Towards Systematic and Accurate Environment Selection for Emerging Cloud Applications

by

Ang Li

Department of Computer Science
Duke University

Date: ____________________

Approved:

_______________________
Xiaowei Yang, Supervisor

_______________________
Jeffrey S. Chase

_______________________
Bruce M. Maggs

_______________________
Ming Zhang

Dissertation submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy in the Department of Computer Science in the Graduate School of Duke University 2012
TOWARDS SYSTEMATIC AND ACCURATE ENVIRONMENT SELECTION FOR
EMERGING CLOUD APPLICATIONS

by

ANG LI

DEPARTMENT OF COMPUTER SCIENCE
DUKE UNIVERSITY

DATE: __________________________

APPROVED:

_______________________________

XIAOWEI YANG, SUPERVISOR

_______________________________

JEFFREY S. CHASE

_______________________________

BRUCE M. MAGGS

_______________________________

MING ZHANG

An abstract of a dissertation submitted in partial fulfillment of the requirements for
the degree of Doctor of Philosophy in the Department of Computer Science
in the Graduate School of Duke University
2012
Abstract

As cloud computing is gaining popularity, many application owners are migrating their applications into the cloud. However, because of the diversity of the cloud environments and the complexity of the modern applications, it is very challenging to find out which cloud environment is best fitted for one’s application.

In this dissertation, we design and build systems to help application owners select the most suitable cloud environments for their applications. The first part of this thesis focuses on how to compare the general fitness of the cloud environments. We present CloudCmp, a novel comparator of public cloud providers. CloudCmp measures the elastic computing, persistent storage, and networking services offered by a cloud along metrics that directly reflect their impact on the performance of customer applications. CloudCmp strives to ensure fairness, representativeness, and compliance of these measurements while limiting measurement cost. Applying CloudCmp to four cloud providers that together account for most of the cloud customers today, we find that their offered services vary widely in performance and costs, underscoring the need for thoughtful cloud environment selection. From case studies on three representative cloud applications, we show that CloudCmp can guide customers in selecting the best-performing provider for their applications.

The second part focuses on how to let customers compare cloud environments in the context of their own applications. We describe CloudProphet, a novel system that can accurately estimate an application’s performance inside a candidate cloud environment without the need of migration. CloudProphet generates highly portable shadow programs
to mimic the behavior of a real application, and deploys them inside the cloud to esti-
mate the application’s performance. We use the trace-and-replay technique to automatic-
cally generate high-fidelity shadows, and leverage the popular dispatcher-worker pattern
to accurately extract and enforce the inter-component dependencies. Our evaluation in
three popular cloud platforms shows that CloudProphet can help customers pick the best-
performing cloud environment, and can also accurately estimate the performance of a
variety of applications.
Contents

Abstract iv
List of Tables ix
List of Figures x
List of Abbreviations and Symbols xiii
Acknowledgements xiv

1 Introduction 1

1.1 Difficulties in Cloud Environment Selection 3

1.1.1 A Multitude of Cloud Services 3
1.1.2 Diverse Cloud Environments 4
1.1.3 Application Complexity 6

1.2 Thesis Overview 7

2 Background and Related Work 9

2.1 Performance and Cost in Public Clouds 9
2.2 Model-based Performance Estimation 10
2.3 Trace Replay-Based Performance Estimation 12
2.4 Other Performance Estimation Work 13

3 CloudCmp: A Systematic Comparator of Cloud Providers 14

3.1 Introduction 14
3.2 Goals and Approach 18
3.3 Measurement Methodology .............................................. 20
  3.3.1 Selecting Providers ............................................... 20
  3.3.2 Identifying Common Services .................................. 20
  3.3.3 Choosing Performance Metrics ................................. 22
3.4 Implementation ......................................................... 27
  3.4.1 Computation Metrics ............................................. 27
  3.4.2 Storage Metrics .................................................. 28
  3.4.3 Network Metrics .................................................. 29
3.5 Results ................................................................. 30
  3.5.1 Elastic Compute Cluster ......................................... 31
  3.5.2 Persistent Storage ............................................... 37
  3.5.3 Intra-cloud Network ............................................ 44
  3.5.4 Wide-area Network ............................................. 47
3.6 Using CloudCmp: Case Studies ...................................... 48
  3.6.1 E-commerce Website ............................................. 49
  3.6.2 Parallel Scientific Computation ............................... 50
  3.6.3 Latency Sensitive Website ..................................... 51
3.7 Discussion ............................................................ 52
3.8 Summary .............................................................. 53

