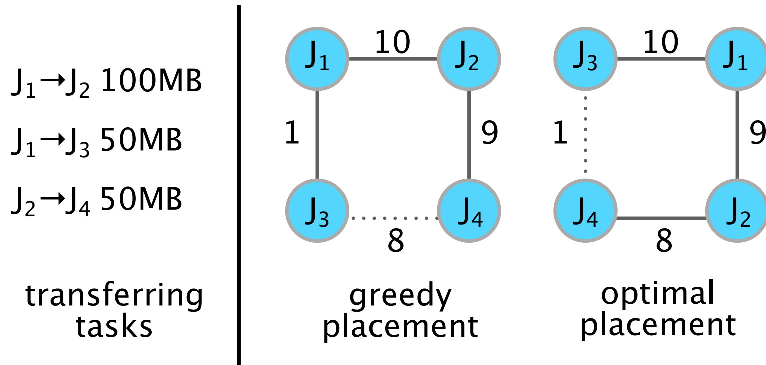


Figure 4-1: An example topology where the greedy network-aware placement is sub-optimal. Because the greedy placement algorithm first places tasks J_1 and J_2 on the path with rate 10, it must also use the path with rate 1 to place J_1 and J_3. The optimal placement avoids this path by placing J_1 and J_2 on the path with rate 9.

4.2.5 Placing Applications Using Heuristics

In practice, we found that ILPs sometimes took a very long time to solve (on the order of tens of minutes to hours), hampering our ability to place applications quickly. Moreover, the larger the number of machines in the network and the number of tasks in the application, the longer the mathematical optimization will take.

To cope with this challenge, Cicada can use heuristics instead. We present the heuristic for minimizing application completion time below. In our experiments, the heuristic takes only milliseconds to run; see §4.3.4.

Greedy Heuristic for Minimizing Application Completion Time

To minimize application completion time, Cicada uses Algorithm 4. This algorithm works by trying to place the pairs of tasks that transfer the most data on the fastest paths. Because intra-machine links are modeled as paths with (essentially) infinite rate, the algorithm captures the heuristic of placing pairs of transferring tasks on the same machines. Though not guaranteed to result in a globally optimal placement (see Figure 4-1), this method scales better to larger topologies. We compared our greedy algorithm to the optimal algorithm on a subset of our applications, and found that the median completion time with the greedy algorithm was only 4% more than the completion time with the optimal algorithm (see §4.3.2 for a more thorough description of this experiment).

Although Cicada can support any goal that can be formulated as an ILP, it is not clear whether every goal has a corresponding greedy heuristic, as minimizing completion time does.

Algorithm 4 Greedy Network-aware Placement

1: *transfers* = a list of $\langle i, j, b\rangle$ tuples, ordered in descending order of rates, b. $\langle i, j, b\rangle$ means Task i transfers b bytes to Task j.
2: **for** $\langle i, j, b\rangle$ in *transfers* **do**
3:   **if** i has already been placed on machine k **then**
4:     P = set of paths $k \rightsquigarrow N$ $\forall$ nodes N
5:   **if** j has already been placed on machine ℓ **then**
6:     P = set of paths $M \rightsquigarrow \ell$ $\forall$ nodes M
7:   **if** neither i nor j have been placed **then**
8:     P = set of paths $M \rightsquigarrow N$ $\forall$ nodes M and $\forall$ nodes N
9:   **for** path $m \rightsquigarrow n$ in P **do**
10:     **if** placing i on m or j on n would exceed the CPU constraints of m or n **then**
11:       Remove $m \rightsquigarrow n$ from P
12:   **for** path $m \rightsquigarrow n$ in P **do**
13:     *rate*(m, n) = the rate that the transfer from i to j would see if placed on $m \rightsquigarrow n$. This rate takes into account all other task pairs already placed on $m \rightsquigarrow n$ for a "pipe" model, or all other connections out of m for a "hose" model. (See §4.4.2 for an evaluation of the effect of network variability on Cicada's placements.)
14:   Place i and j on path $m \rightsquigarrow n \in P$ such that *rate*(m, n) is maximized.

4.2.6 Handling Multiple Applications

A tenant may not know all of the applications it needs to run ahead of time, or may want to start some applications after others have begun. To run a new application while existing ones are running, Cicada re-measures the network, and places its tasks as it would normally (presumably there is more variation in the network in this case, because existing applications create cross traffic). It is possible, however, that placing applications in sequence in this manner will result in a sub-optimal placement compared to knowing their demands all at once. For that reason, every T minutes, Cicada re-evaluates its placement of the existing applications, and updates the placement accordingly. T can be chosen to reflect the cost of updating; for instance, if an update causes a virtual machine to be migrated, T may reflect the length of downtime of the virtual machine. If migration is cheap, T should be smaller. This re-evaluation also allows Cicada to react to major changes in the network.

The actual migration of a task could be done in at least two different ways.

- Migrating at the application level. Certain applications may be able to send control messages assigning a task to a new virtual machine (as an example, consider a MapReduce master assigning a map task to a new machine). This type of migration would have low overhead, but is very specific to the application involved, and may not be possible in all cases.

- Migrating the virtual machine on which the task is placed. This type of migration is much more general, but may cause an unacceptable amount of overhead if done frequently.

§4.4.2 gives a sense of how frequently placement updating may occur on the Amazon EC2 network, and how the overhead of migration may affect an application.

4.3 Application Placement Evaluation

The first goal of our evaluation is to show that Cicada does indeed improve application completion times compared to other placement algorithms. Additionally, we also show that this same placement algorithm can be used in a second context: to improve network utilization within the datacenter.

4.3.1 Improving Application Completion Time

We evaluate Cicada's placement algorithm in two different scenarios. First, a case where a tenant wants to run multiple applications all at once, and second, a case where a tenant wants to run multiple applications, the entire sequence of which is not known up front. In each case, we compare Cicada's placement to three alternate existing placement algorithms: random, round-robin, and minimum-machines (we evaluate Cicada's placement against the optimal placement on a smaller set of applications in §4.3.2).

Random Placement

Tasks are assigned to random VMs. This assignment makes sure that CPU constraints are satisfied, but does not take the network into account. This type of placement acts as a baseline for comparison.

Round-robin Placement

This algorithm assigns tasks in a round-robin order to VMs; a particular task is assigned to the next machine in the list that has enough available CPU. As before, CPU constraints are satisfied, but the network is not taken into account. This placement is similar to one that tries to load balance, and minimize the amount of CPU used on any particular VM.

Minimum-machines Placement

This algorithm attempts to minimize the number of machines used. If possible (given CPU constraints), a task will be placed onto a VM that is already used by another task; a new VM will be used only when no existing machine has enough available CPU. This algorithm may be of interest to cloud customers who aim to save money;

in many clouds, the fewer machines a customer uses, the lower the cost (at, perhaps, the expense of longer-running applications).

4.3.2 Dataset and Experimental Set-up

To evaluate Cicada's placement algorithm, we ran experiments on Amazon EC2, as this allowed us to run our own traffic on top of unknown (and real) background traffic. Our evaluation utilizes Cicada's customer-centric measurement module, Choreo, to measure EC2 topologies; see Chapter 6 for a description and evaluation of Choreo.

In each experiment, we measure an EC2 topology using Choreo, and then place an application or sequence of applications on that topology. The applications are composed from the ground truth traffic matrices in the HPCS dataset, not Cicada's predicted traffic matrices, as we seek to evaluate our placement algorithm alone (see §4.4 for a complete end-to-end evaluation).

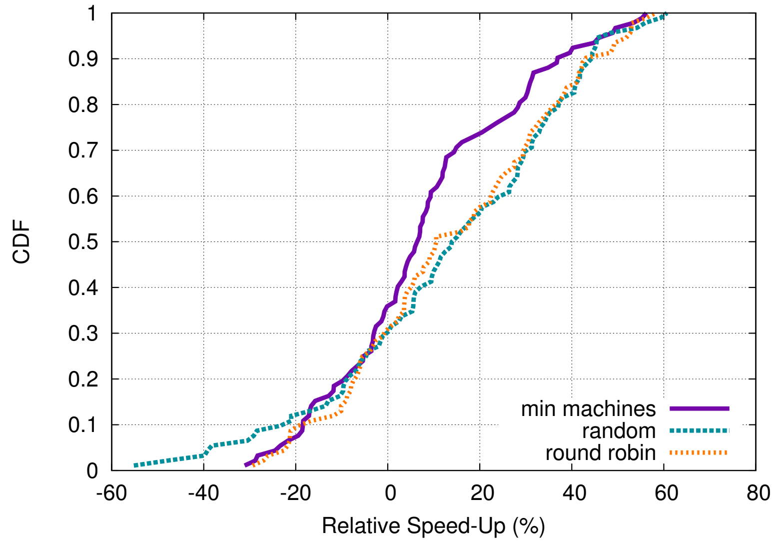
For each application, we know the observed start time on the cloud as well as its traffic matrix. From this data, we can accurately model sequences of applications. We model each component of an application as using between 0.5 and 4 CPU cores, and each cloud machine as having four available cores (the actual CPU data is not available from our dataset).

Once the applications are placed, we transfer data as specified by the placement algorithm and the traffic matrix. Note that these experiments transfer *real traffic* on EC2; we do not merely calculate what the application completion time would have been based on the measured network, placement, and traffic matrix. Thus, these experiments are susceptible to any changes in the network, as well as the effects of any cross traffic.

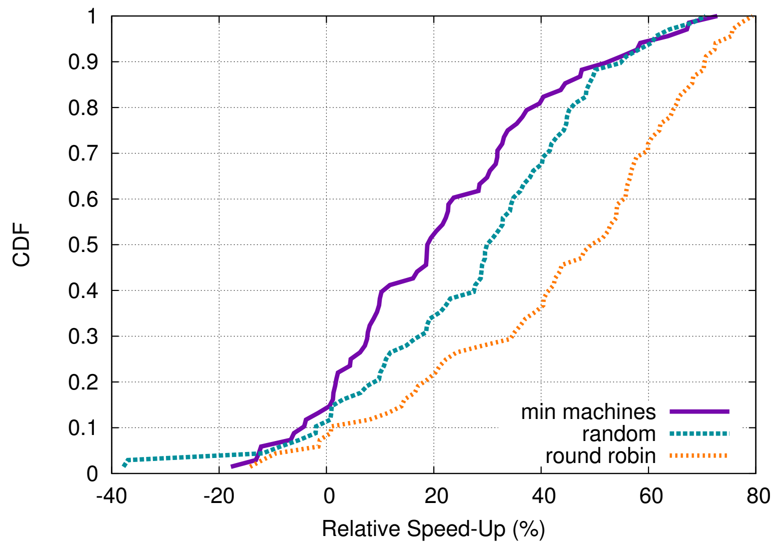
All Applications at Once

For our first set of experiments, we modeled a tenant with multiple applications, whose network demands (amount of data transferred between tasks) are known a priori. We randomly chose between one and three applications and made one combined application out of them, combining each application's traffic demand matrix and CPU vector in the obvious way. Then, we placed this combined application and ran it, using each of the four placement algorithms in turn. Our CDFs do not explicitly account for measurement time, because it is the same (approximately three minutes for a ten-node topology) regardless of the length of an application's run. We note that Cicada is not meant for short-lived applications where the three-minute time-to-measure will exceed any reductions in the completion time from Cicada.

Figure 4-2(a) shows the results from this experiment. Each line in the CDF compares Cicada to one of the alternate placement algorithms. The x-axis plots the relative speed-up, defined as the amount of time that Cicada saved (or added) to the completion time of an application. For instance, if an application took five hours with



(a) All applications at once



(b) Applications arriving in sequence

Figure 4-2: Relative speed-up for applications using Cicada vs. alternate placement algorithms.

the random placement algorithm, and four hours using Cicada, the relative speed-up would be $(5-4)/5 = 20\%$.

From Figure 4-2(a), we can see that in roughly 70% of the applications, Cicada improves performance, with improvements as large as 60%. The mean improvement

over other approaches in *all* cases is between 8% and 14%, while the median improvement is between 7% and 15%; restricted to the 70% of applications that show improvement, these values rise to 20%–27% (mean) and 13%–28% (median). In the other 30% of applications, Cicada reduces performance; in these cases, the median slow-down is (only) between 8% and 13%.

These numbers imply that, if we could predict the types of applications where Cicada typically provides improvement, it would improve performance by roughly 13%–28%. We leave this type of prediction to future work.

Applications in Sequence

For our second set of experiments, we modeled a tenant wishing to run multiple applications whose demands are not all known a priori. Instead, the applications arrive one-by-one, and are placed as they arrive. We randomly chose between two and four applications and ordered them by their observed start times. We placed the applications in sequence according to these start times. Applications may overlap in time in this scenario.

Because multiple applications arrive in sequence, it does not make sense to measure the entire sequence's completion time. Instead, we determine the total running time of each application, and compare the sum of these running times for each placement algorithm.

Figure 4-2(b) shows the results from this experiment. As in Figure 4-2(a), each curve in the CDF compares Cicada to one of our alternate placement algorithms. We see similar results as in Figure 4-2(a), in that for most applications, Cicada provides an improvement over all other placement schemes. With sequences of applications, we see an improvement in $85\% - -90\%$ of all applications, with a maximum observed improvement of 79%. Over all applications, the mean improvement ranges from 22%–43%; the median from 19%–51% across different alternative algorithms. Restricted to the applications that show improvement, the mean rises slightly to 26%–47% and the median to 23%–53%. For the applications whose performance degraded, the median slow-down was only 10%.

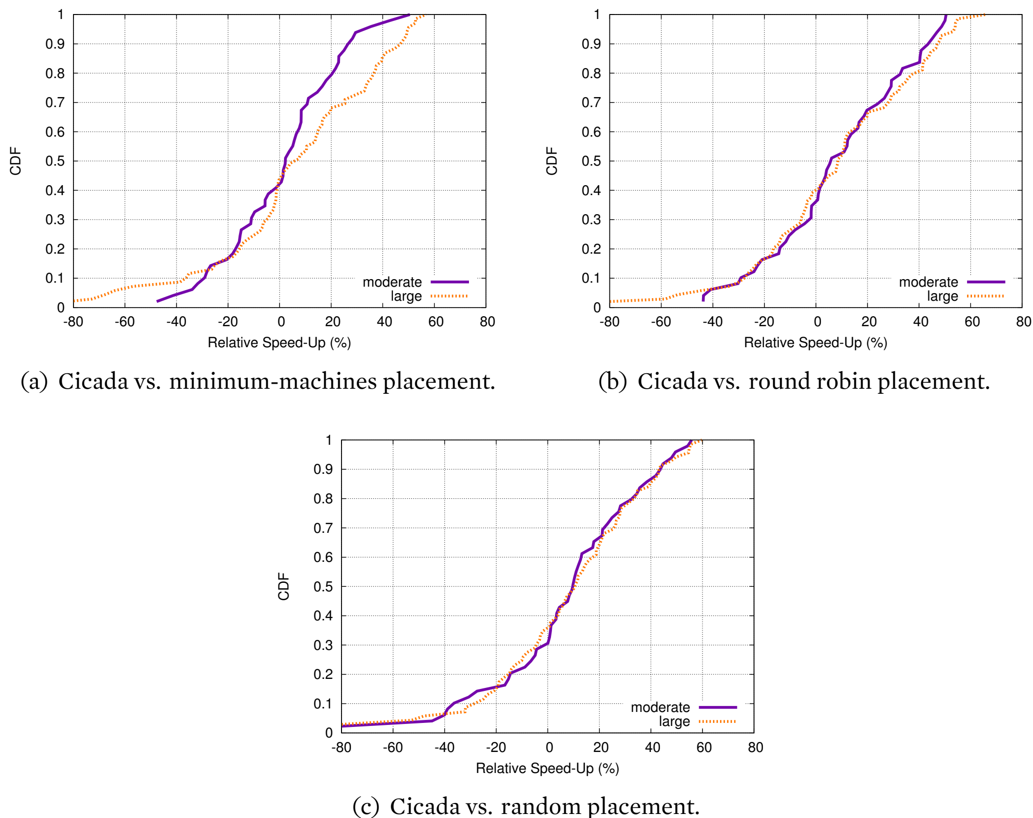
In general, Cicada performs better when applications arrive in sequence than when all demands are known up front and applications can be placed at once. This result is likely due to the fact that applications arriving in sequence allows us to spread their network demands out more, as some of the transfers from earlier applications may have finished by the time later applications are placed. Placing multiple applications at once will use more network resources than placing the applications as they arrive, in general.

In cases where an in-sequence placement seems to be going poorly, however, Cicada can re-evaluate its placement during a run, and migrate applications if the tenant deems that worthwhile. However, we can see from Figure 4-2(b) that even without this behavior, Cicada's relative performance improvement over the other schemes when constrained to no re-evaluations is significant.

Effects of Traffic Magnitude

In the previous results, we made no distinction between Cicada's performance on differently-sized applications. Since Cicada is meant for large network-intensive applications, it would be problematic if Cicada's performance improvement decreases as applications send more traffic.

To study this, we perform the same experiment as in Figure 4-2 on a subset of our applications, but divide the applications into two types: those that send a very large amount of traffic, and those that send a moderate amount of traffic. The applications in the large set send at least an order of magnitude more traffic than those in the moderate set. (due to the restrictions of our dataset, we are not permitted to divulge information about the precise total amount of traffic in any application).



(a) Cicada vs. minimum-machines placement.

(b) Cicada vs. round robin placement.

(c) Cicada vs. random placement.

Figure 4-3: Cicada's performance improvement compared to the three alternative placements, divided into large and moderate applications. Cicada's performance improvement is comparable for large applications when compared to moderate applications, indicating that large, network-intensive applications are well-served by Cicada. (This experiment was done after that in Figure 4-2, and so the results are not exactly the same, as the network almost certainly changed between the two experiments.)