4 CloudProphet: Estimating Application Behavior in the Cloud 54
  4.1 Introduction .......................................................... 54
  4.2 Overview ............................................................. 58
     4.2.1 Challenges ...................................................... 59
     4.2.2 What CloudProphet skips (for now) ......................... 62
  4.3 Design Details ....................................................... 63
List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>The services offered by the cloud providers we study. The intra-cloud networks of all four providers is proprietary, and are omitted from the table.</td>
<td>18</td>
</tr>
<tr>
<td>3.2</td>
<td>The operations we use to measure the performance of each storage service.</td>
<td>24</td>
</tr>
<tr>
<td>3.3</td>
<td>Information of the cloud instances we benchmark. With ( C_3 ), the first six CPU hours per day per application are free.</td>
<td>32</td>
</tr>
<tr>
<td>3.4</td>
<td>The average cost per operation for all three table services. The cost is in the unit of milli-cent, i.e., one-thousandth of a cent.</td>
<td>39</td>
</tr>
<tr>
<td>3.5</td>
<td>The maximum throughput an instance obtains from each blob service when downloading many 10MB blobs concurrently.</td>
<td>42</td>
</tr>
<tr>
<td>3.6</td>
<td>The geographical location of the cloud data centers.</td>
<td>45</td>
</tr>
<tr>
<td>4.1</td>
<td>Popular servers in different domains that follow the dispatcher-worker pattern.</td>
<td>60</td>
</tr>
<tr>
<td>4.2</td>
<td>The processor models we observed inside three cloud platforms (AWS, Rackspace, and Joyent). We found these models by launching 10 instances for each instance type offered by the clouds.</td>
<td>64</td>
</tr>
<tr>
<td>4.3</td>
<td>The applications we use to evaluate CloudProphet and the main resource types they consume.</td>
<td>76</td>
</tr>
<tr>
<td>4.4</td>
<td>The instance types used in our evaluation. We discovered two common types of CPUs used in the AWS instances, but only one CPU type for Joyent and Rackspace instances. The prices are quoted in March 2012.</td>
<td>76</td>
</tr>
<tr>
<td>4.5</td>
<td>Three ways to split the application over two data centers. The ratio corresponds to the portion of budget spent in each data center. The database component cannot be split, and always remains in Joyent’s cloud for policy reason. We use the small instance type in both data centers.</td>
<td>79</td>
</tr>
</tbody>
</table>
## List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>A typical cloud environment inside which an application runs.</td>
<td>4</td>
</tr>
<tr>
<td>3.1</td>
<td>The finishing time of benchmark tasks on various cloud instances. The time values are normalized using the longest finishing time to emphasize each provider’s relative performance. We show both single-threaded and multi-threaded results. The multi-threaded and I/O results are missing for $C_3$ because it does not support multi-threading or accessing the local disk.</td>
<td>31</td>
</tr>
<tr>
<td>3.2</td>
<td>The per-task monetary cost on each type of cloud instance.</td>
<td>33</td>
</tr>
<tr>
<td>3.3</td>
<td>The scaling latencies of the lowest end instance of each cloud provider.</td>
<td>35</td>
</tr>
<tr>
<td>3.4</td>
<td>The cumulative distribution of the response time when using the large table with 100K entries. Note that for the query operation, the x-axis is in a logarithmic scale, due to the significant performance gaps between different services.</td>
<td>36</td>
</tr>
<tr>
<td>3.5</td>
<td>The cumulative distribution of the time to consistency for $C_1$’s table service.</td>
<td>38</td>
</tr>
<tr>
<td>3.6</td>
<td>The cumulative distribution of the response time to download or upload a blob using Java-based clients.</td>
<td>39</td>
</tr>
<tr>
<td>3.7</td>
<td>The cumulative distribution of the time to download a 10MB blob using non-Java clients.</td>
<td>41</td>
</tr>
<tr>
<td>3.8</td>
<td>The blob downloading time from each blob service under multiple concurrent operations. The number of concurrent requests ranges from 1 to 32. Note that the x-axes are on a logarithmic scale.</td>
<td>41</td>
</tr>
<tr>
<td>3.9</td>
<td>The cumulative distributions of the response time to send or retrieve a message from a queue, and the propagation delay of a queue.</td>
<td>42</td>
</tr>
<tr>
<td>3.10</td>
<td>The intra-datacenter TCP throughput between two instances in all data centers we measure.</td>
<td>45</td>
</tr>
</tbody>
</table>
3.11 The TCP throughput between two different US data centers of a cloud provider. .......................... 46

3.12 This figure shows the cumulative distribution of the optimal round trip time (RTT) to the instances deployed on a cloud provider from 260 global vantage points. For $C_3$ we also show the actual RTT from a vantage point to the instance returned by the cloud’s DNS load balancing.  ............ 47

3.13 The page generation time of TPC-W when deployed on all three cloud providers that support it. The y-axis is in a logarithm scale. .......... 49

3.14 The job execution time of Blast when deployed on both $C_{1.1}$ and $C_{4.1}$. We show the results for five example jobs. ......................... 51

3.15 The distribution of the page downloading time of our website. We show the results for two different page sizes: 1KB and 100KB. ............... 52

4.1 The example of the inter-component dependencies between two threads. The work in the grey area cannot be executed (or replayed) in parallel due to the dependencies. ................................. 60

4.2 This figure shows how CloudProphet replays the behavior of each application component leveraging the dispatcher-worker pattern. CloudProphet uses its own dispatcher to mimic the real dispatcher’s behavior, and replays the events in each request’s handler. .......................... 61

4.3 The architecture of CloudProphet as applied to a three-tier web application. The key parts of CloudProphet are shown with shaded boxes: the tracer, an offline analyzer, and the replayers (shadows). Note that the database component is not replayed to accommodate managed database services in the cloud (§ 4.3.1). ................................. 63

4.4 Example of the linear relationship between the benchmark running times on two different processors (AMD Opteron 2374HE vs. Intel Xeon E5507). There is one point per benchmark. The dashed line shows the best linear fit. 64

4.5 The pseudo-code of CloudProphet’s dispatcher. ....................... 71

4.6 The real and estimated throughput of RUBiS and MediaWiki inside the candidate cloud environments. We also show the CPU benchmarking results for comparison (on y2-axis). ........................ 80

4.7 The real and estimated end-to-end response time for both applications. In comparison, we also show the time estimated with the modeling approach. 81

4.8 The real and estimated cost for RUBiS and MediaWiki. ................ 85
4.9 The real and estimated response times of the applications we evaluate. The
times are averaged over 10 runs and normalized to fit between 0 and 1. . . 87

4.10 The real and estimated response time distribution for the custom lock
benchmark with log4j. The lock blocking time of each request during
the replay run is also shown. To focus on lock contention, we run the
experiments on a 8-core instance type c1.xlarge offered by AWS. . . . . 88

4.11 Comparison between CloudProphet and the naive approach that replays
the traces thread-by-thread. We run the experiments under different work-
load levels. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 89

4.12 An example of the false dependency problem. During tracing, requests $A$
and $B$ arrive sequentially and are handled by the same thread. However,
during replay $B$ happens to arrive slightly earlier. With the naive approach,
the replay thread is blocked until $A$ finally arrives, whereas in the real
application both requests can be handled simultaneously by different threads. 90

4.13 Comparison between CloudProphet and the naive approach that treats vir-
tual cores as physical ones. We run the experiments under different work-
load levels. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 91

4.14 The average slowdown ratio of the applications when tracing is enabled. . 93

4.15 The average trace size generated per second by an actively running appli-
cation thread. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 94

4.16 The comparison of the migration overhead of the real applications and the
deployment overhead of CloudProphet. We use CMP points [99] as our
metric. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 94
# List of Abbreviations and Symbols