Overall, Cicada performed comparably for large and moderate applications. Figure 4-3 shows the results when Cicada is compared to each of the three placements. The most dramatic change comes when Cicada is compared to the minimum-machines placement: Cicada's median performance improvement is 7% for large applications and 2% for moderate applications in this case (the mean improvements are 6% and 1%, respectively). The minimum-machines placement utilizes many of the same network paths; since it is using fewer machines, it has fewer available paths. As traffic in an application increases, its performance should decrease, as shown.

Compared to a round-robin placement, Cicada's median improvement is 9% for large applications and 6% for moderate applications (mean: 9% for large, 9% for moderate); compared to a random placement, Cicada's median improvement is 11% for large applications and 10% for moderate applications (mean: 8% for large, 8% for moderate). More traffic in an application has less of an effect on these types of placements, since they effectively spread data out across the network. Thus, Cicada doesn't see as much improvement for large applications over moderate applications in these scenarios.

Cicada vs. Optimal Placement

Evaluating Cicada's placement against the optimal placement is difficult due to the computational complexity of the optimal ILP. In simulation, we observed that it often took multiple hours to determine the optimal placement for applications in our dataset, and even longer for applications with random traffic patterns (those took as long as eight hours). With these times, network conditions may change before the ILP is solved, which means that the proposed "optimal" placement is no longer optimal.

However, we can get a sense of how well Cicada performs against the optimal placement in cases where the ILP can be solved quickly. We limited ourselves to applications whose ILP could be solved in under five minutes, and compared the completion time of the application using the optimal placement to using Cicada's placement. Figure 4-4 shows the results.

First, there are some applications that complete faster using Cicada's placement time rather than the optimal placement time. Even though we chose applications that had an ILP that could be quickly solved, the network may still have changed in that time (Cicada's placement algorithm, on the other hand, takes only a few milliseconds; see §4.3.4).

For the applications where Cicada's placement performs worse compared to the optimal placement, the median slow-down is 23% (mean: 28%). Over all applications, Cicada's median slow-down is 4% (mean: 11%).

Again, due to the time it takes to solve the ILP, it is difficult to do a full comparison between Cicada's placement and the optimal placement. But this fact speaks to a benefit of Cicada's placement: even though the completion time of the application may be greater using Cicada, the total time to calculate the placement and then run

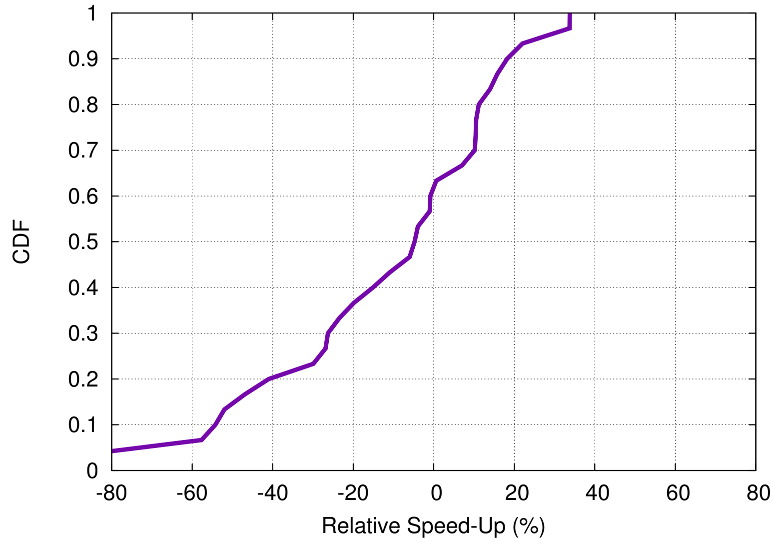


Figure 4-4: Relative speed-up of Cicada's placement vs. the optimal placement. The times when Cicada performs better are likely due to changes in the network.

the application will often be shorter using Cicada, because its placement algorithm is so much faster (§4.3.4).

4.3.3 Improving Network Utilization

We designed Cicada—and, in particular, its placement algorithm—with the goal of satisfying various customer objectives, such as minimizing application completion time. Because our prediction method can output pipe-model predictions (i.e., different predictions for each pair of virtual machines), Cicada is able to make a more precise placement than systems that do not; for example, VOC (see §3.4 as well as [20]).

As a result, we have found that Cicada's placement algorithm has the pleasant side effect of also improving network utilization. By that, we mean that Cicada wastes less bandwidth than other models. We define "wasted bandwidth" as the bandwidth that is predicted for a tenant but not used by the tenant; in some scenarios, this bandwidth may actually be guaranteed to or reserved for the tenant (see Chapter 5). Wasted bandwidth is a proxy for estimating how much money a customer would save—methods that waste less bandwidth will likely save the customers money—but allows us to avoid defining a particular cost model.

In cloud networks, it is important to specify between wasted intra-rack and inter-rack bandwidth, as it is not clear that cloud providers treat wasted intra-rack and inter-rack bandwidth equally. Inter-rack bandwidth may cost more, and even if intra-rack bandwidth is free, over-allocating network resources on one rack can prevent other tenants from being placed on the same rack (due to a presumed lack of network

resources).

Dataset and Experimental Set-up

We compare Cicada's placement algorithm (Algorithm 4) to the VOC placement algorithm detailed in [20], which tries to place clusters on the smallest subtree that will contain them. This algorithm is similar in spirit to Cicada's algorithm; Cicada's algorithm tries to place the most-used VM pairs on the highest-bandwidth paths, which in a typical datacenter corresponds to placing them on the same rack, and then the same subtree. However, since Cicada uses fine-grained, pipe-model predictions, it has the ability to allocate more flexibly; VMs that do not transfer much data to one another need not be placed on the same subtree, even if they belong to the same tenant.

We compare Cicada's placement algorithm against VOC's, on a simulated physical infrastructure with 71 racks with 16 servers each, 10 VM slots per server, 10G links between servers and ToRs, and $(10\text{G}/O_p)$ Gbit/s inter-rack links, where $O_p > 1$ is the physical oversubscription factor (this infrastructure resembles the one on which we collected our own dataset; see 2.5.3). For each algorithm, we select a random tenant, and use the *ground truth* data to determine this tenant's bandwidth needs for a random hour of its activity, and place its VMs. We repeat this process until 99% of the VM slots are filled. Using the ground-truth data allows us to compare the placement algorithms explicitly, without conflating this comparison with prediction errors. To get a sense of what would happen with more network-intensive tenants, we also evaluated scenarios where each VM-pair's relative bandwidth use was multiplied by a constant bandwidth factor ($1\times$, $25\times$, or $250\times$).

Results

Figure 4-5 shows how the available inter-rack bandwidth—what remains unallocated after the VMs are placed—varies with O_p, the physical oversubscription factor. In all cases, Cicada's placement algorithm leaves more inter-rack bandwidth available. When O_p is greater than two, both algorithms perform comparably, since this is a constrained environment with little bandwidth available overall. However, with lower over-subscription factors, Cicada's algorithm leaves more than twice as much bandwidth available, suggesting that it uses network resources more efficiently in this setting.

Over-provisioning reduces the value of our improved placement, but it does not necessarily remove the need for better predictions or placements. Even on an over-provisioned network, a tenant whose reserved bandwidth is too low for its needs may suffer if its VM placement is unlucky. A "full bisection bandwidth" network is only that under optimal routing; bad routing decisions or bad placement can still waste bandwidth.

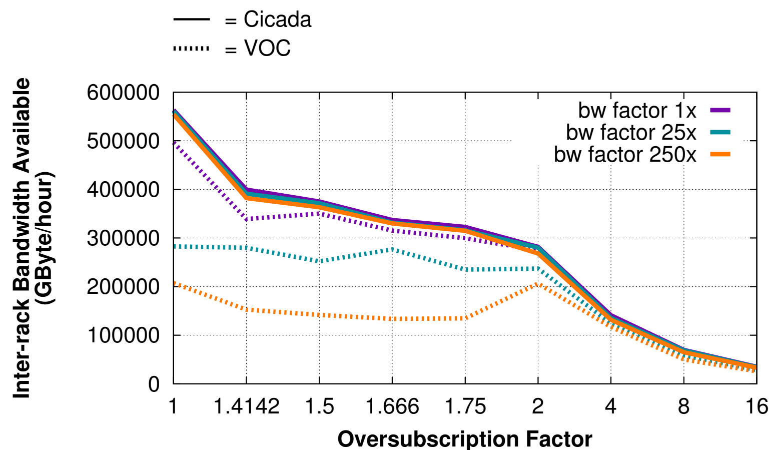


Figure 4-5: Inter-rack bandwidth available after placing applications (note that the x axis is roughly log-scale). Solid lines represent Cicada's placement, dotted lines represent VOC's placement.

4.3.4 Scalability

Cicada's placement algorithm takes only milliseconds to run. On applications in our dataset, running on Amazon EC2, a placement never took more than three milliseconds, though the time to calculate the placement grew as the number of tasks grew. To push the limits of scale, we simulated random network conditions, where the number of machines varied from 10 to 100, and random applications, where the number of tasks varied from 2 to 500. In every configuration, Cicada was able to calculate a placement in under 1.64 seconds (in fact, it took our simulation more time to generate the random conditions than it did to calculate the placement).

End-to-end, the time it takes to place an application will be dominated by the time it takes to do the measurement, not the time it takes to do make a prediction or do a placement. See §6.5.4 for a discussion of the scalability of client-side measurement, and §3.5.5 for a discussion of the scalability of predicting application workloads.

4.3.5 Elasticity

One benefit of public clouds is that they allow users to launch new machines on demand, and often automatically (via a CPU-based threshold, e.g., "launch a new VM if 70% of my current VMs are using over 80% of their CPU capabilities"). Cicada does not address the problem of whether it would be better to launch additional VMs to improve application performance rather than placing the VMs in a particular way.

Even with additional VMs to handle load, Cicada would still be useful to determine how to place (or migrate) the application to best take advantage of the new collection of VMs.

4.4 End-to-end Evaluation

In this section, we present an end-to-end evaluation of Cicada, studying the effect of its predictions on its placement decisions, and the expected frequency of placement updates.

4.4.1 Determining Placement with Predictions

To determine how Cicada's predictions affect its placement, rather than using the ground truth traffic matrices from the HPCS dataset as we did in the previous section, we use Cicada's predictions for each application. Since it is possible for Cicada's predictions to have errors (see §4.3), this evaluation will illustrate whether Cicada can still improve application completion time despite these errors.

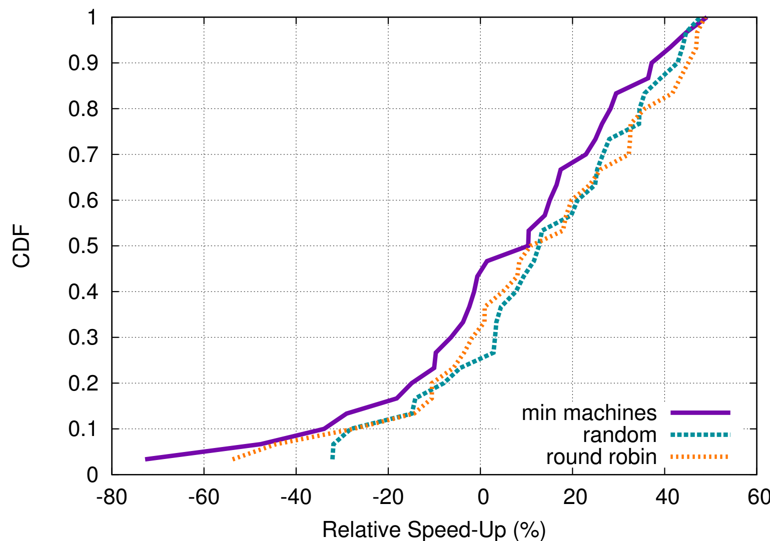


Figure 4-6: Relative speed-up for applications using Cicada, along with its predictions, vs. alternate placement algorithms.

Figure 4-6 shows the affects on application completion time when using Cicada's predictions instead of the ground-truth data as in Figure 4-2. Here, Cicada provides improvement for roughly 55%–75% of applications. The median improvement is 11%–18% over all applications, and 25%–26% restricted to the set of applications that Cicada improves. These are comparable to the numbers reported in §4.3.

To further understand the impact of Cicada's predictions on its placement, we performed an additional experiment. For each application A, we placed an application with Cicada once using its ground truth data, and once using its predicted data (measuring the network before each placement, and randomizing whether we used the ground-truth data first or the predicted data). We then get two completion times: A_g, the completion time when using the ground-truth data to determine the placement, and A_p, the completion time when using the predicted data to determine the placement. If Cicada's predictions hinder A's performance, then A_p will be greater than A_g.

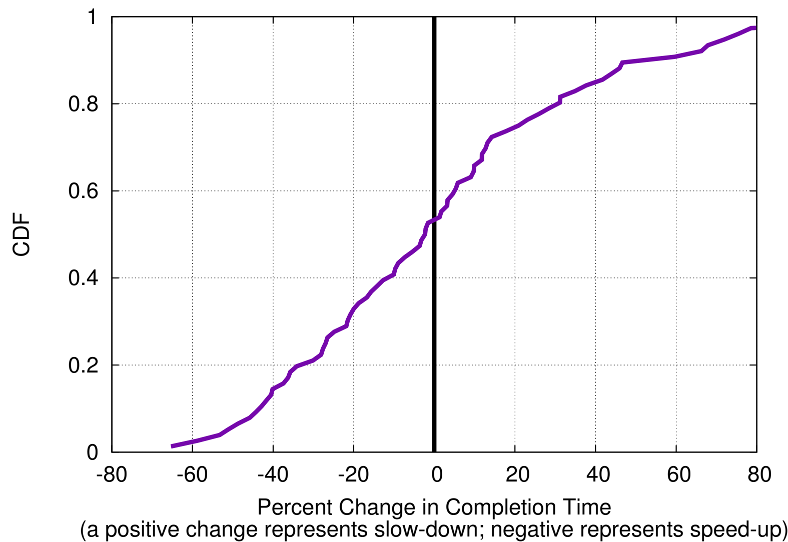


Figure 4-7: Change in completion time between using Cicada's predictions to place an application and using ground truth data to place an application.

Figure 4-7 shows a CDF of the differences between A_g and A_p. 53% of the time, Cicada's predictions result in a *decrease* in completion time. This result may be due to changes in the network between the two placements of the application. During the 47% of the time when the predictions increase completion time, the median increase is 17%. There is, however, a relatively long tail (not pictured); the maximum increase is 126%.

We believe that most of the differences between A_g and A_p are due to network conditions changing. This interpretation is supported by the fact that Figure 4-7 is relatively symmetric about $x = 0$. To that end, it seems that even if Cicada's predictions are slightly incorrect, its placement will still improve completion time. This result is encouraging, with respect to Cicada's use as an end-to-end system. It is also understandable: to obtain a correct placement, it is more important that Cicada predict the correct *relationship* between the amount of data sent between task pairs,

rather than the exact value. For instance, suppose tasks i and j transmit the most data within an application: 100GB. Cicada's placement algorithm will try to place i and j on the fastest path in the network. As long as Cicada's predictions correctly indicate that i and j transfer more data than any other task pair, the placement algorithm will still place them on the fastest path; whether Cicada actually predicts 100GB correctly is immaterial.

Note, though, that it is still worthwhile for Cicada to make accurate predictions, as they can lead to better performance guarantees; see Chapter 5.

4.4.2 Frequency of Placement Updates

We are also interested in determining how frequently the placement of an application should be updated. After a change in the network, or a change in the application's workload, the original placement may no longer be optimal. In this section, we study how both of these types of changes affect the frequency of task migration.

To evaluate the frequency of migration, our general approach is to compare the original placement of an application, p_0, with some new placement p, calculated at a later time. As a result, p should be a more accurate placement than p_0.

Rather than calculate the number of migrations between p and p_0, we calculate the completion time of the application at time t using both of those placements, and calculate how much p_0 improves completion time. We choose this method to avoid cases where the migrated tasks had little affect on the overall completion time (and thus could've been avoided); for example, migrations that caused only short network transfers to change paths, or migrations that moved network transfers to new paths where they would achieve similar throughput as on their original path.

Effect of Network Variability

To determine how network variability affects the frequency of migration, we performed the following experiment. We collected measurements of network stability on two different 20-VM networks in Amazon EC2. These measurements consist of the TCP throughput measured every five minutes for one hour, on every path in the network.

For each application in our dataset, we calculate its placement on one of these networks every five minutes for an hour; we refer to the placement at time 0 as the original placement, p_0. The application's workload matrix remains constant, but the network-rate matrix will change as a result of variability in the network. We compare the completion time of each new placement p_i to p_0.

Figure 4-8 shows a CDF of the changes in completion time. From this figure, we see that network variability has very little effect on the placement, changing the total completion time by no more than 5% in virtually every case (the maximum value, not pictured, is 15.6%). This result is consistent with the network stability results in §6.5.1), and indicates that on EC2, network variability does not affect applications in an

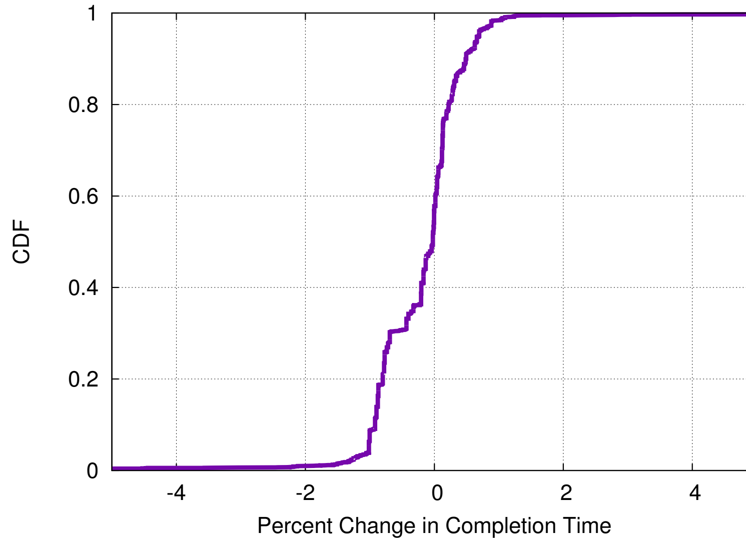


Figure 4-8: Percent change in completion time when measuring the network only once vs. every five minutes.

appreciable way. Thus, applications will not need to migrate prohibitively frequently, if at all, due to changes in the network.

Effect of Application Variability

Although network variability may have little effect on task migration, the application's workload can. Here, we examine how much Cicada's placement changes as an application progresses.

For each application in our dataset, we calculated its ground-truth traffic matrix for each hour of its runtime; we refer to these matrices as M_0, M_1, and so on. These traffic matrices reflect the variation in an application's traffic each hour. We calculate the total completion time of the application under two scenarios:

1. Allowing a different placement for each M_i. This is equivalent to re-running Cicada's placement method every hour.
2. Allowing a single placement p for all M_i's, where p is calculated with M_0. This is equivalent to running Cicada's placement method once with an hour's worth of history, and never running it again.

Figure 4-9 shows the percent increase in completion time when only running Cicada's placement method once vs. every hour (item 2 above). The majority of applications saw no increase in completion time; the maximum increase was 10.8%. This indicates that on our dataset, many applications could be placed once and never migrated. Others would benefit from being migrated every hour.

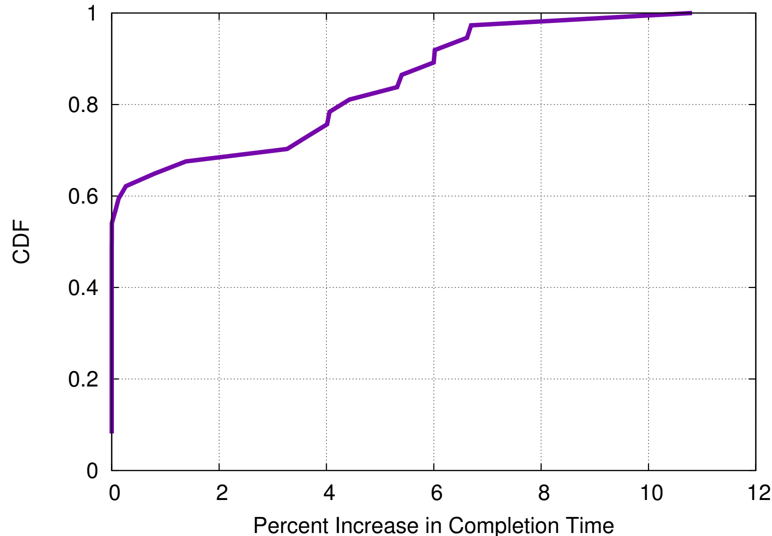


Figure 4-9: Percent increase in completion time when running Cicada's placement algorithm only once vs. every hour.

As discussed in §4.2.6, migrating tasks may cause a concerning amount of overhead, particular if task migration involves virtual machine migration. However, recent work [92] suggests that virtual machine migrations take on the order of one minute, from start to finish, with only a few seconds of downtime (during the rest of the time, services experienced degraded performance but were not inaccessible). These VMs had up to two virtual CPUs and had up to 2GB of memory, which is comparable to the types of VMs used in today's clouds (although the amount of virtual memory available is growing) [12]. As such, we believe that migrating virtual machines on an hourly basis is reasonable. It is possible that the migration will transfer more data over an already-congested path. We believe this effect will be negligible on the types of applications Cicada targets, namely long-running applications. With applications that run for many hours or days, and placements that do not need to be updated more than once per hour, having migrations occur for one minute per hour is reasonable.

Note that these migration results apply to our dataset and the Amazon EC2 network. It is possible that other applications may need to be migrated more frequently. Cicada has the ability to migrate virtual machines with any frequency; since it continually collects measurements and makes predictions, it can continually update an application's placement. Cicada can also take the cost of migration into account; see §4.2.6. Our results indicate that applications do not need to be migrated frequently, but this is a statement about our dataset, not about Cicada's inherent capabilities. Both Cicada's prediction and placement algorithms take only milliseconds to run (see §3.5.5 and §4.3.4, respectively), and so don't prohibit frequent migration.

4.4.3 Cicada with Multiple Customers

Our evaluation has focused primarily on how Cicada performs when used by a single customer (with the exception of §4.3.3, which assumes that every application in the network is using Cicada's placement algorithm). In general, we believe that Cicada would succeed in the case of multiple customers, because its placement algorithm begins with measurement, either via the provider or the customer (see §4.2.2). As a result, Cicada would be able to place each application with the knowledge of how the network was being affected by the other Cicada customers. How this approach compares to using a centralized Cicada controller to orchestrate all customers remains to be seen.

4.5 Conclusion

In this chapter, we motivated the need for network-aware application placement on cloud computing infrastructures. As applications become more network-intensive, they can become bottlenecked by the network, even in well-provisioned clouds. Without a network-aware system for placing workloads, poor paths can be chosen while faster, more reliable paths go unused. By placing applications with the goal of minimizing the total completion time, Cicada is able to improve application-level performance. Cicada's placement also tends to place tasks that transfer large amount of data on the same machines if possible, avoiding any network transmission time, as well as avoiding slow paths in the network.

Our experiments on Amazon EC2 showed that Cicada improves application completion time by an average of 8%–14% (maximum improvement: 61%) when applications are placed all at once, and 22%–43% (maximum improvement: 79%) when they arrive in real-time, compared to alternative placement schemes studied on realistic workloads.

We also note that tenants may be interested in adding other requirements to their workload; some of the tasks could be specified as "latency-constrained", or certain tasks could be specified as being placed "far apart" for fault tolerance purposes. We believe that all of these types of constraints are reasonable and would be beneficial to tenants. Moreover, they can be formulated as part of our optimization problem, as shown in Appendix A.

Chapter 5

Provider-centric Cicada: Bandwidth Guarantees

5.1 Introduction

Until now, we have viewed Cicada as an end-to-end system that either the customer or the provider can use. In this chapter, we detail a specific use-case for Cicada beyond placing applications: providing bandwidth guarantees. This use-case requires the provider, not the customer, to run Cicada, as providers are in the position of offering bandwidth guarantees to their customers.

Bandwidth guarantees are simply a guarantee on the amount of bandwidth that an application will have available to it. These guarantees are of particular interest in public clouds, such as those offered by Amazon, HP, Google, Microsoft, and others, as they are being used not just by small companies, but also by large enterprises. For distributed applications involving significant network inter-node communication, such as in [20], [81], and [82], current cloud systems fail to offer even basic network performance guarantees; this inhibits cloud use by enterprises that must provide service-level agreements (SLAs). Moreover, having accurate bandwidth guarantees can prevent over-provisioning, thus lowering costs and improving resource utilization, and under-provisioning, thus improving application performance by making sure that adequate bandwidth is allocated to tenants.

In this chapter, we show how to use Cicada's predictions from Chapter 3 as a basis for **predictive guarantees**, a new abstraction for bandwidth guarantees in cloud networks. A predictive guarantee improves application-performance predictability for network-intensive applications, in terms of expected throughput, transfer completion time, or packet latency.

Because predictive guarantees are based on Cicada's prediction module, they provide a better abstraction than prior approaches, for three reasons. First, the predictive guarantee abstraction is simpler for the tenant, because the provider automatically predicts a suitable guarantee and presents it to the tenant.

Second, the predictive guarantee abstraction supports time-varying and space-varying demands. Prior approaches typically offer bandwidth guarantees that are static in at least one of those respects, but these approaches do not capture general

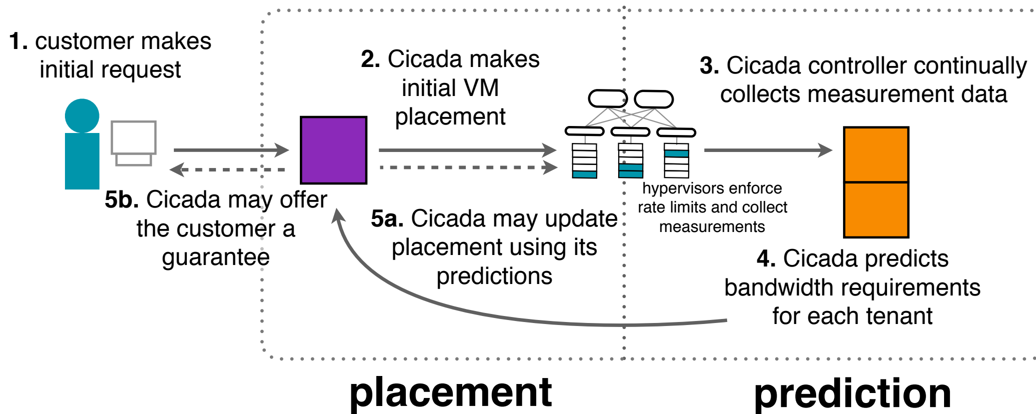


Figure 5-1: Cicada's architecture for providing bandwidth guarantees.

cloud applications (see §3.2).

Third, the predictive guarantee abstraction easily supports fine-grained guarantees. By generating guarantees automatically, rather than requiring the tenant to specify them, we can feasibly support a different guarantee on each VM-to-VM directed path, and for relatively short time intervals. Fine-grained guarantees are potentially more efficient than coarser-grained guarantees, because they allow the provider to pack more tenants into the same infrastructure.

Figure 5-2 depicts Cicada's architecture, when it is used to provide bandwidth guarantees. This figure is the same as Cicada's original architecture (Figure 2-2), with the addition of hypervisor-enforced rate-limits, and the option to offer guarantees to the customer (additionally, changing application placement has become optional).

In this chapter, we describe the interaction between a cloud customer and a predictive guarantees-offering provider. Because offering bandwidth guarantees requires enforcing rate-limits, we also present implementation results that indicate that the necessary rate-limiting can be done scalably.

5.2 Related Work

Cicada's predictive guarantees are related in spirit to Internet QoS research, which involves making reservations for specific types of traffic. QoS on the Internet was a vibrant research area for many years, with architectures such as IntServ developed for end-to-end guarantees. End-to-end Internet QoS has seen little practical deployment, in part because most Internet paths involve multiple providers, making the economics and payment structure of any end-to-end guaranteed QoS scheme difficult. In contrast, a cloud network is run by a single operator, and inherently has a mechanism to bill its customers.

Another drawback of proposals such as IntServ is that they force the application,

or end point, to make explicit reservations and to specify traffic characteristics. For all but the simplest of applications, this task is challenging, and many application developers or operators have little idea what their traffic looks like over time. Cicada resolves this issue through its use of predictions.

5.2.1 Determining Guarantees

Recent research has proposed various forms of cloud network guarantees. Oktopus supports a two-stage "virtual oversubscribed cluster" (VOC) model [20] (also see §3.4), intended to match a typical application pattern in which clusters of VMs require high intra-cluster bandwidth and lower inter-cluster bandwidth. VOC is a hierarchical generalization of the *hose model* [30]; the standard hose model, as used in MPLS, specifies for each node its total ingress and egress bandwidths. The finer-grained *pipe model* specifies bandwidth values between each pair of VMs. Cicada's predictive guarantees can support any of these models (see Chapter 3).

The Proteus system [96] profiles specific MapReduce jobs at a fine time scale, to exploit the predictable phased behavior of these jobs. It supports a "temporally interleaved virtual cluster" model, in which multiple MapReduce jobs are scheduled so that their network-intensive phases do not interfere with each other. Proteus assumes uniform all-to-all hose-model bandwidth requirements during network-intensive phases, although each such phase can run at a different predicted bandwidth. Unlike Cicada, it does not generalize to a broad range of enterprise applications.

Other recent work has focused on making traffic predictions to produce short-term (ten-minute) guarantees for video streaming applications [68]. Although this work considers VM-to-VM guarantees, it is not clear that the approach generalizes to long-term guarantees, or to applications beyond video streaming.

Hajjat et al. [37] describe a technique to decide which application components to place in a cloud datacenter, for hybrid enterprises where some components remain in a private datacenter. Their technique tries to minimize the traffic between the private and cloud datacenters, and hence recognizes that inter-component traffic demands are spatially non-uniform. In contrast to Cicada, they do not consider time-varying traffic nor how to predict it, and they focus primarily on the consequences of wide-area traffic, rather than intra-datacenter traffic.

In contrast to the above related work, and the work on enforcing guarantees that we detail below, Cicada provides a method for determining an application's workload—and what guarantees it needs—ahead of time. Cicada is appropriate for a general class of cloud applications (not just MapReduce or video-streaming applications), and can provide long-term guarantees.

5.2.2 Enforcing Guarantees and Fairness

Cicada does not focus on the problem of enforcing guarantees. This problem can be solved with any of the following systems.

- SecondNet [35], which supports either pipe-model or hose-model guarantees (their "type-0" and "type-1" services, respectively), and focuses on how to place VMs such that the guarantees are satisfied.
- Distributed Rate Limiting [79], which supports a tenant-aggregate limit (similar to a hose model), and focuses on enforcing a limit at multiple sites, rather than within one cloud datacenter.
- GateKeeper [84], which provides a pure hose-model guarantee (with the option of allowing additional best-effort bandwidth) and focuses on protecting each VM's input-bandwidth guarantee against adversarial best-effort traffic.
- NetShare [55], which focuses on how to provide enforcement mechanisms for cloud network guarantees.
- ElasticSwitch [74] and EyeQ [48], which provide work-conserving, hose-model guarantees. ElasticSwitch is implemented entirely in the hypervisor and requires no special topology, while EyeQ uses ECN-enabled switches to enforce guarantees in topologies with congestion-free cores.

Cicada also does not focus on the tradeoff between guarantees and fairness, as in FairCloud [73], which develops mechanisms to support various points in the tradeoff space. Though the issues of enforcing guarantees and fairness would arise for a provider using Cicada's predictive guarantees, they can be addressed with any of the techniques above.

5.3 Provider/Customer Interactions

The goal of using Cicada as a means to offer bandwidth guarantees is to free tenants from choosing between under-provisioning for peak periods, or over-paying for unused bandwidth. For example, suppose that a given tenant's network workload has a ratio of 10:1 between its peak hourly periods and its weekly average. With constant network bandwidth guarantees, the tenant could either request a guarantee equal to its peak needs, and thus overpay, on average, by 10:1, or request a guarantee closer to its average needs, which would substantially under-provision its peaks. Predictive guarantees permit a provider and customer to agree on a guarantee that varies in time and/or space. The customer can get the network service that its tenants need at a good price, while the provider can avoid allocating unneeded bandwidth and can amortize its infrastructure across more tenants.

5.3.1 Architecture Overview

Though we have detailed Cicada's architecture in previous chapters, we briefly describe it again here, so that we can place predictive guarantees in the appropriate context.

Cicada has several components, corresponding to the steps in Figure 5-2. After determining whether to admit a tenant—taking CPU, memory, and network resources into account—and making an initial placement (steps 1 and 2), Cicada measures the tenant's traffic (step 3), and delivers a time series of traffic matrices to a logically centralized controller (see Chapters 4 and 6). The controller uses these measurements to predict future bandwidth requirements (step 4; see Chapter 3).

In the context of this chapter, in most cases, Cicada converts a bandwidth prediction into an offered guarantee for some future interval. Customers may choose to accept or reject Cicada's predictive guarantees (step 5). The customer might also propose its own guarantee, for which the provider can offer a price. Because Cicada collects measurement data continually, it can make new predictions and offer new guarantees throughout the lifetime of the tenant.

Cicada interacts with other aspects of the provider's infrastructure and control system. The provider needs to rate-limit the tenant's traffic to ensure that no tenant undermines the guarantees sold to other customers. We distinguish between *guarantees* and *limits*. If the provider's limit is larger than the corresponding guarantee, tenants can exploit best-effort bandwidth beyond their guarantees.

A provider may wish to place and perhaps migrate VMs based on their associated bandwidth guarantees, to improve network utilization (step 5a). We describe a method for doing so in Chapter 4 (§4.3.3). The provider could also migrate VMs to increase the number of guarantees that the network can support [31].

5.3.2 Assumptions

As Cicada's predictions require at least an hour or two of data before they become useful, we assume that the customer's application is a long-running one (this is an assumption across all of Cicada, not just as it is applied to bandwidth guarantees).

Any shared resource that provides guarantees must include an admission control mechanism, to avoid making infeasible guarantees. We assume that Cicada will incorporate network admission control using an existing mechanism, such as [20], [47], or [53]. We also assume that the cloud provider has a method to enforce guarantees, such as those in §5.2.2. Existing cloud stacks do admission control for CPU and memory resources, and previous cloud-related work has incorporated network-admission control [20].

5.3.3 Measurement Collection

Cicada collects a time series of traffic matrices for each tenant. As discussed in §4.2.2, one could do this passively, by collecting NetFlow or sFlow data at switches within the network, or by using an agent that runs in the hypervisor of each machine and using heuristics to map VMs to tenants. However, switch-based measurements create several challenges, including correctly ascribing VMs to the correct tenant.

Our design collects VM-pair traffic measurements, using an agent that runs on each compute server (see §5.4), and periodically reports these to the controller.

We would like to base predictions on offered load, but when the provider imposes rate-limits, we risk underestimating peak loads that exceed those limits, and thus under-predicting future traffic. We observe that we can detect when a VM's traffic is rate-limited (see §5.4.3), so these underestimates can be detected, too, although not precisely quantified. Currently, Cicada does not account for this potential error.

5.3.4 Prediction Model

Cicada determines a predictive guarantee for a tenant using the prediction algorithm detailed in Chapter 3. We include a high-level overview of Cicada's prediction model here, but refer the reader to Chapter 3 for more detailed information.