## Abbreviations

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DC</td>
<td>Data Center</td>
</tr>
<tr>
<td>CPU</td>
<td>Central Processing Unit</td>
</tr>
<tr>
<td>OS</td>
<td>Operating System</td>
</tr>
<tr>
<td>VM</td>
<td>Virtual Machine</td>
</tr>
<tr>
<td>DB</td>
<td>Database</td>
</tr>
<tr>
<td>IPC</td>
<td>Inter-Process Communication</td>
</tr>
<tr>
<td>TCP</td>
<td>Transport Control Protocol</td>
</tr>
<tr>
<td>RTT</td>
<td>Round-Trip-Time</td>
</tr>
<tr>
<td>AWS</td>
<td>Amazon Web Services</td>
</tr>
<tr>
<td>IaaS</td>
<td>Infrastructure-as-a-Service</td>
</tr>
<tr>
<td>PaaS</td>
<td>Platform-as-a-Service</td>
</tr>
</tbody>
</table>
Acknowledgements

First and foremost, I am grateful to my advisor Professor Xiaowei Yang. She has taught me a lot of things from how to think critically, how to find promising topics to work on, how to write clearly and accurately, and most importantly, how to do solid work. I consider myself very lucky to have the chance to work with her and learn from her.

I especially thank Srikanth Kandula and Ming Zhang, my collaborators at Microsoft Research. They have given me lots of excellent and wise advice not only on the work presented in this dissertation, but also on many general issues on both research and life. I also thank them for giving me the opportunity to intern at Microsoft Research, where I spent the most unforgettable summer during my graduate life.

I am fortunate to have collaborated with Pierre Francois and David Wetherall on the intra-domain routing projects. Both of them are extremely knowledgeable and helpful. I am also very lucky to have worked with Xin Liu, my labmate and roommate, from whom I can always draw great inspirations.

I thank Professor Jeffery Chase and Bruce Maggs for serving as my committee members. You have provided me with invaluable suggestions during my preliminary exam and the final defense.

This research has been supported by several NSF, USENIX, and ACM grants. I thank the department of Computer Science at the University of California, Irvine for the many learning opportunities they offered me. I also express my gratitude to the administrative personnels at UCI and Duke for providing me with all the resources I needed and for
ensuring the seamless transitions through the program transfer.

This Ph.D journey will be much less enjoyable without all the fun and joy brought by my friends. To Licong, Shengyue, Tong, Cheng, Yanbin, Dan, Yiming, thanks for all your help during my first few years in the United States. To Yuchen, our crazy discussions always revive me after a long day of work. Michael, Xin, Qiang, Xuanran, Yang, Lijun, Yu, Jun, Bingyang, Jannie, Eduardo, and so many others, thank you for bringing so much fun to my life at Duke.

Finally, I thank Miyuan for all your caring and love. Despite of being three thousand miles away, you are always close to my heart, and always stand by me no matter how difficult things might seem. Words cannot describe how much I appreciate you for sharing your life with me for four years, even though we have only been physically together for only a few months.

This work is dedicated to my parents, Hong Xiang and Yaping Li. Your trust, support, and love makes this happen.
Arguably, the era of utility computing has arrived. Computation resources no longer need to be owned before they can be used. Instead, customers can rent the resources as metered utilities, and use them only when demand comes. Under the new paradigm, a customer no longer needs to maintain her own dedicated infrastructure, which is both costly and requires special expertise. Further, the ability to acquire more or less resource in a short time frame allows customers to dynamically adjust their resource usage according to their applications’ demand, avoiding over- or under-provisioning. On the utility provider’s side, the economy of scale and statistical multiplexing amortizes the cost of their infrastructures over many customers, and makes offering utility computing services a profitable business.

As a modern incarnation of utility computing, public cloud computing has grown tremendously in the recent years. Plenty of companies are providing public cloud computing services over the Internet, such as Amazon, Google, Microsoft, Rackspace, and GoGrid. On the other hand, attracted by the benefits of utility computing, many customers are migrating their applications into the clouds. For instance, Netflix, the lead online streaming service provider, has moved its entire software stack, from streaming servers to the recommendation system, to Amazon’s cloud [36]. GitHub, the popular source code
management and sharing application, has been migrated to Rackspace’s cloud [18]. In a recent Symantec survey of 1,780 data center managers in 26 countries [1], over 72% of respondents indicated they were considering or using public cloud computing in order to reduce their data center costs. Analysts further predict that in 2012 over 80% of new commercial enterprise applications will be deployed in the cloud [73].

Although migrating applications to the cloud has become a popular trend, it is difficult to do it right. A chief problem is how to decide the most suitable cloud environment for an application. The cloud environment of an application includes the physical infrastructure the application runs on and all the cloud services it uses. They are crucial to how the application behaves inside the cloud. For instance, an environment with poor hardware or high interference from other applications can significantly affect an application’s performance. Further, an environment that scales slowly when resource demand increases can limit how fast an application can react to workload changes. Finally, as many cloud services are not offered free-of-charge, their prices and billing models can also impact how much an application would cost.

The goal of this dissertation is to make cloud migration easier by helping customers find the most suitable cloud environments. We first aim to systematically compare the performance and cost characteristics of today’s public cloud providers. Such comparison is necessary for two main reasons. First, it helps us better understand the problem. For instance, if the various cloud environments are essentially similar, it would be a moot point to choose between them; if they are drastically different, e.g., one cloud's virtual machine is 5X faster or 2X cheaper than that of another cloud, choosing the right cloud environment becomes crucial. Secondly, cloud customers and providers may directly benefit from the comparison results. Customers may choose the highest ranked cloud providers, while providers can know what aspects of their platforms need improvement.

Although the comparison allows customers to evaluate the general fitness of the cloud environments, they are insufficient to compare them in the context of a specific application.
This is because each application has its unique characteristics in many aspects, such as what services to use, how much resource is consumed, how its components depend on each other, and etc. It is difficult to accurately predict the performance of a specific application from the application-independent comparison results.