Cicada's goal is to predict the bandwidth guarantee that best matches a tenant's future needs, taking into account both spatial and temporal variations in bandwidth demand. Some applications, such as backup or database ingestion, require bulk bandwidth—that is, they need guarantees that the average bandwidth over a period of H hours will meet their needs. Other applications, such as user-facing systems, require guarantees for peak bandwidth over much shorter intervals. Thus, Cicada's predictions describe the maximum bandwidth expected during any averaging interval δ during a given time interval H. If $\delta = H$, the prediction is for the bandwidth requirement averaged over H hours, but for an interactive application, δ might be just a few milliseconds.

Note, of course, that the predictive guarantees offered by Cicada are limited by any caps set by the provider; thus, a proposed guarantee might be lower than suggested by the prediction algorithm.

Converting Predictions to Guarantees

A predictive guarantee entails some risk of either under-provisioning or over-provisioning. Different tenants will have different tolerances for these risks, typically expressed as a percentile (e.g., the tenant wants sufficient bandwidth for 99.99% of the 10-second intervals). Cicada uses the results of the prediction to determine whether it can issue reliable predictive guarantees for a tenant; if not, it does not propose such a guarantee.

5.3.5 Recovering from Faulty Predictions

Cicada's prediction algorithm may make faulty predictions because of inherent limitations or insufficient prior information. Because Cicada continually collects measurements, it can detect when its current guarantee is inappropriate for the tenant's current network load. Cicada does this detection by checking for packet drops; see §5.4.1.

When Cicada detects a faulty prediction, it can take one of many actions: stick to the existing guarantee, propose a new guarantee, upgrade the tenant to a higher, more expensive guarantee, etc. How and whether to upgrade guarantees, as well as what to do if Cicada over-predicts, is a pricing-policy decision, and outside our scope. We note, however, that typical System Level Agreements (SLAs) include penalty clauses, in which the provider agrees to remit some or all of the customer's payment if the SLA is not met. A Cicada-based provider could bear some fraction of the cost of upgrading a guarantee before it ends.[1]

We also note that a Cicada-based provider must maintain the trust of its customers: it cannot regularly under-predict bandwidth demands, or else tenants will have insufficient guarantees and their own revenues may suffer; it also cannot regularly over-predict demands, or else customers will be over-charged and take their business elsewhere. The results in §3.5 focus on evaluating the quality of Cicada's predictions, including its decision whether a tenant's demands are in fact predictable.

5.4 Implementation

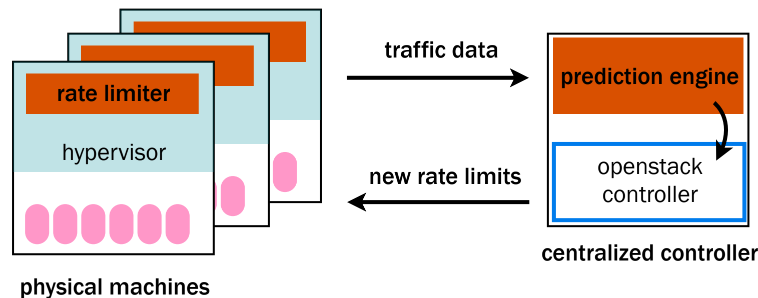


Figure 5-2: Cicada's implementation. Rate-limiters run in the hypervisor of every physical machine, and communicate with a centralized controller.

We have implemented a provider-centric version of Cicada as part of the OpenStack [69] framework. Figure 5-2 details the implementation, which has two components: rate-limiters, which run in the hypervisor of every physical machine, and a controller that runs somewhere in the datacenter. We envision that Cicada extends the OpenStack API to allow the tenant's application to automatically negotiate its network guarantees, rather than requiring human interaction. A typical setting might be "Accept all of Cicada's guarantees as long as my bill does not exceed D dollars," or "Accept all of Cicada's guarantees, but add a buffer of 10 Mbit/s to each VM-pair's

[1]The provider might need to rate-limit its discounting, as there is now incentive for tenants to shape their bandwidth usage to trick the provider into offering a low-cost guarantee, then to increase their traffic in order to obtain the upgraded guarantee at a discount.

allocated bandwidth" (if the customer anticipates some amount of unpredictable traffic).

5.4.1 Server Agents

On each server, Cicada uses an agent process to collect traffic measurements and to manage the enforcement of rate-limits. The agent is a Python script, which manages the Linux `tc` module in the hypervisor. Cicada's agent uses `tc` both to count packets and bytes for each VM-pair with a sender on the server, and, via the the HTB queuing discipline, to limit the traffic sent on each pair in accordance with the current guarantee-based allocation. This type of collection is more fine-grained than sFlow, and `tc` also allows us to detect, but not precisely quantify, traffic demands in excess of the rate-limit, by counting packet drops. This technique is similar to the mechanism ElasticSwitch [74] uses to increase bandwidth guarantees when the initial guarantee is too low, and could also be used for this purpose in Cicada if needed.

The agent receives messages from the controller that provide a list of rate-limits, for the $\langle src, dst \rangle$ pairs where src resides on that server. It also aggregates counter values for those pairs, and periodically reports them to the controller. To avoid overloading the controller, the reporting period P is larger than the peak-measurement period δ.

5.4.2 Centralized Controller

The centralized controller is divided into two components. The prediction engine receives and stores the data about the tenants from each server agent. Once a prediction is made and approved by the tenant, the rate controller uses the OpenStack API to determine which VMs reside on which physical server, and communicates the relevant rates to each rate-limiter. At this point, Cicada could also use its placement algorithm (Algorithm 4) to determine whether any VMs should be migrated, and migrate them via the OpenStack API. Since Cicada makes long-term traffic predictions, this migration would be done at long timescales, mitigating the overhead of migrating VMs.

5.4.3 Scalability

While our current controller implementation is centralized, it does not need to be. Since Cicada makes predictions about applications individually, the prediction engine can be spread across multiple controllers, as long as all of the data for a particular application is accessible by the controller assigned to that application.

Scalability of Pipe-model Rate-limiting

Cicada's use of pipe-model rate-limiting—that is, one rate-limiter for each VM-pair, at the source hypervisor—could potentially create scaling problems. Some prior work

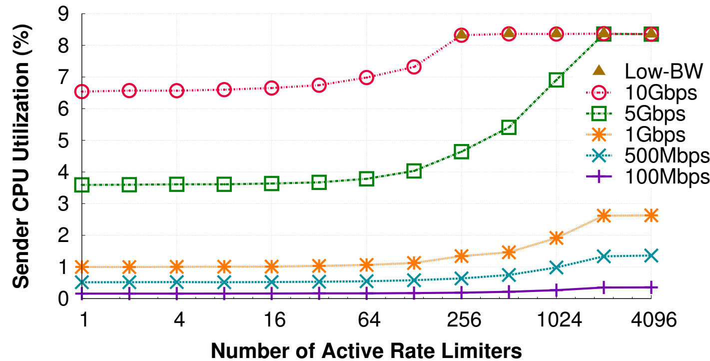


Figure 5-3: Effect of rate-limiters on CPU utilization. We were able to run up to 4096 rate-limiters without significantly effecting on CPU performance.

has also used pipe-model rate-limiting; for example, while Oktopus implements hose-model guarantees, it enforces these using "per-destination-VM rate-limiters" [20]. EyeQ [48] and ElasticSwitch [74] behave similarly.

Existing studies show that pipe-model rate-limiting does scale; see, for instance, SENIC [78] and FasTrak [65]. In addition to these studies, we also performed our own testing of Cicada's rate-limiting.

We tested the scalability of the `tc` rate-limiters used in our implementation on a pair of 12-core Xeon X5650 (2.67GHz) servers running Linux 3.2.0-23, with 10 Gbit/s NICs. We used `netperf` to measure sender-side CPU utilization for 60-second TCP transfers under various rate-limits, while varying the number of sender-side rate-limiters. (Only one TCP stream was active at any time.)

Fig. 5-3 shows mean results for eleven trials. Each curve corresponds to a rate-limit; we marked cases where the achieved bandwidth was under 90% of the target limit. On this hardware, we could run many low-bandwidth limiters without much effect on CPU performance. When we used more than about 128 high-bandwidth limiters, both CPU utilization and TCP throughput suffered, but it is unlikely that a real system would allow both a 10 Gbit/s flow and lots of smaller flows on the same server. Given our results here, and those in related work [65, 78], we conclude that pipe-model rate-limiting does indeed scale.

5.5 Conclusion

This chapter described a provider could use Cicada as a means for offering bandwidth guarantees to its clients. We introduced predictive guarantees, a guarantee abstraction based on Cicada's workload predictions. Predictive guarantees allow a provider to offer fine-grained, temporally- and spatially-varying guarantees without requiring the clients to specify their demands explicitly. In Chapter 4, we showed how the

fine-grained structure of these types of guarantees can be used by a cloud provider to improve network utilization in certain datacenter topologies; in this chapter, we focused on the interactions between the provider and customer, describing how the two might come to terms on an agreement for a bandwidth guarantee. We also showed that an implementation of provider-centric Cicada, including a mechanism to limit rates (and enforce the predictive guarantees), scalable.

Chapter 6

Customer-centric Cicada: Client-side Measurement

6.1 Introduction

To complement the previous chapter, we now turn away from viewing Cicada as it might be implemented and used by a provider to viewing it as it might be used by a customer. In this chapter, we discuss the necessary additions that allow Cicada to be run *entirely by a customer*. That is, for both its prediction and placement to be performed by the customer, without the provider knowing anything about Cicada.

In Chapter 4, we saw how Cicada uses a description of an application's workload, combined with a snapshot of the network, to place applications on the cloud network. Providers can easily retrieve snapshots of their network, via passive measurements from switches or hypervisors. Cloud customers, however, do not have control over the network; they cannot observe any traffic other than their own, and can only see their traffic at endpoints (i.e., they cannot observe their traffic in the middle of the network). One option might be for cloud providers to export certain API functions to allow clients to get some measurement information directly, i.e., without having to actively measure themselves. However, there still may be some information about the network that *only* cloud providers have access to; for instance, the complete physical topology, and detailed information about the cross traffic of other users. Additionally, cloud providers may not want to allow customers access to all of this information (the layout of their entire physical topology, in particular).

Thus, to estimate these quantities, customers must perform their own (active) measurements. In this chapter, we describe a method for doing so. Specifically, we are interested in determining how the achievable TCP throughput in public clouds changes, if at all, and how to best measure these changes so that Cicada can take advantage of them using its placement algorithm.

This chapter discusses the development of a network-measurement system, called **Choreo**, which Cicada can use to obtain its network measurements should they not be available via other means. Figure 6-1 shows how Choreo fits into Cicada's architecture. Choreo uses low-overhead measurements to obtain inter-VM TCP throughputs.

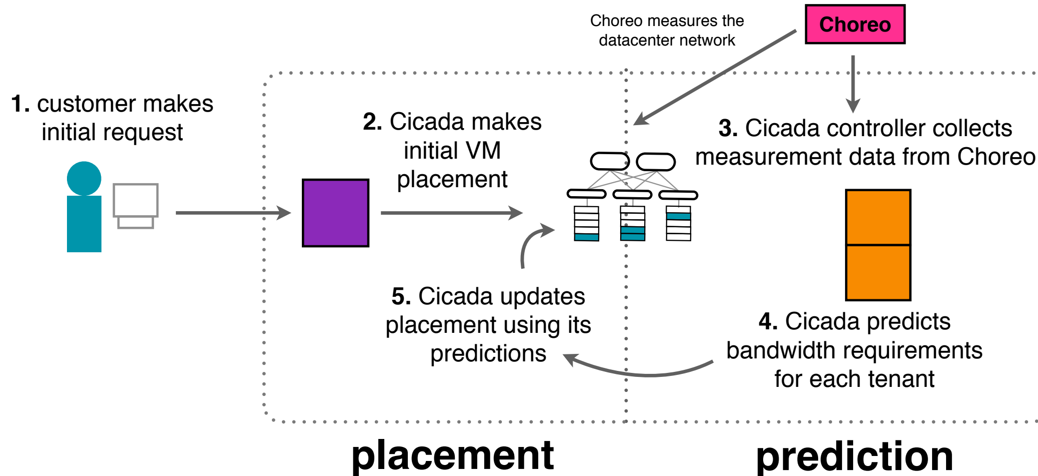


Figure 6-1: Cicada's architecture with the addition of Choreo.

These rates are not constant [22], and are influenced by background traffic from other tenants. Moreover, cloud providers often use a "hose model" to control the maximum output rate from any VM.

We collect network performance data from Amazon's EC2 and Rackspace to evaluate Choreo. We find that it is possible to use packet trains to estimate TCP throughput within a few minutes for ninety VM pairs, and also develop methods for estimating cross traffic and locating bottlenecks within the network. As a side effect of our experiments evaluating Choreo's accuracy, we also present results indicating how the achievable TCP throughput for a connection varies on today's cloud networks, and how this quantity has changed over the past few years.

6.2 Related Work

In accordance with the popularity of datacenters, there have been many recent measurement studies in this environment [21, 22, 58]. These studies corroborate the assumptions in this chapter, for example the typical datacenter topology used in §6.4.3. However, these studies are not focused on the performance of the network and how it varies, which can affect the performance of network-intensive applications. There have been a few efforts to understand the network performance of Amazon EC2 [67], though only at a very small scale.

In [26], Butler conducts measurement for network performances for five major cloud providers, focusing on the throughputs when cloud users upload/download files from outside the cloud, while Choreo focuses on measuring the achievable TCP throughputs between VMs inside the cloud. CloudTalk [80] measures EC2 topology to help optimize latency sensitive applications such as web search. As we show

in §6.5.2, the topology information, i.e, hop-count from traceroute, has no strong correlation with achieved TCP throughput. Thus, compared to CloudTalk, Choreo's measurement techniques are more helpful to big data applications whose performance depends more on throughput instead of latency. Schad, et al [87] measured network throughput between Amazon EC2 instances using iperf. They observed similar throughput variation results as ours in both US and European zones. Wang, et al [93] measured TCP/UDP throughput between VMs in EC2. Similarly, they observed spatial variation in both Small and Medium instances, and they observed temporal variation over a 150-hour period, which is a much larger time scale than we discussed in this chapter.

We borrow the basis of many of our measurement techniques from existing techniques, such as the early work by Bolot [25] and Jain [46] on packet trains. Further related work in this area [72, 75] has indicated that packet trains are not always an accurate measure of path throughput due to their susceptibility to cross traffic. Packet trains are also similar to the "history-based" methods in He et al. [39], but requires significantly less data (He's method requires $10--20$ TCP transfers to make a throughput prediction on a path). We did not consider methods that estimate the available bandwidth of a path [43, 44], as that is not equivalent to the achievable TCP throughput [45].

6.3 Understanding Today's Cloud Networks

Choreo measures the network path between each pair of VMs to infer the potential TCP throughput between them. When we started this project in 2012, we found that the achievable TCP throughput between the virtual machines obtained on Amazon's EC2 was highly variable, as shown in Figure 6-2. In 2013, however, we observed a significant change in how EC2 manages their internal network.

To measure achievable TCP throughput (in 2013), we used 19 10-instance topologies made up of medium Amazon EC2 instances, and 4 10-instance topologies made up of 8-GByte Rackspace instances. For each of the paths in a given topology, we measured the achievable TCP throughput on the path by running netperf for 10 seconds. This method gives us 1710 data points in total across the 19 topologies for EC2, and 360 data points for Rackspace. Figure 6-3 shows a CDF of the throughputs we observed on the instances in both networks. In EC2 (Figure 6-3(a)), although the CDF shows a large range of spatial variability, with throughputs ranging from 296 Mbit/s to 4405 Mbit/s, most paths (roughly 80%) have throughputs between 900 Mbit/s and 1100 Mbit/s (the mean throughput is 957 Mbit/s and the median is 929 Mbit/s). Not shown on the CDF are 18 paths that had extremely high throughput (near 4 Gbit/s). We believe that most of these paths were on the same physical server (see §6.5.2). Because all of these measurements were taken at roughly the same time, we can conclude that the variability in 20% of the paths is not strictly a function of the time of day.