To address this, in the second part of this dissertation, we aim to estimate an application’s behavior (e.g., end-to-end performance, scalability, and cost) inside one or multiple candidate cloud environments. What’s more, we wish to achieve this goal without requiring the customer to port her application to each of the candidate environments, because application porting can be complex and time-consuming. We discovered that it is possible to generate simple and highly portable shadows programs that mimic the behavior of the real application components. Using the shadows, we are able to accurately estimate an application’s behavior without going through the trouble to port it.

In the following sections, we first explain why it is difficult to find the most suitable cloud environment for an application. We then present an overview of this dissertation.

1.1 Difficulties in Cloud Environment Selection

Compared to the traditional problem of choosing what type of physical machine or storage sub-system to be used by a legacy application, the problem of selecting a cloud environment is more general in scope and harder to deal with. We have identified three key challenges in this context: a) a cloud environment involves many different services, b) the available cloud environments are highly diverse and incomparable, and c) modern cloud applications are complex. We discuss each of them in more detail.

1.1.1 A Multitude of Cloud Services

Figure 1.1 shows a typical cloud environment an application runs in. It includes a cluster of virtual machines (VMs) to run the application’s code, one or more storage services to host the application’s data, an intra-cloud network that connects the virtual machines
with themselves and with the storage services, and the wide-area network that deliver the application’s content to its end-users. These services work together to offer a coherent running environment for the application, and therefore the performance and cost of each service would affect the overall performance and cost. When a customer is selecting a cloud environment, she must take into account all the services required by her application. This is more involved than only choosing the fastest machine or the highest throughput network.

1.1.2 Diverse Cloud Environments

As a growing number of companies are offering cloud computing services, a customer has many candidate cloud environments to choose from. These environments, although similar in their offered services, differ dramatically in many aspects and are not directly comparable. Below we show a subset of the differences and explain why they matter.

First of all, the cloud providers may use different physical machine types and have different internal network architectures. From the study of five popular cloud providers, we did not find any processor model used by more than one cloud. In addition, each
cloud provider has its own proprietary way to build its intra-datacenter network, following the numerous proposals on datacenter network architectures [68, 85, 69, 77, 102]. Such differences can affect the computation and network performance of the applications.

The services offered inside the environments can also be implemented differently. For instance, the computation services can be built upon different virtualization technologies [57, 28, 52, 49]. They can also be offered with different models and abstractions. For instance, some offer the so-called Infrastructure-as-a-Service (IaaS), where a customer runs applications inside bare-metal virtual machines (VMs), using the APIs provided by their chosen guest operating systems. Others offer Platform-as-a-Service (PaaS), where a customer can only build applications using the APIs provided by the cloud. It is unclear how one can compare across the different service models.

Because a cloud infrastructure is shared by many applications, the level of performance isolation can also differ across cloud environments. Some clouds offer better isolation through techniques such as CPU capping, cache partitioning [91], and explicit network bandwidth allocation [92]. The others offer less isolation and applications might suffer from interference.Existing work has shown that the interference on resources such as CPU, cache, I/O, and network bandwidth can adversely affect an application’s performance [103, 84, 90, 92].

Lastly, we discovered that different cloud environments can also have different charging models. For instance, Amazon’s AWS charges the computation resource by the number and duration of virtual machines used by a customer, while Google’s AppEngine charges by the number of CPU cycles consumed by a customer’s application. As another example, Microsoft’s Azure charges the storage services only based on how many accesses are made to those services, while AWS and AppEngine charge by the amount of CPU resources consumed by the services to serve the storage requests. The heterogeneous charging models make it difficult to compare the cost-effectiveness across different cloud environments.
1.1.3 Application Complexity

Even if we have managed to make the diverse cloud environments comparable, there is still one remaining question: how to select the most suited cloud environment for a particular application? In many cases, a customer already has an application to migrate, and she only cares about which cloud environment offers the best performance or lowest cost for her application.

This question is difficult to answer because the applications are complex. A modern cloud application can include many distributed components such as front end load balancers, web servers, business servers, cache servers, and databases. Each of these components depends on many third-party software, and can use multiple cloud services simultaneously. In addition, complex dependencies can exist between the components, which further complicates the application’s overall behavior.

The complexity poses two difficulties. First, it makes it very time-consuming or even impractical to migrate an application to multiple candidate environments and compare their performance. For instance, Tran et al. empirically studied the migration process of 6 multi-tiered web applications [99], and found that each migration process can take up to 50 human-hours, due to the numerous work items to move all the binaries, data, and configurations to the cloud. The complexity also increases the likelihood of errors during migration. Zhang et al. analyzed the errors encountered while migrating several real applications to Amazon AWS [108], and found that configuration error is one of the main culprits. Although cloud providers usually support VM images for fast startup, these images are often not transferable across providers, which means a customer must set up her application from scratch in every new provider.

Second, the complexity also makes it difficult to model an application’s behavior in a new environment. The existing application performance models such as [96, 95, 94, 100] require extensive expert effort to build and calibrate, but each of them only captures one or
a few aspects of the application or the environment. A holistic model that can accurately predict an application’s behavior by capturing all dimensions of a complex networked application running in a new cloud environment is still missing, probably because of the sheer complexity to describe everything analytically.

1.2 Thesis Overview

The rest of this thesis is structured as follows. In Chapter 2, we describe the related work in this area and provide a background for our work. We first discuss the recent studies on cloud performance and cost. We then describe work related to estimating an application’s performance in new environments.

Chapter 3 presents the design and implementation of CloudCmp, a systematic comparator for public cloud providers. We first describe the design of the toolset, including what services to compare, what metrics to use, and how we normalize the differences between services to make them comparable. We then describe the results from a two-month measurement study using CloudCmp over four popular cloud providers in today’s market. Finally, we discuss our experience in using the comparison results to predict the best-performing provider for three representative applications.

Chapter 4 presents CloudProphet, a system that uses portable shadows to estimate an application’s behavior in a new environment without migration. CloudProphet uses the trace-and-replay technique to build application shadows with low overhead. The system further leverages common programming patterns to extract and enforce the inter-component dependencies. We implemented a Linux prototype of CloudProphet and evaluated its effectiveness in choosing the best cloud environment for a variety of applications.