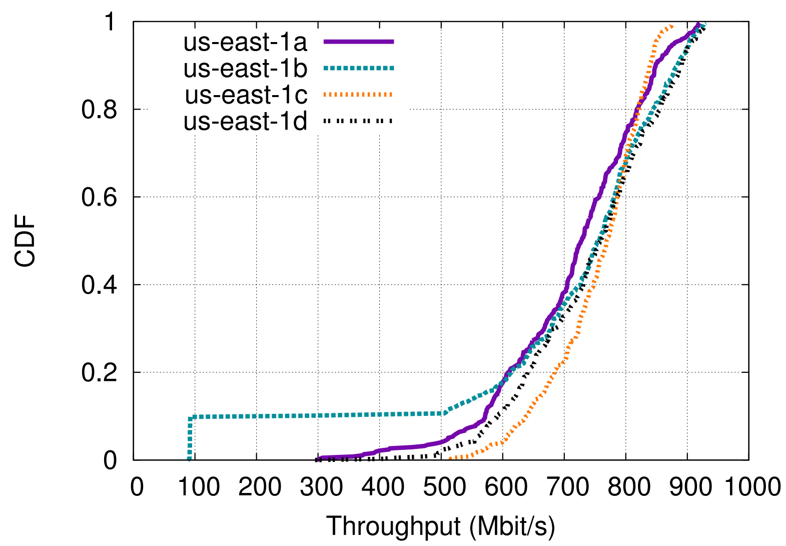
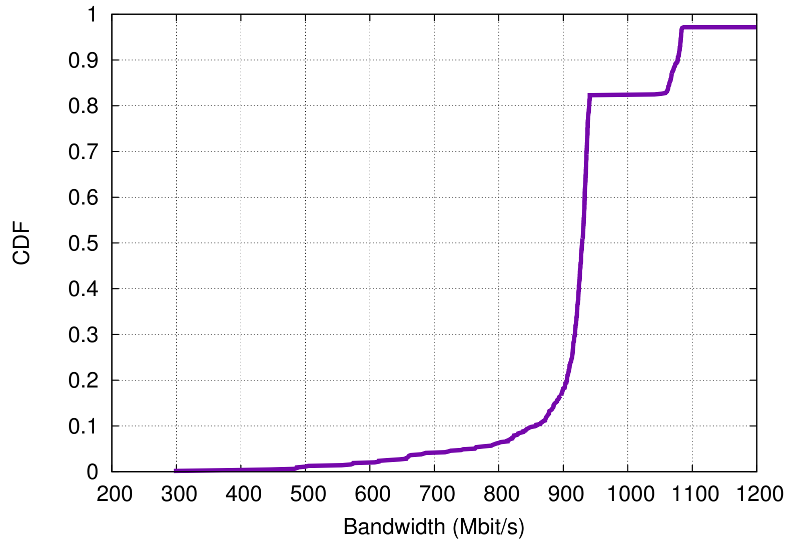
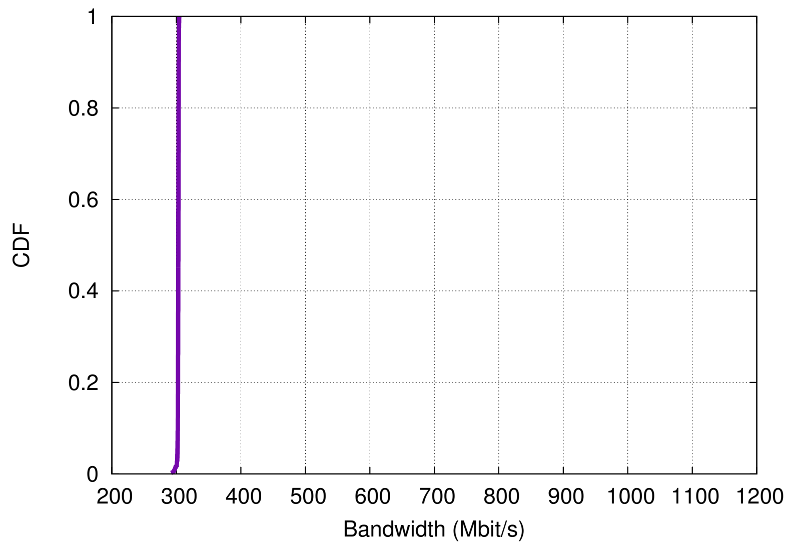


Figure 6-2: TCP throughput measurements on Amazon EC2 taken roughly one year ago, in May, 2012. Each line represents data from a different availability zone in the US East datacenter. Path throughputs vary from as low as 100 Mbit/s to almost 1 Gbit/s.



(a) EC2



(b) Rackspace

Figure 6-3: Achievable TCP throughput measured in May, 2013 on 1710 paths for EC2 and 360 paths for Rackspace. We see some variation on EC2, but less than in Figure 6-2. Rackspace exhibits almost no spatial variation in throughput.

Another interesting feature of Figure 6-3(a) is the distinct knees around 950 Mbit/s and 1100 Mbit/s. We suspect that the knee around 950 Mbit/s is due to rate-limiting, with a raw packet rate of 1000 Mbit/s and a goodput of around 950 Mbit/s (we confirm our rate-limiting hypothesis in §6.5.3). The knee around 1100 Mbit/s may be a result of VMs placed on the same server but with a different (and perhaps older) server architecture than the pairs that achieve a throughput of roughly 4 Gbit/s.

In Rackspace (Figure6-3(b)), there is very little spatial variation. In fact, every path achieves a throughput of almost exactly 300 Mbit/s, which is the advertised internal network rate-limit for the 8-GByte Rackspace instances.[1] This implies that if a tenant were placing a single application on the Rackspace network, there would be virtually no variation for Choreo to exploit. However, Choreo is still effective in this type of network when a tenant is placing multiple applications in sequence, as traffic from existing applications causes variations in achieved TCP throughput (§4.3.2).

In all our experiments, we have found that both of these networks are rate-limited and implement a *hose model* [30] to set a maximum outgoing rate for each VM. Figures 6-3(a) and 6-3(b) give evidence for such rate-limiting, as the typical observed throughput on a path remains close to a particular value for each network (1 Gbit/s for EC2, 300 Mbit/s for Rackspace). We develop techniques to confirm both of these hypotheses in §6.4, and verify them on EC2 and Rackspace in §6.5.3.

Our approach to estimating achievable TCP throughput in these figures was to use `netperf`, but this approach induces significant overhead for online use. Choreo must also measure the network quickly, so that applications can be placed in a timely manner. Moreover, if Choreo's measurement phase takes too long, there is a chance the network could change between measurement time and placement (we discuss the temporal stability of public cloud networks in §6.5.1).

To overcome this issue, Choreo uses packet trains to measure achievable TCP throughput. We discuss this method, as well as Choreo's methods for assessing rate-limits and bottlenecks, in §6.4. Briefly, packet trains allow Choreo to get a snapshot of the network within just a few minutes for a ten-node topology, including the overhead of sending measurements from each machine back to a centralized server.

6.4 Measurement Techniques

In order for Cicada to place an application on a set of virtual machines, Choreo must measure three things. First, the achievable TCP throughput on each path, which tells Choreo what the possible throughput of a single connection on that path is. Second, the amount of cross traffic in the network, which tells Choreo how multiple connections on the same path will be affected by each other. For instance, if one connection gets a throughput of 600 Mbit/s on a path, it is possible that two connections on that path will each see 300 Mbit/s—if they are the only connections on a path with link speed

[1] `http://rackspace.com/cloud/servers/pricing`

of 600 Mbit/s—or that they will each see 400 Mbit/s—if there is one background TCP connection on a path with a link speed of 1200 Mbit/s. Third, which paths share bottlenecks, which tells Choreo how connections between different pairs of VMs will be affected by each other. For instance, knowing that $A \rightsquigarrow B$ shares a bottlenecks with $C \rightsquigarrow D$ informs Choreo's estimated achievable throughput of a new connection on $A \rightsquigarrow B$ when connections exist on $C \rightsquigarrow D$.

6.4.1 Measuring Pairwise Throughput

The first measurement that Choreo does is to estimate the pairwise achievable TCP throughput between the cloud VMs, to understand what the throughput of a single connection will be. Estimating the pairwise achievable TCP throughput between N VMs by running bulk TCP transfers takes a long time for even modest values of N. Packet trains, originally proposed in [46] and also used in [25], have been adapted in various ways, but with only varying success over Internet paths [72, 75]. The question is whether more homogeneous and higher-rate cloud infrastructures permit the method to be more successful.

Choreo sends K bursts of P-byte UDP packets, each burst made up of a sequence of B back-to-back packets.[2] Bursts are separated by δ milliseconds to avoid causing persistent congestion. This collection of K bursts is one packet train; to estimate achievable throughput on a path, we send only one packet train on that path.[3]

At the receiver, Choreo observes the kernel-level timestamps at which the first and last packet of each burst b_i was received using the `SO_TIMESTAMPNS` socket option, and records this time difference, t_i. If either the first or last packet of a burst is lost (which we can determine via the sequence numbers we inserted into the UDP payload), we adjust t_i to take into account what the time difference should have been, by calculating the average time it took to transfer one packet and adjusting for the number of packets missing from the beginning or end of the burst. We also record the number of packets received in each burst, $n_i \leq B$.

Using this data, we estimate the achievable TCP throughput as

$$\frac{P \cdot \sum_{i=1}^{K} n_i}{\sum_{i=1}^{K} t_i}.$$

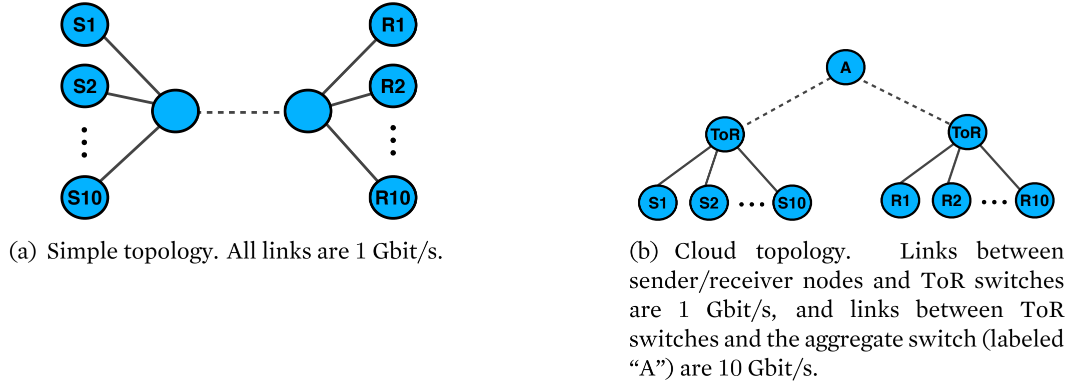
This throughput estimate is equivalent to estimating the throughput as

$$\frac{P(N-1)(1-\ell)}{T},$$

where N is the number of packets sent, ℓ is the packet loss rate, and T is the time between the receipt of the first packet and the last packet. An alternative method for estimating achievable TCP throughput is to use the formula

[2] In our experiments, "packet pairs", in which only two packets are sent, were not very accurate on any cloud environment, and so Choreo does not use that method.

[3] We use a slightly different terminology than Jain uses in [46]. What we call a burst, Jain calls a train; what we call a train is a sequence of Jain's trains.



(a) Simple topology. All links are 1 Gbit/s.

(b) Cloud topology. Links between sender/receiver nodes and ToR switches are 1 Gbit/s, and links between ToR switches and the aggregate switch (labeled "A") are 10 Gbit/s.

Figure 6-4: ns-2 simulation topologies for our cross traffic method. The dashed lines are bottlenecks where cross traffic will interfere.

$$\frac{MSS \cdot C}{RTT \cdot \sqrt{\ell}},$$

where C is the constant of proportionality, roughly $\sqrt{3/2}$ [60]. This formula, however, is an upper-bound on the actual throughput, and departs from the true value when ℓ is small, and is not always useful in datacenter networks with low packet loss rates. Our estimator combines these two expressions and estimates the throughput as:

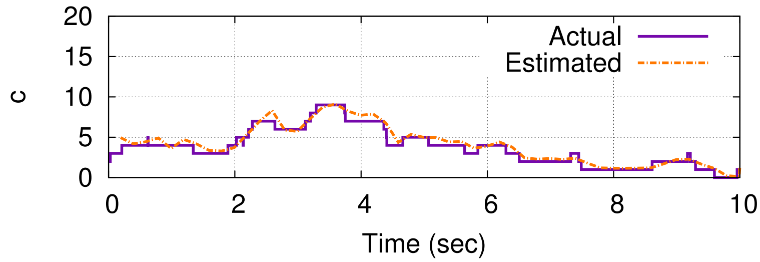
$$\min\left\{ {}^{P(N-1)(1-\ell)}/_{T}, {}^{MSS \cdot C}/_{RTT \cdot \sqrt{\ell}} \right\}.$$

§6.5.1 evaluates this method on EC2 and Rackspace.

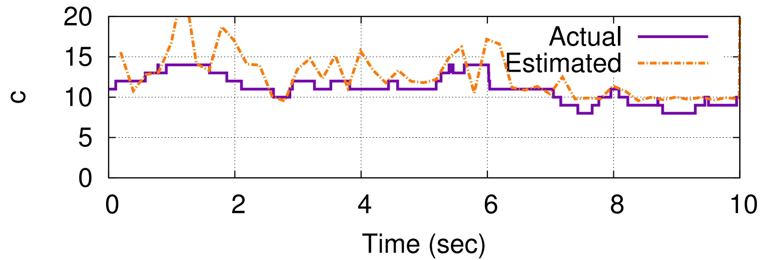
6.4.2 Estimating Cross Traffic

The second measurement that Choreo makes estimates cross traffic on the network, in order to understand how multiple connections on the same path will be affected. As part of this task, Choreo also needs to understand how the cross traffic on a path varies over time; e.g., does the number of other connections on a path remain stable, or does it vary significantly with time?

To estimate the "equivalent" number of concurrent bulk TCP connections along a path between two VMs, we send one bulk TCP connection between two instances (e.g., using `netperf`), run it for ten seconds, and log the timestamp of each receiving packet at the receiver. This method allows us to measure the throughput of the connection every ten milliseconds. Given the maximum path rate, the throughput of our connection should indicate how many other connections there are. For instance, if the path rate is 1 Gbit/s, and our connection experiences a throughput of 250 Mbit/s, then there is the equivalent of three other bulk TCP connections on the path. In general, if the path rate is c_1, and our connection experiences a throughput of $c_2 \leq c_1$, then there are $c = {}^{c_1}/_{c_2} - 1$ other bulk connections on the path. We measure the



(a) Simulation results for $c < 10$ on a simple topology.



(b) Simulation results for $c \geq 10$ on a more realistic cloud topology.

Figure 6-5: ns-2 simulation results for our cross traffic method. We are able to quickly and accurately determine the number of background connections, particularly for small values of c, even when the number of background connections changes rapidly.

throughput frequently to account for the fact that background connections may come and go.

In this method, we have made a few assumptions: first, that we know the maximum rate of the path; second, that TCP divides the bottleneck rate equally between bulk connections in cloud networks; and third, that the background connections are backlogged, sending as fast as possible. In public clouds, it is not hard to obtain a maximum link rate (e.g.., it is usually 1 Gbit/s today on EC2) using simple measurements or because the provider advertises it (e.g., Rackspace). If the maximum rate is unknown, however, then we can solve this problem by sending one connection on the path, and then two connections; the change in the observed throughputs will allow us to estimate c.

For the second assumption, we ran `netperf` on the EC2 network and observed that when one connection achieved 1 Gbit/s of throughput (the maximum rate), the rate did decrease by roughly 50% when we added a second connection. This result is unsurprising, given the homogeneity of cloud networks, and the fact that both connections were capable of sending at the maximum rate.

As for the third assumption, our method of estimating c also gives us a reasonable estimate of how achievable throughput will be affected in many cases even when the other connections are not backlogged. For instance, if there is one background

connection on a 1 Gbit/s link that has an offered load of only 100 Mbit/s, our measured throughput will be 900 Mbit/s, and our estimated throughput of two connections on that link will be 450 Mbit/s. Although Choreo will incorrectly assume that there is no background traffic, it will not be a problem in terms of achievable throughput estimates until Choreo tries to place a significant number of connections on the path (ten, in this example—enough to decrease the rate of the initial 100 Mbit/s connection).

The quantity c is an estimate of the cross traffic on the bottleneck link, measured in terms of the equivalent number of concurrent bulk transport connections. We should interpret the quantity c as a measure of load, not a measure of how many discrete connections exist. A value of c that corresponds to there being one other connection on the network simply means that there is load on the network equivalent to one TCP connection with a continuously backlogged sender; the load itself could be the result of multiple smaller connections.

To test this method, we used ns-2 simulations. First, we simulated a simple topology where ten sender-receiver pairs share one 1 Gbit/s link, as shown in Figure 6-4(a). In this simulation, the pair $S_1 \rightsquigarrow R_1$ serves as the foreground connection, which transmits for ten seconds, and the rest ($S_2 \rightsquigarrow R_2, \ldots, S_{10} \rightsquigarrow R_{10}$) serve as the background connections, and follow an ON-OFF model [4] whose transition time follows an exponential distribution with $\mu = 5$s. Figure 6-5(a) shows the actual number of current flows and the estimated value using c. Here, $c_1 = 1$ Gbit/s. Second, we simulated a more realistic cloud topology, shown in Figure 6-4(b). Figure 6-5(b) shows the actual and estimated number of cross traffic flows. In this cloud topology, the links shared by cross traffic are links between ToR switches and an aggregate switch, A. Because these links have a rate of 10 Gbit/s each, compared to the 1 Gbit/s links between sender/receiver nodes and their ToR switches, the cross traffic throughput will decrease only when more than ten flows transfer at the same time. Thus, in Figure 6-5(b), the smallest estimated value is ten.

The difference in accuracy between Figure 6-5(a) and Figure 6-5(b) comes from the possible throughputs when c is large. For instance, given a 1 Gbit/s link, if we see a connection of 450 Mbit/s, we can be relatively certain that there is one existing background connection; the expected throughput was 500 Mbit/s, and 450 Mbit/s is not far off. However, if we see a connection of 90 Mbit/s, multiple values of c (e.g., $c = 10, c = 11$) are plausible.

§6.5.2 describes the results of this method running on EC2 and Rackspace.

6.4.3 Locating Bottlenecks

Choreo's final measurement determines what paths share bottlenecks, which it can use to determine how connections on one path will be affected by connections on a different path. There is a long-standing body of work on measuring Internet topologies [50, 89, 91]. Typically, these works use `traceroute` to gain some knowledge of the topology, and then use various clever techniques to overcome the limitations of `traceroute`. Datacenter topologies, however, are more manageable than the Inter-

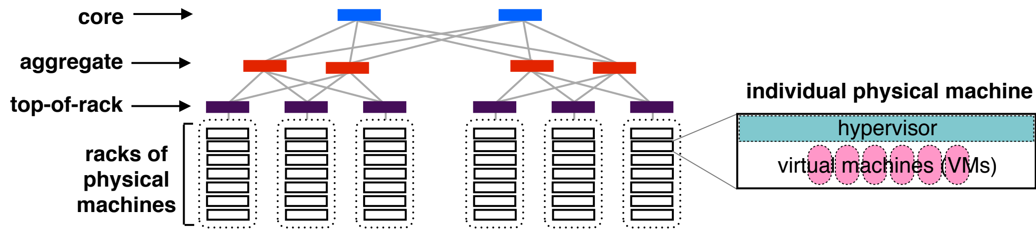


Figure 6-6: A typical datacenter network (also shown in Figure 2-1).

net topology; typically they have a structured, hierarchical form. In this section, we explore how `traceroute` measurements combined with topological assumptions can allow us to discover features of datacenter networks that we typically cannot discover in the Internet at-large.

Typical Datacenter Topologies

As explained in Chapter 2, datacenter topologies are often multi-rooted trees (Figure 2-1, which we repeat here as Figure 6-6 for convenience). In these topologies, virtual machines lie on physical machines, which connect to top-of-rack (ToR) switches. These switches connect to aggregate switches above them, and core switches above those. To route between virtual machines on the same physical machine, traffic need travel only one hop. To route between two physical machines on the same rack, two hops are needed: up to the ToR switch and back down. To route to a different rack in Figure 6-6, either four or six hops are needed, depending on whether traffic needs to go all the way to the core. In general, all paths in this type of topology should use an even number of hops (or one hop). If we assume that a datacenter topology conforms to this structure, inferring where machines are placed in the topology is easier; in some sense, we can make the same measurements as traditional techniques, and "fit" a multi-rooted tree onto it.

Locating Bottlenecks

With this knowledge of the topology, we turn our attention to determining whether two paths share a bottleneck link, and where that bottleneck link occurs: at the link to the ToR switch, the link from the ToR switch to the aggregate layer, or the link to the core layer. One way to determine whether two paths share a bottleneck is by sending traffic on both paths concurrently. To determine whether path $A \rightsquigarrow B$ shares a bottleneck with $C \rightsquigarrow D$, we send `netperf` traffic on both paths concurrently. If the achieved throughput on $A \rightsquigarrow B$ decreases significantly compared to its original value, we infer that $A \rightsquigarrow B$ shares a bottleneck with $C \rightsquigarrow D$.

We are interested in whether a connection from $A \rightsquigarrow B$ will interfere with one from $C \rightsquigarrow D$. We note the following rules:

1. If the bottleneck is on the link out of the ToR switch, the two connections will interfere if either of the following occur:
   (a) they come from the same source, i.e., A's physical machine is the same as C's.
   (b) A and C are on the same rack, and neither B nor D is on that rack.
2. If the bottleneck is on the link out of the aggregate layer and into the core, the two connections will potentially interfere if they both originate from the same subtree and must leave it, i.e., if A and C are on the same subtree, and neither B nor D is also located on that subtree. Note that even in this case, the connections may not interfere; $A \rightsquigarrow B$ may not get routed through the same aggregate switch as $C \rightsquigarrow D$.

These rules allow Choreo to estimate bottlenecks more efficiently than a brute force method that measures all pairs of paths. Because we can cluster VMs by rack, in many cases, Choreo can generalize one measurement to the entire rack of machines. For instance, if there is a bottleneck link on the ToR switch out of rack R, then any two connections out of that rack will share a bottleneck; Choreo does not need to measure each of those connections separately.

Choreo's bottleneck-finding technique can also determine what type of rate-limiting a cloud provider may be imposing. For instance, if the datacenter uses a hose model, and rate-limits the hose out of each source, our algorithm will discover bottlenecks at the end-points—indicating rate-limiting—and that the sum of the connections out of a particular source remains constant—indicating a hose model.

§6.5.3 evaluates this method on EC2 and Rackspace.

6.4.4 Exploring the Network with Additional VMs

All of Choreo's techniques involve measuring the paths between the virtual machines that the customer already owns. As such, there may be better paths in the network that it does not have access to. A possible extension to Choreo is to allow Choreo to launch additional VMs, to "explore" other paths on the network.

There is certainly a trade-off here: additional VMs cost extra money, and the benefit that they bring may not be significant (e.g., the additional VMs may not find any better paths in the network that the ones Choreo already knew about). We believe that this trade-off can be made precise with the following knowledge: the cost of launching an additional VM, and a distribution (or appropriate distribution) of TCP throughput in the network, similar to Figure 6-3(a)). Then tenants can judge whether their current VMs have atypically poor paths, in which case they may be willing to launch additional VMs in order to find new ones.

6.5 Evaluation

We evaluated the techniques described in the previous section on EC2 and Rackspace to validate the techniques and measure how these two cloud networks perform.

6.5.1 Packet Train Accuracy and Temporal Stability

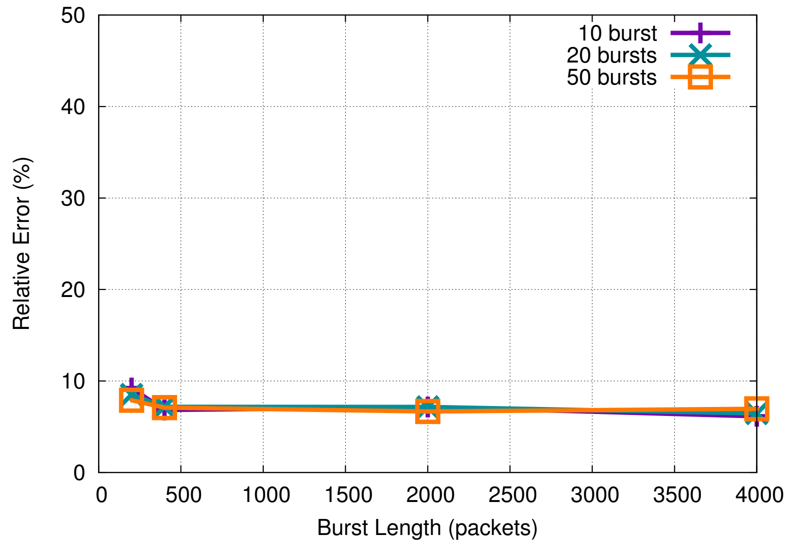
The packet train method from §6.4.1 is parameterized by the packet size, P, the number of bursts, K, and the length of each burst, B. To tune these parameters, we compared the accuracy of various packet sizes, burst lengths, and number of bursts, against the achieved throughputs measured with `netperf` on 90 paths in EC2 and Rackspace (we use the `netperf` throughputs as the "ground truth" for this experiment). Each path gives us one datapoint (one `netperf` measurement and one packet train estimate). Figure 6-7 displays the accuracy of estimating achievable TCP throughput via packet trains averaged over all paths, for a packet size of 1472 bytes and a time δ of 1 millisecond.

In general, we see from Figure 6-7 that the accuracy improves with the burst length and number of bursts, as expected. Beyond a point, however, the accuracy does not improve (diminishing returns). We found that 10 bursts of 200 packets works well on EC2, with only 9% error, and sending 10 bursts of 2000 packets worked well on Rackspace, with only 4% error (this configuration also works well on EC2). In our measurements, an individual train takes less than one second to send, compared to the ten seconds used by `netperf` (in our experiments, we found that using `netperf` for a shorter amount of time did not produce a stable throughput estimate). To measure a network of ten VMs (i.e., 90 VM pairs) takes less than fifteen seconds in our implementation, including the overhead of setting up and tearing down tenants/servers for measurement, and transferring throughput data to a centralized server (see §6.5.4).

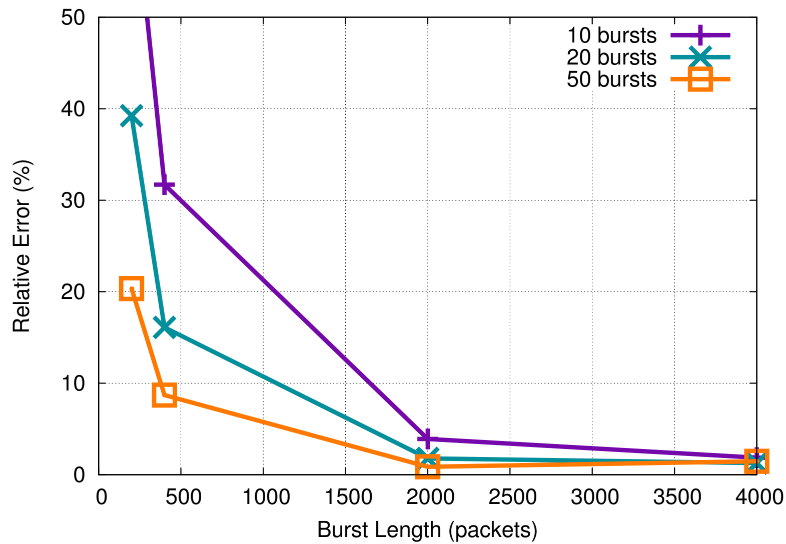
Because the best packet train parameters for EC2 and Rackspace differ, before using a cloud network, a tenant should calibrate their packet train parameters using an approach similar to the one proposed above. This phase takes longer than running only packet trains, but needs to be done much less frequently (possibly only once, although if a provider changes its network infrastructure, the trains likely should be re-calibrated). We discuss how the parameters effect scalability in §6.5.4.

Although Choreo can get an accurate snapshot of the network quickly using packet trains, such a snapshot will not be useful if the network changes rapidly, i.e., has low temporal stability. In this case, Choreo will have trouble choosing the "best" VMs on which to place tasks. To measure temporal stability, we used 258 distinct paths in Amazon EC2, and 90 in Rackspace. On each of these paths, we continuously measured the achieved TCP throughput of 10-second `netperf` transfers for a period of 30 minutes, giving us one throughput sample every ten seconds for each path.

We analyze temporal throughput stability by determining how well a TCP throughput measurement from τ minutes ago predicts the current throughput. Given the current throughput measurement, λ_c, and the throughput τ minutes ago, $\lambda_{c-\tau}$, Fig-



(a) EC2

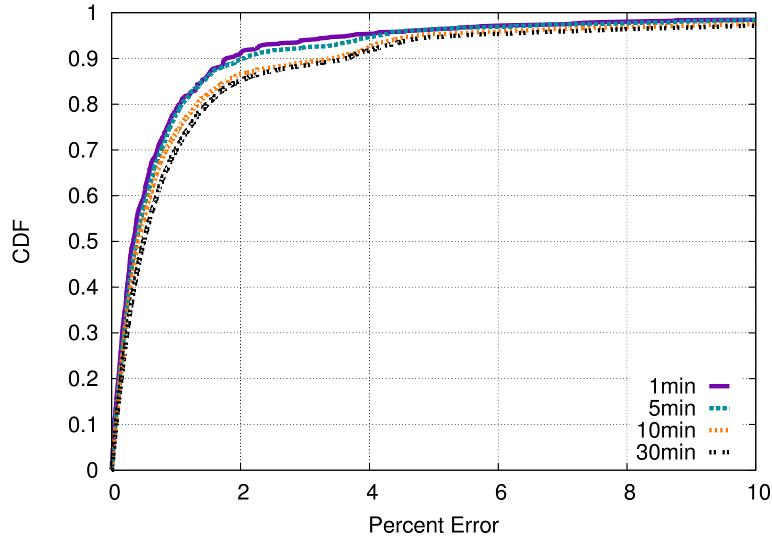


(b) Rackspace

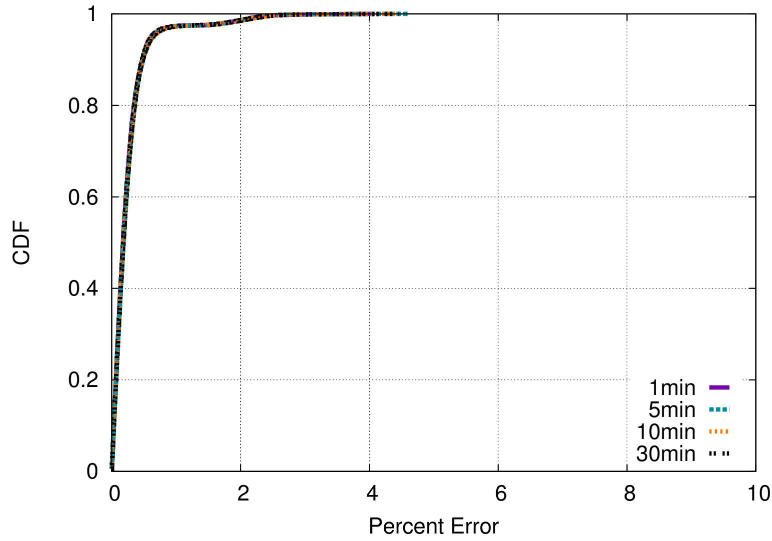
Figure 6-7: Percent error for packet train measurements using different burst lengths and sizes. EC2 displays a consistently low error rate over all configurations, while Rackspace's error rate improves dramatically once the burst length is at least 2000.

ure 6-8 plots a CDF of the magnitude of the relative error, i.e., $|\lambda_c - \lambda_{c-\tau}|/\lambda_c$, for various values of τ ($1, 5, 10$, and 30 minutes). In EC2, for every value of τ, at least 95% of the paths have 6% error or less; the median error varies between $.4\%$ and $.5\%$,

while the mean varies between 1.4% and 3%. As suspected from Figure 6-3(b), the error in Rackspace is even lower: at least 95% of the paths have 0.62% error or less; the median varies between 0.18% and 0.27%, while the mean varies between 0.27% and 0.39%.



(a) EC2. More recent measurements are more accurate, but even with values of $\tau = 30$min, over 90% of paths see 4% error or less.



(b) Rackspace. The results for Rackspace are virtually the same for all values of τ, hence the appearance of only one line on the graph.

Figure 6-8: Percent error when a measurement from τ minutes ago is used to predict the current path throughput.

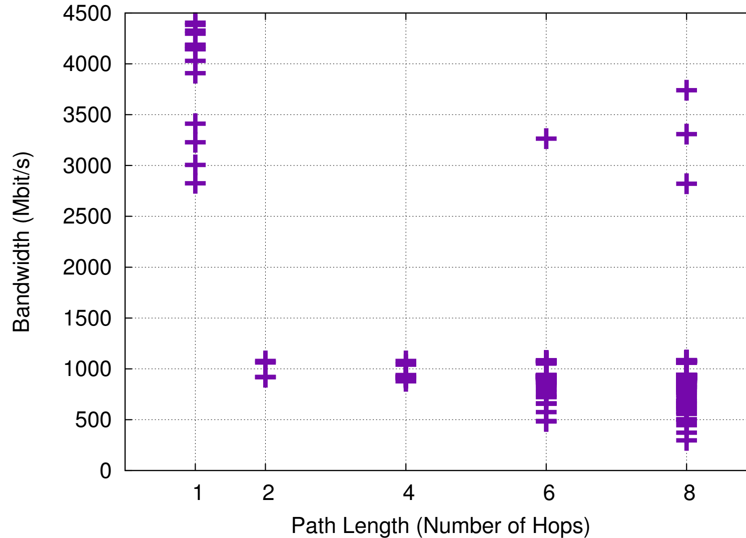


Figure 6-9: Comparison of path length with achievable throughput. Path length is not entirely correlated with throughput, as evidenced by the eight-hop paths with throughput over 2500 Mbit/s.

Thus, we find that although some path throughputs vary spatially, they exhibit good temporal stability. Hence, Choreo can measure path throughputs infrequently to gather useful results for making decisions about application placement. The temporal stability result is somewhat surprising because EC2 and Rackspace are popular cloud infrastructures, and one would expect significant amounts of cross traffic. In the next section, we measure the amount of cross traffic on these networks, finding that it is small. We then explore the locations of the bottlenecks, finding that the hose model provides a good explanation of the currently observed network performance on EC2 and Rackspace.

6.5.2 Cross Traffic

We first return to a result from §6.3, shown in Figure 6-3(a). Initially, we expected that achievable path throughput would be correlated with path length, in part because it should correlate with physical topology: the longer the paths, the more likely they are traversing the middle of the network, and thus cross traffic there would be to interfere and lower the throughput.

Figure 6-9 plots the various path lengths observed over a collection of 1710 paths in EC2; these are the same paths that we used in §6.3. The first thing to note from this figure is that the path lengths are only in the set $\{1, 2, 4, 6, 8\}$. These lengths are consistent with a multi-rooted tree topology. Second, many of the paths are more than one or two hops, indicating that a significant number of VMs are not located on

the same rack. In the Rackspace network (not shown in the picture), we saw paths of only one or four hops. Because the achievable throughputs in the Rackspace network are all approximately 300 Mbit/s, there is no correlation with path length. It is also curious that we do not see paths of any other lengths (e.g., two); we suspect that Rackspace's `traceroute` results may hide certain aspects of their topology, possibly via packet encapsulation, although we cannot confirm that.

From Figure 6-9, we see that there is little correlation between path length and achievable throughput. In general, we find that the pairs with the highest achievable throughput are one hop apart, an unsurprising result as we would expect the highest-throughput pairs to be on the same physical machine. However, there are four high-throughput paths that are six or eight hops away. Moreover, although we see that the lower throughputs tend to be on longer paths, we also see that a "typical" throughput close to 1 Gbit/s appears on all path lengths. This result leads us to believe that there is very little cross traffic that affects connections on the EC2 network, so we did not run our cross traffic algorithm from §6.4.2 on it; instead, we turned our attention towards locating bottlenecks, speculating that the bottlenecks may be at the source. This result would imply that Choreo can determine how multiple connections on one path are affected by each other simply by knowing the achievable throughput of that path.

6.5.3 Bottleneck Locations and Cross Traffic

From Figure 6-9, we hypothesized that Amazon rate-limits at the source, and that cross traffic in the middle of the network has very little effect on connections (and thus the bottleneck link is the first hop). To test this hypothesis, we ran Choreo's bottleneck-finding algorithm from §6.4.3. We ran an experiment on twenty pairs of connections between four distinct VMs, and twenty pairs of connections from the same source. We found that concurrent connections among four unique endpoints never interfered with each other, while concurrent connections from the same source always did. This result indicates that EC2 bottlenecks occur on the first hops of the paths. Figure 6-8 also supports this conclusion, as paths with little cross traffic tend to be relatively stable over time.