Chapter 5 describes our vision for future research work in this area. We are going to continue our research in several directions. First, we plan to use the CloudCmp tool to launch a temporal comparison of the cloud providers to see how they evolve over time.
Second, we plan to extend the CloudProphet system to cover more applications types. Third, we plan to build an efficient search framework upon CloudProphet to let customers quickly locate the best environment from a large candidate set.

Chapter 6 concludes with a summary of the dissertation. We have made two main contributions in this work. First, we built tools that enable the comparison of public cloud providers. Our measurement results have also motivated much following work in this area (e.g., [92, 56, 81]). Second, we demonstrated that it is possible to accurately estimate a complex application’s behavior in a new environment without migration.
In this chapter, we provide a background for our work and give an overview of the related work in this area. There has been a considerable amount of research efforts on studying the performance and cost inside cloud platforms and helping customers make decisions. Beyond the cloud scope, there is also much existing work that aims at estimating an application’s behavior in other contexts. We will focus on the most relevant and discuss their difference from our approaches. We first describe the recent studies on the performance and cost in today’s public cloud platforms. We then discuss the research efforts that try to estimate an application’s performance in new environments. Based on the different approaches they use, we classify them into model-based, trace replay-based, and others.

2.1 Performance and Cost in Public Clouds

As the public clouds are being adopted by more and more mission-critical business and enterprise applications, there is a growing attention on the performance and cost characteristics inside those cloud platforms. Wang and Ng studied how virtualization affects the network performance in Amazon’s EC2 compute cluster [103]. Their finding suggests
that CPU core sharing in some of the low-end instance types can lead to dramatic instabilities in network throughput and latency, even when the data center network is lightly loaded. Walker investigates the performance of scientific applications on Amazon EC2 using both macro and micro benchmarks [101]. Tak [98] and Chen [59] use analytical cost models to analyze the cost in real public clouds (Amazon AWS and Microsoft Azure) and try to find out when it is economical to migrate an application to those clouds. Some work [62, 109, 70] studies how to exploit the spot instance market in Amazon EC2 [3] to attain high application performance at low cost.

The industry has also put much focus on investigating the performance of the current public cloud providers. Numerous industrial reports and blog posts present performance benchmarking and comparison results along a few dimensions and across a couple of providers. More recently, a commercial service called CloudHarmony [14] has emerged to offer continuous benchmarking results for the compute clusters inside a number of IaaS cloud providers.

Our work differs from the existing work in its goal: we aim to compare the different cloud providers. This new goal allows us to downplay some aspects of our study, such as the thoroughness in investigating each individual cloud service, but it also introduces new challenges. For instance, we need to normalize the differences between the cloud providers and make them comparable. Moreover, we need to cover a comprehensive set of cloud services, so that the comparison results would be helpful for general applications. We describe how we address those challenges in Chapter 3.

2.2 Model-based Performance Estimation

We now turn to the research efforts that estimate an application’s performance in a new environment. One area of work is to use abstract models to capture the characteristics of the applications and the environments. A variety of models have been proposed where
each captures a different set of characteristics and is suitable for a specific context. In the following we discuss some important and representative work in this area.

Stewart and Shen propose a queue-based model for multi-component online service applications [96]. The model captures the resource usage of each application component, the network delay between them, and the extra communication resource consumption when two components are deployed on separate machines. Urgaonkar et al. propose a similar model for multi-tiered web applications [100] but choose a closed queueing network that better describes session-based applications. The models can be used to predict a multi-tiered application’s performance (throughput and response time) after varying how the components are placed on a set of physical machines. However, because the models do not capture how the applications interact with the underlying machines, it cannot be used when an application is migrated to a completely new platform with unprofiled machines, such as a cloud platform.

Some models are built to carry out cross-platform performance prediction. Ipek et al. use Artificial Neural Networks (ANNs) to predict how an application would behave under a specific set of CPU parameters, such as frequency, cache size, branch prediction algorithm, etc [74]. Ben et al. adopt a similar approach but use regression models rather than ANNs [80]. Those models only work for simple single-threaded applications, because they do not capture the interaction between different application threads and components.

With the rise of virtualization and shared infrastructure, a number of models have been proposed to specifically predict an application’s performance or resource usage when virtualized. The early work of Doyle et al. studies internal models to estimate the service response time of a static-object hosting web service under different loads and resource allocations [64]. Kundu et al. use ANNs to predict the application’s performance under different Xen parameter settings [79]. Wood et al. aim at a slightly different goal of estimating the virtualization overhead of an application using a regression model [106].
2.3 Trace Replay-Based Performance Estimation

Trace replay is a general technique that records some important application events and replays them later to mimic the original application’s behavior. It has been adopted in many contexts, such as debugging applications [93, 71] and detecting race conditions in multi-core servers [88].

Trace replay can also be used to estimate an application’s performance. traceFS [54] and replayFS [76] are a pair of systems that trace and replay I/O activities at the Virtual File System (VFS) level. The design goal of the systems is to re-execute the exact same VFS calls with the exact same time intervals to faithfully replay the I/O activities. TBBT [111] is a similar system for replaying Network File System (NFS) traces. //Trace [83] targets at parallel applications with deterministic data dependencies between the threads. It discovers the dependencies by throttling each thread in turn, and enforces the same dependencies during replay. Monkey [60] is a tool that can replay the TCP connections between client and server to evaluate the network infrastructure and the TCP stack implementation. Tcpreplay [45] can faithfully replay a packet trace to test network equipments such as firewalls, Intrusion Detection Systems (IDS), etc.

Our work is also motivated by the trace replay technique, but the existing work cannot be directly applied in our context. Because our goal is to estimate the end-to-end behavior of an entire application, we need a much broader coverage on the various resources an application might use, while the existing work mostly focuses on one resource (e.g., disk I/O or network). Moreover, the existing work assumes no or highly simplified dependencies between different application components. This might significantly limit their applicability to general cloud applications.
2.4 Other Performance Estimation Work

Hoste et al. propose to use architecture-independent characteristics to find the most similar benchmarks to predict the running time of CPU-intensive applications across a large collection of CPU types [72]. Such approach would only work when there already exists a simple benchmark similar to the target application. However, it is difficult to find pre-existing benchmarks that are similar to the large variety of applications one might deploy on today’s cloud platforms.

JustRunIt [110] is a novel system to test the performance of an application component in a new virtualization setting. It first clones an existing VM that hosts the application component, and benchmarks the new VM by replaying real application requests. Such mechanism can achieve high accuracy with little overhead, but it only works if the application is already hosted in a virtualization environment. This is not necessarily true for many legacy applications which are still hosted on bare-metal machines.
3

CloudCmp: A Systematic Comparator of Cloud Providers

3.1 Introduction

As Internet-based cloud computing are gaining momentum. A growing number of companies are riding this wave to provide public cloud computing services, such as Amazon, Google, Microsoft, Rackspace, and GoGrid. These cloud providers offer a variety of options in pricing, performance, and feature set. For instance, some offer platform as a service (PaaS), where a cloud customer builds applications using the APIs provided by the cloud; others offer infrastructure as a service (IaaS), where a customer runs applications inside virtual machines (VMs), using the APIs provided by their chosen guest operating systems. Cloud providers also differ in pricing models. For example, Amazon’s AWS charges by the number and duration of VM instances used by a customer, while Google’s AppEngine charges by the number of CPU cycles consumed by a customer’s application.

The diversity of cloud providers leads to a practical question: how well does a cloud provider perform compared to the other providers? Answering this question will benefit both cloud customers and providers. For a potential customer, the answer can help
it choose a provider that best fits its performance and cost needs. For instance, it may choose one provider for storage intensive applications and another for computation intensive applications. For a cloud provider, such answers can point it in the right direction for improvements. For instance, a provider should pour more resources into optimizing table storage if the performance of its store lags behind competitors.

Despite the practical relevance of comparing cloud providers, there have been few studies on this topic. The challenge is that every provider has its own idiosyncratic ways of doing things, so finding a common ground needs some thought. A few efforts have characterized the performance of one IaaS provider (Amazon AWS) [66, 101]. Some recent blog posts [15, 39, 104] compare Amazon AWS with one other provider each. These measurements are limited in scope; none of them cover enough of the dimensions (e.g., compute, storage, network, scaling) to yield meaningful conclusions. Further, some of the measurement methodologies do not extend to all providers, e.g., they would not work for PaaS providers.

In this chapter, we consider the problem of systematically comparing the performance of cloud providers. We identify the key requirements for conducting a meaningful comparison (§3.2), develop a tool called CloudCmp, and use CloudCmp to evaluate a few cloud providers (§3.3–§3.4) that differ widely in implementation but together dominate the cloud market. Our results (§3.5) provide a customer with the performance-cost trade-offs across providers for a wide set of metrics. For providers, the results point out specific areas for improvement in their current infrastructures.

Several technical challenges arise in realizing a comparator for cloud providers. The first is the choice of what to measure. Rather than focusing on the nitty-gritty such as which virtualization technology a provider uses or how it implements its persistent storage, we take an end-to-end approach that focuses on the dimensions of performance that customers perceive. Doing so has the advantage that the measurement methodology remains stable even as the implementations change over time or differ widely across providers. To this
end, we identify a common set of services offered by these providers, including elastic computing, persistent storage, and intra-cloud and wide-area networking (§3.3.2).

The second challenge is the choice of how to measure customer perceived performance of these services. For each service, we focus on a few important metrics, e.g., speed of CPU, memory, and disk I/O, scaling latency, storage service response time, time to reach consistency, network latency, and available bandwidth (§3.3.3). We leverage pre-existing tools specific to each metric. However, when applying the tools, we had to be careful along several axes, such as using variable number of threads to test multi-core, piecing apart the interference from colocated tenants and the infrastructure itself, and covering the wide set of geographically distributed data centers offered by the providers. The individual tools per se are simple, but this specific collection of them that enables comparing cloud providers is novel.

Third, as with all significant measurement efforts, we trade off development cost to the completeness of the study. We skip functionality that is specific to small classes of applications but are comprehensive enough that our benchmark results allow predicting the performance of three representative applications: a storage intensive e-commerce web service, a computation intensive scientific computing application, and a latency sensitive website serving static objects. By deploying these applications on each cloud, we demonstrate that the predictions from CloudCmp align well with the actual application performance. CloudCmp enables predicting application performance without having to first port the application onto every cloud provider.

Finally, unlike other measurement efforts, we are constrained by the monetary cost of measuring the clouds and the acceptable use policies of the providers. We note that the results here were achieved under a modest budget by judicious choice of how many and how often to measure. CloudCmp complies with all acceptable use policies.

We used CloudCmp to perform a comprehensive measurement study over four major cloud providers, namely, Amazon AWS, Microsoft Azure, Google AppEngine, and
Rackspace CloudServers. We emphasize that the infrastructure being measured is ephemeral. Providers periodically upgrade or regress in their software or hardware and customer demands vary over time. Hence, these results are relevant only for the time period in which they were generated. To keep the focus on the value of this comparison method and its implications rather than rank providers, our results use labels $C_1 - C_4$ instead of provider names.

From the comparison results, we find that the performance and price of the four providers vary significantly with no one provider standing out (§3.5). For instance, while the cloud provider $C_1$ has the highest intra-cloud bandwidth, its virtual instance is not the most cost-effective. The cloud provider $C_2$ has the most powerful virtual instances, but its network bandwidth is quite limited. $C_3$ offers the lowest wide-area network latency, but its storage service is slower than that of its competitors. We highlight a few interesting findings below:

- Cloud instances are not equally cost-effective. For example, while only 30% more expensive, $C_4$’s virtual instance can be twice as fast as that of $C_1$.

- $C_2$ in our study allows a virtual instance to fully utilize the underlying physical machine when there is no local resource competition. Hence, an instance can attain high performance at low cost.

- The performance of the storage service can vary significantly across providers. For instance, $C_1$’s table query operation is an order of magnitude faster than that of the others.

- The providers offer dramatically different intra-datacenter bandwidth, even though intra-datacenter traffic is free of charge. For instance, $C_1$’s bandwidth is on average three times higher than $C_2$’s.
Table 3.1: The services offered by the cloud providers we study. The intra-cloud networks of all four providers is proprietary, and are omitted from the table.