We also hypothesized that rate-limiting at the source was prevalent on Rackspace, since most path throughputs are so close to 300 Mbit/s, and verified that connections out of the same source always interfered. These results imply that connections placed on a particular path are affected by other connections out of the source, not just other connections on the path. However, they are not affected by connections emanating from a different source.

6.5.4 Scalability

Choreo's throughput measurements take fewer than fifteen seconds on a ten-VM topology. Using packet trains rather than longer-lived TCP connections (e.g., as

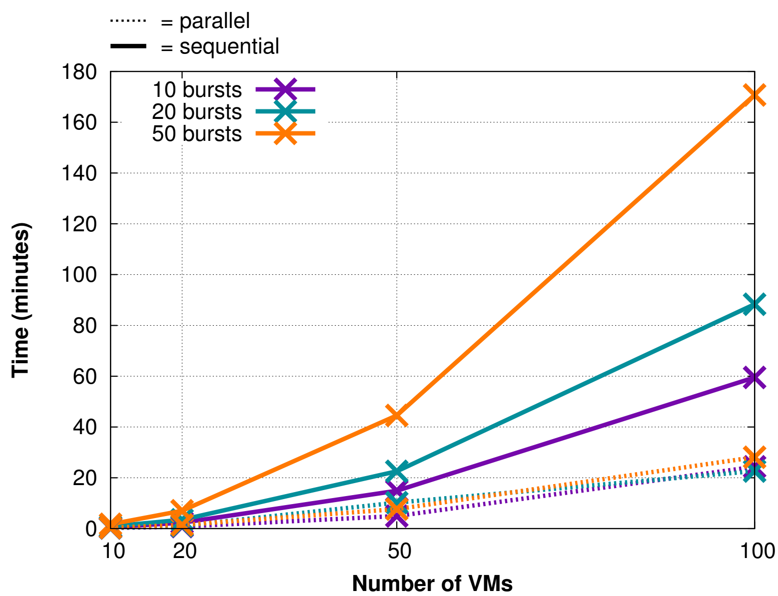


Figure 6-10: Time it takes for Choreo to measure an n-VM topology using a burst length of 400 packets, and varying burst sizes (n varies with the x-axis). Choreo's measurements perform faster when done in parallel, though at the cost of accuracy in many cases; see Figure 6.5.4 and the accompanying discussion.

would be sent by `netperf`) is the main feature that decreases Choreo's measurement time, but we were able to decrease measurement time further by doing the following:

- Maintaining persistent SSH connections to the measurement VMs. This decreases the overhead of connecting and closing a connection.
- Keeping the measurement server inside of the cloud (this resulted in only a slight performance improvement; it took around fifteen seconds longer to measure a ten-VM topology using a measurement server outside of the cloud).
- Choosing the correct parameters for the packet trains.

In Figure 6-7, we showed that on EC2, Choreo's packet trains were accurate for a large range of numbers of bursts and burst lengths. However, the smaller the values of these two parameters, the less time measurements take.

Figure 6.5.4 presents the time it takes Choreo to measure an n-VM ($n \in \{10, 20, 50, 100\}$) using a burst length of 400 packets and varying burst sizes; each of these parameter configurations had relative error less than 10% (see Figure 6-7). When Choreo performs its measurements sequentially, the measurement time scales with $O(n^2)$, as it must measure each pair, and only one pair at a time. The impact of sending fewer bursts becomes apparent as n grows; we see almost a three-fold increase between sending 10 bursts and 50 bursts when $n = 100$.

For up to 50 VMs, Choreo is capable of measuring the entire topology in fewer than twenty minutes. Today's tenants are typically much not larger than this [20]. Moreover, based on the stability of the EC2 network (Figure 6-8), this is a short enough time period that the measurements are still accurate. However, customers may not want to wait twenty minutes before their application can be placed. One option is to run Choreo continuously, so that measurements are ready when an application arrives. Another option is to run the measurements in parallel.

Figure 6.5.4 also plots the same results from parallel measurements. In this case, measurements scale with $O(n)$, not $O(n^2)$, and even a 100-VM topology only takes around twenty minutes to measure. However, there is a concern when doing active measurements in parallel: measurement traffic may interfere, and the resulting measurements will not reflect the true values.

Figure 6.5.4 plots the relative error of packet trains when run in parallel (the same type of figure as in Figure 6-7). In our implementation, we start each train at roughly, but not exactly, the same time (we allow up to $.1$ seconds of time between the start of trains), as we found that this improved accuracy.

For many parameter configurations, the relative error is unacceptable; near 50% as the amount of traffic grows. However, for some configurations where each train sends very little traffic (e.g., twenty bursts of 200 packets each), the accuracy is near 10%, which is comparable to the accuracy of sequential packet trains. This result comes from the fact that with less traffic per train, it is less likely that trains interfere.

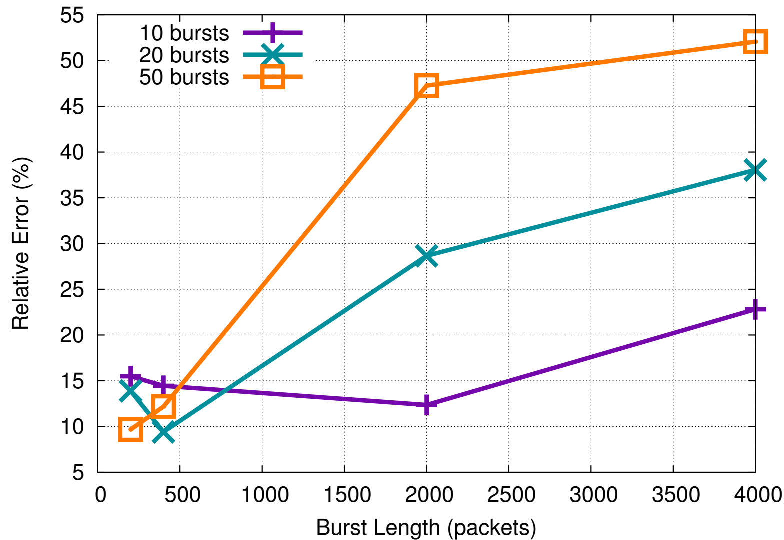


Figure 6-11:

We hesitate to recommend using parallel trains on today's clouds. For today's tenants, which typically do not require hundreds of VMs [20], sequential trains are fast and accurate across many configurations; parallel trains are also fast, but are accurate for only a few of our tested configurations. As tenants grow larger, it may be worth exploring parallel trains in more depth.

6.5.5 Discussion

In summary, these results suggest that the EC2 and Rackspace networks are both rate-limited and use the hose model. Rackspace has very little spatial variability in throughput; EC2 has some, but the overwhelming majority of paths see similar throughputs near 1 Gbit/s.

Despite this lack of a lot of natural variation in the two public cloud networks, Cicada is still able to exploit variation that comes with a tenant running multiple applications at once (see §4.3), and to avoid the few paths in EC2 that do see low throughput. Moreover, paths on these networks are temporally stable, indicating that Choreo does not need to update its measurements very frequently to take into account high-variance paths.

We also note that cloud networks can change over time. In fact, our results from Figure 6-2 indicated a larger range of spatial variation. For that reason, we believe that Choreo's measurement techniques are worthwhile beyond simply measuring a tenant's own applications, because higher network variation could appear in other cloud networks, or in EC2 and Rackspace in the future.

One piece of future work is studying how the accuracy of Choreo's measurements

trades off with how much Cicada can improve application performance. Though Choreo's measurement phase does not require much overhead, it is possible that Cicada would still be able to improve performance with even less accurate measurements. For instance, if Choreo's measurements were only 75% accurate, as opposed to approximately 90% accurate (as in §6.5.1), would the performance improvement also fall by 15%, or only by a few percent?

6.6 Conclusion

In this chapter, we studied the extensions that Cicada requires in order to be run without any provider assistance. We developed Choreo, a network-measurement module that allows customers to measure the network between their virtual machines. Choreo allows customers to get their own "network snapshot", which Cicada then uses as input to its placement algorithm (Algorithm 4).

Choreo lets customers measure public cloud networks quickly and accurately. We found that packet trains could be used to measure public cloud networks, with a mean error of only 9% on EC2, and a mean error of 4% on Rackspace. As shown in Chapter 4, Choreo's measurements are accurate enough that applications see improvement over existing placement methods.

Chapter 7

Conclusion

In this dissertation, we presented a system, Cicada, aimed at solving the problem of improving the performance of long-running, network-intensive cloud applications. Cicada predicts application traffic patterns, and uses these to intelligently place the applications on the cloud network. Cicada's prediction and placement algorithms can be used to improve application completion time and utilization (Chapter 4), lower monetary cost, enforce fault tolerance (Appendix A), and a variety of other objectives.

Cicada can be used either by a cloud customer or a cloud provider. To enable customer-only use, Cicada must be paired with its measurement module, Choreo, which allows the customers to perform their own network measurements (Chapter 6; the measured values are already available to the provider). When run by a provider, Cicada can be used to offer customer bandwidth guarantees (Chapter 5), which can improve the predictability of cloud applications [20, 96], as well as encourage companies that have their own service contracts to uphold to use public clouds,

Using traces from real cloud networks, we showed that Cicada meets its goals. Cicada can accurately predict traffic for most cloud applications (and recognize those for which it cannot make prediction; §3.5), and it improves application completion time and network utilization (§4.3). Choreo can quickly measure real cloud networks with novel techniques (§6.5), allowing customers to run Cicada without the provider's help.

7.1 Critiques

Why use application-level techniques rather than network-level ones?

Cicada does not impose a particular network topology, nor make any changes to the routing algorithm as related work does [27, 34]. Our original motivation for using *application-level* techniques was so that we could develop techniques that could be used immediately—topology or routing changes often require hardware upgrades—and often by the customer alone (the latter is particularly the case when Cicada is paired with Choreo).

However, we do not view this dissertation as a triumph of application-level techniques over network-level ones. Application-level techniques are simply a different approach. They have their benefits (particular with regards to ease of deployment), but there is no reason for cloud providers or customers to make a decision between application-level and network-level solutions; both can coexist.

Are providers really motivated to use Cicada?

We believe that providers will be motivated to allow customers to use Cicada, or to use Cicada themselves.

For instance, if providers use Cicada to offer bandwidth guarantees, companies that have their own service contracts to uphold are more likely to use the public cloud services that these providers provide. As today's clouds offer little in the way of network QoS or SLAs to their customers, some companies avoid public clouds [23, 52, 62]. For better or for worse, accurate bandwidth guarantees also give the providers means to charge their customers for internal bandwidth usage. As shown in §4.3.3, Cicada also increases network utilization, another benefit for providers.

Cicada's method of placing applications to minimizing application completion time may face criticism from providers. As we showed in Chapter 4, Cicada can appreciably decrease application completion times. At first glance, it may seem that this decrease in application time results in a decrease in provider revenue: customers do not need to rent machines for as long.

However, decreased application completion time means that providers can pack more tenants onto the network. In most cloud pricing schemes, there is an up-front payment to launch the machines, and then smaller payments over time to keep them running. As a result, providers can make more money in a time period by renting machines to many tenants than they do renting the same machines to fewer tenants. Moreover, offering decreased completion time is a competitive advantage for cloud providers, and may encourage customers to do *more* work on the cloud, leading to increased revenue.

Providers may also object to customers using Choreo to obtain network measurements, either because they worry about active measurement techniques injecting a lot of traffic into the network and affecting other tenants, or because they worry about tenants inferring confidential aspects of their networks (e.g., topology). In response to the first criticism, we note that Choreo was designed to send very little traffic on the network (§6.4). Its measurement traffic should have little to no effect on other tenants.

Regarding the second criticism, it is hard to deny that Choreo may learn things about the network that providers would like to keep secret. But such is the nature of any network measurement system. The network topology, e.g., is reflected in the achieved throughput of various connections and how those throughputs change. If a provider desires to keep this information entirely a secret, then he must change this effect, and lose a lot of benefits of the topology itself.

7.2 The Future

Can we apply these techniques to all applications on all public clouds?

Cicada was designed with long-running, network-intensive applications in mind. Though it is a general framework, there are certain applications for which it likely will not provide any improvement. For example, for applications that only need a few minutes to run, the cost of Cicada's measurement phase would cancel out any improvement. Applications that have relatively uniform network-usage also would not see much improvement from Cicada, because there is not a lot of room for improvement. Here, because every pair of VMs sends roughly the same amount of data, it does not help to put the "largest" pair on the fastest link. We observed this traffic pattern in some MapReduce applications.

Interactive applications may not see much improvement from the version of Cicada described here, as the interactive phases would likely be difficult to model as they are dependent on user activity. However, Cicada could re-evaluate its placement after every interactive phase for some applications; as long as the phases could be correctly identified, Cicada may be useful.

Cicada was tested on applications observed in HP Cloud Services, and tested with experiments on Amazon EC2 and Rackspace. We believe that Cicada will offer improvement on most other cloud networks, but not all. For instance, Cicada will not offer a great deal of improvement in cloud environments with an abundance of network bandwidth. After all, if the cloud provider can afford to give every pair of tasks a dedicated path, for instance, there is not much need to optimize task placement. Cicada will offer more improvement as cloud networks become more utilized. Although we do not believe that today's cloud networks are blessed with an excess of network bandwidth, they also may not be as heavily utilized as they could be; thus, Cicada may offer more more substantial improvements as cloud networks evolve and become more heavily utilized.

Can we apply these techniques to other types of networks?

There is nothing stopping future researchers for using our techniques on more general networks. Though proposed bandwidth guarantees are somewhat specific to public clouds, the idea of predicting application traffic to improve network utilization or application performance is not. Measuring networks and placing applications intelligently also transcends public clouds. However, we do offer some caveats below:

- Cicada was tested on public cloud applications. Its techniques do not make any assumptions about the nature of these applications, but they have not been evaluated on every type of application. It is possible that Cicada's prediction algorithm may not work on other applications, or would need to be modified.

- Choreo's measurement techniques work well in public cloud environments partly due to the homogeneity and speeds of these networks. Some of its techniques—notably, packet trains—will likely *not* work as well on all networks, and any assumptions it uses about the underlying network topology will need to be adjusted if a different network topology is used. However, if measurement data can be obtained in other ways, and if the underlying network is stable (and its topology at least somewhat known), it is likely that Choreo's techniques (and thus Cicada's) could be used elsewhere.

Longevity of Cicada

Cloud networks are an evolving environment. The way providers manage their network, including enforcing rate-limits, or migrating VMs, could change, and it is difficult to predict how Cicada will be affected by these changes. However, because Cicada starts by measuring the cloud network, we believe that it could provide improvement—or, at the very least, not be detrimental—in the face of some of these changes. For example, if rate-limits are enforced, Cicada should be able to pick up on that when it measures the links (either through provider-provided measurements, or via Choreo). If VMs are migrated during an application's run, Cicada will account for that, as it measures periodically. Determining the correct frequency for these measurements would likely require some fine-tuning.

Today's version of Cicada may not be appropriate for cloud networks years from now. But we believe that the general approach of predicting application workloads, measuring the network, and adapting to it will remain useful, and that future versions of Cicada could be built under the same principles to adapt to future cloud networks.

Appendix A

ILPs for Expressing Additional Goals in Cicada

Our evaluation in Chapter 4 focused around the goal of minimizing an application's completion time. However, it is possible that an application may have other goals, such as minimizing latency or increasing fault tolerance, or even goals unrelated to the network, such as minimizing the monetary cost to the user. Users may also want to impose more constraints on CPU or storage systems. The basic principles of Cicada can be extended to all of these types of goals. In fact, Cicada can support any goal that can be formulated as an optimization problem. We give examples here.

Minimizing Latency

Rather than requiring a change to the optimization problem, minimizing latency would require a change to Cicada's network measurements. Instead of expressing the network as a matrix of achieved TCP throughputs between virtual machines, we would instead express it as the latency between machines.

It would be easy for Cicada to determine latency. This value is available to cloud providers, or can be measured with pings via Choreo (this measurement could be piggy-backed onto the packet trains Choreo already uses to measure TCP throughput).

One challenge Cicada would face in this scenario is making sure its latency estimate is stable, but this can typically be done by sending repeated pings [14].

Fault Tolerance

In addition to the CPU constraints already expressed in §4.2.2, a customer can express her fault-tolerance constraints by stating the tasks that must be located on separate physical machines, across datacenters, or multiple hops away (this fine placement-granularity is not directly supported by most public clouds today, though techniques exist for determining whether two virtual machines are placed on the same physical server [83], and in some cases, IP addresses can be used to determine how close two virtual machines are in the physical topology [80]). As a simple example, to prevent two tasks i and j from being placed on the same machine m, we use the following constraint:

$$X_{im} + X_{jm} \leq 1 \, \forall m \in [1, M]$$

This constraint can easily be extended for multiple pairs of tasks that cannot be placed on the same machine.

Minimizing Cost

The cost to run machines in a cloud network comes from launching VMs, keeping the VMs up for a certain time period, and transferring data between VMs. Suppose launching VMs costs C_l dollars per VM, running VMs costs C_c dollars per second per CPU, and network transfers costs C_t dollars per byte. The total number of VMs launched is $n = \sum_{m=1}^{M} (\prod_{j=1}^{J} X_{jm})$. The total completion time for a particular placement is $t = \max_{m,n}(D_{mn}/R_{mn})$. The total amount of data transmitted between VMs is:

$$d = \sum_{i=1,j=1}^{J,J} \sum_{m=1}^{M} B_{ij} X_{im} (1 - X_{jm})$$

Here, $X_{im}(1 - X_{jm})$ ensures that we only consider the data transmitted between VMs. If two tasks are placed on the same VM, then there is only data exchange via disk I/O. The problem now is to solve:

$$\min(n \cdot C_l + t \cdot C_c + d \cdot C_t)$$

while satisfying the above CPU constraint and the constraint that each task must be placed on exactly one machine.

Bibliography

[1]

[2] 802.1Qpb - Equal Cost Multiple Paths. `http://www.ieee802.org/1/pages/802.1bp.html`.