<table>
<thead>
<tr>
<th>Provider</th>
<th>Elastic Cluster</th>
<th>Storage</th>
<th>Wide-area Network</th>
</tr>
</thead>
<tbody>
<tr>
<td>Amazon AWS</td>
<td>Xen VM</td>
<td>SimpleDB (table)</td>
<td>3 DC locations</td>
</tr>
<tr>
<td></td>
<td></td>
<td>S3 (blob), SQS (queue)</td>
<td>(2 in US, 1 in EU)</td>
</tr>
<tr>
<td>Microsoft Azure</td>
<td>Azure VM</td>
<td>XStore (table, blob, queue)</td>
<td>6 DC locations</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(2 each in US, EU, and Asia)</td>
<td></td>
</tr>
<tr>
<td>Google AppEngine</td>
<td>Proprietary</td>
<td>DataStore (table)</td>
<td>Unpublished number of Google DCs</td>
</tr>
<tr>
<td>Rackspace CloudServers</td>
<td>Xen VM</td>
<td>CloudFiles (blob)</td>
<td>2 DC locations</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(all in US)</td>
<td></td>
</tr>
</tbody>
</table>

We believe that this is the first study to comprehensively characterize the performance and cost of the major cloud providers in today’s market. Though we present results for four providers in this paper, we believe the techniques in CloudCmp can be extended to measure other providers. We have released the measurement data and the CloudCmp toolset on our website: http://www.cloudcmp.net.

3.2 Goals and Approach

In this section, we highlight the design goals of CloudCmp and briefly describe how we meet them.

1. **Guide a customer’s choice of provider:** Our primary goal is to provide performance and cost information about various cloud providers to a customer. The customer can use this information to select the right provider for its applications. We choose the cost and performance metrics that are relevant to the typical cloud applications a customer deploys. These metrics cover the main cloud services, including elastic computing, persistent storage, and intra-cloud and wide-area networking.

2. **Relevant to cloud providers:** We aim to help a provider identify its under-performing services compared to its competitors. We not only present a comprehensive set of
measurement results, but also attempt to explain what causes the performance differences between providers. This enables a provider to make targeted improvements to its services.

3. **Fair:** We strive to provide a fair comparison among various providers by characterizing all providers using the same set of workloads and metrics. This restricts our comparative study to the set of common services offered by all providers. The core functionality we study suffices to support a wide set of cloud applications. However, we skip specialized services specific to some applications that only a few providers offer. While support for functionality is a key decision factor that we will consider in future work, our focus here is on the performance-cost trade-off.

4. **Thoroughness vs. measurement cost:** For a thorough comparison of various cloud providers, we should measure all cloud providers continuously across all their data centers. This, however, incurs significant measurement overhead and monetary costs. In practice, we periodically (e.g., once an hour) measure each provider at different times of day across all its locations. The measurements on different providers are loosely synchronized (e.g., within the same hour), because the same measurement can take different amount of time to complete in different providers.

5. **Coverage vs. development cost:** Ideally, we would like to measure and compare all cloud providers on the market. Achieving this goal, however, can be cost and time prohibitive. We cover a representative set of cloud providers while restricting our development cost. We choose the cloud providers to compare based on two criteria: popularity and representativeness. That is, we pick the providers that have the largest number of customers and at the same time represent different models such as IaaS and PaaS. Our measurement methodology, however, is easily extensible to other providers.
6. **Compliant with acceptable use policies:** Finally, we aim to comply with cloud providers’ use policies. We conduct experiments that resemble the workloads of legitimate customer applications. We do not overload the cloud infrastructures or disrupt other customer applications.

### 3.3 Measurement Methodology

In this section, we describe how we design CloudCmp to conduct a fair and application-relevant comparison among cloud providers. We first show how we select the providers to compare, and discuss how to choose the common services to ensure a fair comparison. Then for each type of service, we identify a set of performance metrics that are relevant to application performance and cost.

#### 3.3.1 Selecting Providers

Our comparative study includes four popular and representative cloud providers: Amazon AWS [9], Microsoft Azure [35], Google AppEngine [20], and Rackspace Cloud-Servers [38]. We choose Amazon AWS and Rackspace Cloud-Servers because they are the top two providers that host the largest number of web services [44]. We choose Google AppEngine because it is a unique PaaS provider, and choose Microsoft Azure because it is a new entrant to the cloud computing market that offers the full spectrum of computation and storage services similar to AWS.

#### 3.3.2 Identifying Common Services

Despite the complexity and idiosyncrasies of the various cloud providers, there is a common core set of functionality. In this section, we focus on identifying this common set, and describe the experience of a customer who uses each functionality. We defer commenting on the specifics of how the cloud achieves each functionality unless it is relevant.
This allows us to compare the clouds from the end-to-end perspective of the customer and sheds light on the meaningful differences. The common set of functionality includes:

- **Elastic compute cluster.** The cluster includes a variable number of virtual instances that run application code.

- **Persistent storage.** The storage service k.pdf the state and data of an application and can be accessed by application instances through API calls.

- **Intra-cloud network.** The intra-cloud network connects application instances with each other and with shared services.

- **Wide-area network.** The content of an application is delivered to end users through the wide-area network from multiple data centers (DCs) at different geographical locations.

These services are offered by most cloud providers today because they are needed to support a broad spectrum of applications. For example, a web application can have its servers run in the elastic compute cluster, its data stored in the persistent storage, and its content delivered through the wide-area network. Other cloud applications such as document translation, file backup, and parallel computation impose different requirements on these same components.

A few providers offer specialized services for specific applications. For instance, Amazon also offers a MapReduce service [4] for big data processing applications and an in-memory caching service [5] for database-intensive applications. We skip evaluating these offerings to focus on the more general applications. Table 3.1 summarizes the services offered by the providers we study.
3.3.3 Choosing Performance Metrics

For each of these cloud services, we begin with some background and describe the performance and cost metrics we use to characterize that service.

Elastic Compute Cluster

A compute cluster provides virtual instances that host and run a customer’s application code. Across providers, the virtual instances differ in their underlying server hardware, virtualization technology, and hosting environment. Even within a provider, multiple tiers of virtual instances are available, each with a different configuration. For example, the instances in the higher tier can have faster CPUs, more CPU cores, and faster disk I/O access. These differences do impact the performance of customer applications.