[3] Ashraf Aboulnaga, Kenneth Salem, Ahmed A. Soror, Umar Farooq Minhas, Peter Kokosielis, and Sunil Kamath. Deploying Database Appliances in the Cloud. In *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, 2009.

[4] Abdelnaser Adas. Traffic Models in Broadband Networks. *IEEE Communications Magazine*, 35(7):82–89, 1997.

[5] Jung Ho Ahn, Nathan Binkert, Al Davis, Moray McLaren, and Robert S. Schreiber. HyperX: Topology, Routing, and Packaging of Efficient Large-scale Networks. In *SC*, 2009.

[6] Mohammad Al-Fares, Alexander Loukissas, and Amin Vahdat. A Scalable, Commodity Data Center Network Architecture. In *SIGCOMM*, 2008.

[7] Mohammad Al-Fares, Sivasankar Radhakrishnan, Barath Raghavan, Nelson Huang, and Amin Vahdat. Hedera: Dynamic Flow Scheduling for Data Center Networks. In *NSDI*, 2010.

[8] Mohammad Alizadeh, Albert Greenberg, David A. Maltz, Jitendra Padhye, Parveen Patel, Balaji Prabhakar, Sudipta Sengupta, and Murari Sridharan. Data Center TCP (DCTCP). In *SIGCOMM*, 2010.

[9] Mohammad Alizadeh, Abdul Kabbani, Tom Edsall, Balaji Prabhakar, Amin Vahdat, and Masato Yasuda. Less is More: Trading a Little Bandwidth for Ultra-Low Latency in the Data Center. In *NSDI*, 2012.

[10] Mohammad Alizadeh, Shuang Yang, Milad Sharif, Sachin Katti, Nick McKeown, Balaji Prabhakar, and Scott Shenker. pFabric: Minimal Near-Optimal Datacenter Transport. In *SIGCOMM*, 2013.

[11] Amazon EC2. http://aws.amazon.com/ec2/.

[12] Amazon EC2 Instances. https://aws.amazon.com/ec2/instance-types/.

[13] Amazon S3. http://aws.amazon.com/s3/.

[14] David Andersen, Hari Balakrishnan, Frans Kaashoek, and Robert Morris. Resilient Overlay Networks. In *SOSP*, 2001.

[15] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy Katz, Andy Konwinski, Gunho Lee, David Patterson, Ariel Rabkin, Ion Stoica, and Matei Zaharia. A View of Cloud Computing. *Communications of the ACM*, 53(4):50–58, 2010.

[16] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy H. Katz, Andrew Konwinski, Gunho Lee, David A. Patterson, Ariel Rabkin, Ion Stoica, and Matei Zaharia. Above the Clouds: A Berkeley View of Cloud Computing. Technical Report UCB/EECS-2009-28, University of California, Berkeley, 2009.

[17] AWS Case Study: NASA/JPL's Desert Research and Training Studies. http://aws.amazon.com/solutions/case-studies/nasa-jpl/.

[18] AWS Case Study: Netflix. http://aws.amazon.com/solutions/case-studies/netflix.

[19] Hitesh Ballani, Paolo Costa, Thomas Karagiannis, and Ant Rowstron. The Price is Right: Towards Location-Independent Costs in Datacenters. 2011.

[20] Hitesh Ballani, Paolo Costa, Thomas Karagiannis, and Ant Rowstron. Towards Predictable Datacenter Networks. In *SIGCOMM*, 2011.

[21] Theophilus Benson, Aditya Akella, and David A. Maltz. Network Traffic Characteristics of Data Centers in the Wild. In *IMC*, 2010.

[22] Theophilus Benson, Ashok Anand, Adidtya Akella, and Ming Zhang. Understanding Data Center Traffic Characteristics. In *WREN*, 2009.

[23] Bluelock. Choosing the Right Approach to Public Cloud Infrastructure as a Service, 2012.

[24] Peter Bodík, Ishai Menache, Mosharaf Chowdhury, Pradeepkumar Mani, David A. Maltz, and Ion Stoica. Surviving Failures in Bandwidth-Constrained Datacenters. In *SIGCOMM*, 2012.

[25] Jean-Chrysostome Bolot. Characterizing End-to-End Packet Delay and Loss in the Internet. *Journal of High Speed Networks*, 2(3):289–298, 1993.

[26] Brandon Butler. Microsoft Azure Overtakes Amazon's Cloud in Performance Test. http://www.networkworld.com/news/2013/021913-azure-aws-266831.html.

[27] Cisco Data Center Infrastructure 2.5 Design Guide. http://www.cisco.com/c/en/us/td/docs/solutions/Enterprise/Data_Center/DC_Infra2_5/DCI_SRND_2_5a_book.pdf.

[28] IBM ILOG CPLEX Optimizer. http://www-01.ibm.com/software/integration/optimization/cplex-optimizer/.

[29] Shuo Deng and Hari Balakrishnan. Traffic-aware Techniques to Reduce 3G/LTE Wireless Energy Consumption. In *CoNEXT*, 2012.

[30] N.G. Duffield, Pawan Goyal, Albert Greenberg, Partho Mishra, K.K. Ramakrishnan, and Jacobus E. van der Merwe. A Flexible Model for Resource Management in Virtual Private Networks. In *SIGCOMM*, 1999.

[31] David Erickson, Brandon Heller, Shuang Yang, Jonathan Chu, Jonathan Ellithorpe, Scott Whyte, Stephen Stuart, Nick McKeown, Guru Parulkar, and Mendel Rosenblum. Optimizing a Virtualized Data Center. In *SIGCOMM Demos*, 2011.

[32] Nathan Farrington and Alexey Andreyev. Facebook's Data Center Network Architecture. In *Optical Interconnects Conference*, 2013.

[33] Google Compute Engine. https://cloud.google.com/products/compute-engine/.

[34] Albert Greenberg, James R. Hamilton, Navendu Jain, Srikanth Kandula, Changhoon Kim, Parantap Lahiri, David A. Maltz, Parveen Patel, and Sudipta Sengupta. VL2: A Scalable and Flexible Data Center Network. In *SIGCOMM*, 2009.

[35] Chuanxiong Guo, Guohan Lu, Helen J. Wang, Shuang Yang, Chao Kong, Peng Sun, Wenfei Wu, and Yongguang Zhang. SecondNet: A Data Center Network Virtualization Architecture with Bandwidth Guarantees. In *CoNEXT*, 2010.

[36] Apache Hadoop. http://hadoop.apache.org.

[37] Mohammad Hajjat, Xin Sun, Yu-Wei Eric Sung, David Maltz, Sanjay Rao, Kunwadee Sripanidkulchai, and Mohit Tawarmalani. Cloudward Bound: Planning for Beneficial Migration of Enterprise Applications to the Cloud. In *SIGCOMM*, 2010.

[38] Nikhil Handigol, Mario Flagslik, Srini Seetharaman, Ramesh Johari, and Nick McKeown. Aster*x: Load-Balancing as a Network Primitive. In *ACLD (Poster)*, 2010.

[39] Qi He, Constantine Dovrolis, and Mostafa Ammar. On the Predictability of Large Transfer TCP Throughput. In *SIGCOMM*, 2005.

[40] Mark Herbster and Manfred K. Warmuth. Tracking the Best Expert. *Machine Learning*, 32:151–178, 1998.

[41] Christian E. Hopps. Analysis of an Equal-Cost Multi-Path Algorithm. RFC 2992, November 2000.

[42] HP Cloud Services. `http://hpcloud.com/`.

[43] Manish Jain and Constantinos Dovrolis. Ten Fallacies and Pitfalls on End-to-End Available Bandwidth Estimation. In *IMC*, 2004.

[44] Manish Jain and Constantinos Dovrolis. End-to-end Estimation of the Available Bandwidth Variation Range. In *Sigmetrics*, 2005.

[45] Manish Jain and Constantinos Dovrolis. End-to-end Available Bandwidth: Measurement Methodology, Dynamics, and Relation with TCP Throughput. In *SIGCOMM*, 2006.

[46] Raj Jain and Shawn A. Routhier. Packet Trains: Measurements and a New Model for Computer Network Traffic. *IEEE Journal on Selected Areas in Communications*, 4:986–995, 1986.

[47] Sugih Jamin, Scott J. Shenker, and Peter B. Danzig. Comparison of Measurement-based Admission Control Algorithms for Controlled-load Service. In *Infocom*, 1997.

[48] Vimalkumar Jeyakumar, Mohammad Alizadeh, David Mazières, Balaji Prabhakar, Changhoon Kim, and Albert Greenberg. EyeQ: Practical Network Performance Isolation at the Edge. In *NSDI*, 2013.

[49] Srikanth Kandula, Sudipta Sengupta, Albert Greenberg, Parveen Patel, and Ronnie Chaiken. The Nature of Datacenter Traffic: Measurements & Analysis. In *IMC*, 2009.

[50] Ethan Katz-Bassett, Harsha V. Madhyastha, Vijay Kumar Adhikari, Colin Scott, Justine Sherry, Peter van Wesep, Thomas Anderson, and Arvind Krishnamurthy. Reverse Traceroute. In *NSDI*, 2010.

[51] Gyuyeong Kim, Hoorin Park, Jieun Yu, and Wonjun Lee. Virtual Machines Placement for Network Isolation in Clouds. In *RACS*, 2012.

[52] Bennet Klein. Leveraging the Cloud for Data Protection and Disaster Recovery. Technical report, 2012.

[53] Edward W. Knightly and Ness B. Shroff. Admission Control for Statistical QoS: Theory and Practice. *IEEE Network*, 13(2):20–29, 1999.

[54] Terry Lam, Sivasankar Radhakrishnan, Amin Vahdat, and George Varghese. NetShare: Virtualizing Data Center Networks Across Services. Technical Report CSS2010-0957, University of California, San Diego, 2010.

[55] Vinh The Lam, Sivasankar Radhakrishnan, Rong Pan, Amin Vahdat, and George Varghese. NetShare and Stochastic NetShare: Predictable Bandwidth Allocation for Data Centers. *SIGCOMM CCR*, 42(3):5–11, 2012.

[56] Jeongkeun Lee, Myunjin Lee, Lucian Popa, Yoshio Turner, Sujata Banerjee, Puneet Sharma, and Bryan Stephenson. CloudMirror: Application-Aware Bandwidth Reservations in the Cloud. In *HotCloud*, 2013.

[57] Charles E. Leiserson. Fat-Trees: Universal Networks for Hardware-Efficient Supercomputing. *IEEE Transactions on Computers*, c-34(10), 1985.

[58] Ang Li, Xiaowei Yang, Srikanth Kandula, and Ming Zhang. CloudCmp: Comparing Public Cloud Providers. In *IMC*, 2010.

[59] Paresh Malalur. Traffic Delay Prediction from Historical Data. Master's thesis, Massachusetts Institute of Technology, 2010.

[60] Matthew Mathis, Jeffrey Semke, Jamshid Mahdavi, and Teunis Ott. The Macroscopic Behavior of the TCP Congestion Avoidance Algorithm. *SIGCOMM CCR*, 27(3), 1997.

[61] Xiaoqiao Meng, Vasileios Pappas, and Li Zhang. Improving the Scalability of Data Center Networks with Traffic-Aware Virtual Machine Placement. In *Infocom*, 2010.

[62] Jeffrey C. Mogul and Lucian Popa. What We Talk About When We Talk About Cloud Network Performance. *SIGCOMM CCR*, 42(5):44–48, 2012.

[63] Eslam Mohammadi, Mohammadbager Karimi, and Saeed Rasouli Heikalabad. A Novel Virtual Machine Placement in Cloud Computing. *Australian Journal of Basic and Applied Sciences*, 2011.

[64] Claire Monteleoni, Hari Balakrishnan, Nick Feamster, and Tommi Jaakkola. Managing the 802.11 Energy/Performance Tradeoff with Machine Learning. Technical Report MIT-LCS-TR-971, Massachusetts Institute of Technology, 2004.

[65] Radhika Niranjan Mysore, George Porter, and Amin Vahdat. FasTrak: Enabling Express Lanes in Multi-Tenant Data Centers. In *CoNEXT*, 2013.

[66] NetFlow. http://cisco.com/go/netflow.

[67] Network Performance Within Amazon EC2 and to Amazon S3. http://blog.rightscale.com/2007/10/28/network-performance-within-amazon-ec2-and-to-amazon-s3.

[68] Di Niu, Chen Feng, and Baochun Li. Pricing Cloud Bandwidth Reservations Under Demand Uncertainty. In *Sigmetrics*, 2012.

[69] OpenStack. http://openstack.org.

[70] OpenStack Operations Guide. http://docs.openstack.org/trunk/openstack-ops/content/projects_users.html.

[71] Balaji Palanisamy, Aameek Singh, Ling Liu, and Bushan Jain. Purlieus: Locality-aware Resource Allocation for MapReduce in a Cloud. In *Supercomputing*, 2011.

[72] Vern Paxson. End-to-End Internet Packet Dynamics. In *SIGCOMM*, 1997.

[73] Lucian Popa, Guatam Kumar, Mosharaf Chowdhury, Arvind Krishnamurthy, Sylvia Ratnasamy, and Ion Stoica. FairCloud: Sharing the Network in Cloud Computing. In *SIGCOMM*, 2012.

[74] Lucian Popa, Praveen Yalagandula, Sujata Banerjee, Jeffrey C. Mogul, Yoshio Turner, and Jose Renato Santos. ElasticSwitch: Practical Work-Conserving Bandwidth Guarantees for Cloud Computing. In *SIGCOMM*, 2013.

[75] R. S. Prasad, M. Murray, C. Dovrolis, and K. Claffy. Bandwidth Estimation: Metrics, Measurement Techniques, and Tools. *IEEE Network*, 17:27–35, 2003.

[76] Andres Quiroz, Hyunjoo Kim, Manish Parashar, Nathan Gnanasambandam, and Naveen Sharma. Towards Automatic Workload Provisioning for Enterprise Grids and Clouds. In *IEEE/ACM International Conference on Grid Computing*, 2009.

[77] Rackspace. http://rackspace.com/.

[78] Sivasankar Radhakrishnan, Yilong Geng, Vimalkumar Jeyakumar, Abdul Kabbani, George Porter, and Amin Vahdat. SENIC: Scalable NIC for End-Host Rate Limiting. In *NSDI*, 2014.

[79] Barath Raghavan, Kashi Vishwanath, Sriram Ramabhadran, Ken Yocum, and Alex C. Snoeren. Cloud Control with Distributed Rate Limiting. In *SIGCOMM*, 2007.

[80] Costin Raiciu, Mihail Ionescu, and Drago Niculescu. Opening Up Black Box Networks with CloudTalk. In *HotCloud*, 2012.

[81] Alexander Rasmussen, Michael Conley, Rishi Kapoor, Vinh The Lam, George Porter, and Amin Vahdat. Themis: An I/O-Efficient MapReduce. In *SOCC*, 2012.

[82] Alexander Rasmussen, George Porter, Michael Conley, Harsha V. Madhyastha, Radhika Niranjan Mysore, Alexander Pucher, and Amin Vahdat. TritonSort: A Balanced Large-Scale Sorting System. In *NSDI*, 2011.

[83] Thomas Ristenpart, Eran Tromer, Hovav Shacham, and Stefan Savage. Hey, You, Get Off of My Cloud: Exploring Information Leakage in Third-Party Compute Clouds. In *CCS*, 2009.

[84] Henrique Rodrigues, Jose Renato Santos, Yoshio Turner, Paolo Soares, and Dorgival Guedes. Gatekeeper: Supporting Bandwidth Guarantees for Multi-tenant Datacenter Networks. In *WIOV*, 2011.

[85] Matthew Roughan, Mikkel Thorup, and Yin Zhang. Traffic Engineering with Estimated Traffic Matrices. In *IMC*, 2003.

[86] Scaling Hadoop to 4000 Nodes at Yahoo! `http://developer.yahoo.net/blogs/hadoop/2008/09/scaling_hadoop_to_4000_nodes_a.html`.

[87] Jörg Schad, Jens Dittrich, and Jorge-Arnulfo Quiané-Ruiz. Runtime Measurements in the Cloud: Observing, Analyzing, and Reducing Variance. *Proceedings of the VLDB Endowment*, 3(1–2):460–471, 2010.

[88] sFlow. `http://sflow.org/about/index.php`.

[89] Rob Sherwood, Adam Bender, and Neil Spring. DisCarte: A Disjunctive Internet Cartographer. In *SIGCOMM*, 2008.

[90] Ankit Singla, Chi-Yao Hong, Lucian Popa, and P. Brighten Godfrey. Jellyfish: Networking Data Centers Randomly. In *NSDI*, 2012.

[91] Neil Spring, Ratul Mahajan, and David Wetherall. Measuring ISP Topologies with Rocketfuel. In *SIGCOMM*, 2002.

[92] William Voorsluys, James Broberg, Srikumar Venugopal, and Rajkumar Buyya. Cost of Virtual Machine Live Migration in Clouds: A Performance Evaluation. In *CloudCom*, 2009.

[93] Guohui Wang and TS Eugene Ng. The Impact of Virtualization on Network Performance of Amazon EC2 Data Center. In *Infocom*, 2010.

[94] Hao Wang, Haiyong Xie, Lili Qiu, Yang Richard Yang, Yin Zhang, and Albert Greenberg. COPE: Traffic Engineering in Dynamic Networks. In *SIGCOMM*, 2006.

[95] Windows Azure. `http://windowsazure.com`.

[96] Di Xie, Ning Ding, Y. Charlie Hu, and Ramana Kompella. The Only Constant is Change: Incorporating Time-Varying Network Reservations in Data Centers. In *SIGCOMM*, 2012.

[97] Yin Zhang, Matthew Roughan, Nick Duffield, and Albert Greenberg. Fast Accurate Computation of Large-Scale IP Traffic Matrices from Link Loads. In *Sigmetrics*, 2003.