The compute cluster is charged per usage. There are two types of charging models among the providers we study. The IaaS providers (AWS, Azure, and CloudServers) charge based on how long an instance remains allocated, regardless of whether the instance is fully utilized or not. However, the PaaS provider (AppEngine) charges based on how many CPU cycles a customer’s application consumes in excess of a few free CPU hours per application per day.

The compute cluster is also “elastic” in the sense that a customer can dynamically scale up and down the number of instances it uses to withstand its application’s varying workload. Presently, there are two types of scaling mechanisms: opaque scaling and transparent scaling. The former requires a customer herself to manually change the number of instances or specify a scaling policy, such as creating a new instance when average CPU usage exceeds 60%. The latter automatically tunes the number of instances without customer intervention. AWS, Azure, and CloudServers support opaque scaling whereas AppEngine provides transparent scaling.

We use three metrics to compare the performance of the compute clusters: benchmark
finishing time, cost per benchmark, and scaling latency. These metrics reflect how fast an instance can run, how cost-effective it is, and how quickly it can scale.

**Benchmark finishing time.** Similar to conventional computational benchmark metrics for computer architectures [43], this metric measures how long the instance takes to complete the benchmark tasks. The benchmark has tasks that stress each of the main compute resources (CPU, memory, and disk I/O).

**Cost.** This is the monetary cost to complete each benchmark task. Because we use the same tasks across different instances provided by different clouds, customers can use this metric to compare the cost-effectiveness of the instances regardless of their prices and charging models. Together with the above metric, this provides customers with a view of the performance-cost trade-offs across providers. These metrics correspond to the criteria that customers use when choosing, such as best performance within a cost budget or lowest cost above a performance threshold.

**Scaling latency.** This is the time taken by a provider to allocate a new instance after a customer requests it. The scaling latency of a cluster can affect the performance and cost of running an application. An application can absorb workload spikes more quickly and can keep fewer number of instances running continuously if it can instantiate new instances quickly. With this metric, a customer can choose the compute cluster that scales the fastest or design better scaling strategies. She can also make more nuanced decisions based on what it would cost to provide good performance when the workload of her application varies.

There are a few other metrics, such as the customizability of a virtual instance and the degree of automation in management, that capture vital aspects of cloud providers. However, these are harder to quantify. Hence, we focus on the performance and costs of running an application and defer considering other metrics to future work.
Table 3.2: The operations we use to measure the performance of each storage service.

<table>
<thead>
<tr>
<th>Service</th>
<th>Operation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Table</td>
<td>get</td>
<td>fetch a single row using the primary key</td>
</tr>
<tr>
<td></td>
<td>put</td>
<td>insert a single row</td>
</tr>
<tr>
<td></td>
<td>query</td>
<td>lookup rows that satisfy a condition on a</td>
</tr>
<tr>
<td></td>
<td></td>
<td>non-primary key field</td>
</tr>
<tr>
<td>Blob</td>
<td>download</td>
<td>download a single blob</td>
</tr>
<tr>
<td></td>
<td>upload</td>
<td>upload a single blob</td>
</tr>
<tr>
<td>Queue</td>
<td>send</td>
<td>send a message to a queue</td>
</tr>
<tr>
<td></td>
<td>receive</td>
<td>retrieve the next message from a queue</td>
</tr>
</tbody>
</table>

Persistent Storage

Cloud providers offer persistent storage for application state and data. There are currently three common types of storage services: table, blob, and queue. The table storage is designed to store structural data in lieu of a conventional database, but with limited support for complex queries (e.g., table join and group by). The blob storage is designed to store unstructured blobs, such as binary objects, user generated data, and application inputs and outputs. Finally, the queue storage implements a global message queue to pass messages between different instances. Most storage services are implemented over HTTP tunnels, and while not standardized, the usage interfaces are stable and similar across providers.

The cloud storage services have two advantages over their conventional counterparts: scalability and availability. The services are well-provisioned to handle load surges and the data is replicated [8] for high availability and robustness to failures. However, as a trade-off, cloud storage services do not offer strong consistency guarantees [67]. Therefore, an application can retrieve stale and inconsistent data when a read immediately follows a write.

There are presently two pricing models for storage operations. The table services of AWS and AppEngine charge based on the CPU cycles consumed to run an operation. Thus, a complex query costs more than a simple one. Azure and CloudServers have a
fixed per-operation cost regardless of the operation’s complexity.

We use three metrics to compare the performance and cost of storage services: operation response time, time to consistency, and cost per operation.

**Operation response time.** This metric measures how long it takes for a storage operation to finish. We measure operations that are commonly supported by providers and are popular with customers. Table 3.2 summarizes these operations. They include the basic read and write operations for each storage service. For table storage service, we also use an SQL-style query to test the performance of table lookup. In §3.6.1, we show that these operations account for over 90% of the storage operations used by a realistic e-commerce application.

**Time to consistency.** This metric measures the time between when a datum is written to the storage service and when all reads for the datum return consistent and valid results. Such information is useful to cloud customers, because their applications may require data to be immediately available with a strong consistency guarantee. Except for AppEngine, cloud providers do not support storage services that span multiple data centers. Therefore, we focus on consistency when the reads and writes are both done from instances inside the same data center.

**Cost per operation.** The final metric measures how much each storage operation costs. With this metric, a customer can compare the cost-effectiveness across providers.

*Intra-cloud Network*

The intra-cloud network connects a customer’s instances among themselves and with the shared services offered by a cloud. The performance of the network is vital to the performance of distributed applications. Within a cloud, the intra-datacenter network often has quite different properties compared to the inter-datacenter network. Providers vary in the type of network equipment (NICs, switches) as well as in their choice of routing (layer 2
vs. layer 3) and configuration such as VLANs. All providers promise high intra-datacenter bandwidth (typically on the order of hundreds of Mbps to Gbps), approximating a private data center network.

To compare the performance of intra-cloud networks, we use path capacity and latency as metrics. We use TCP throughput as a measure of path capacity because TCP is the dominant traffic type of cloud applications. Path capacity impacts data transfer throughput and congestion events can lead to errors or delayed responses. Path latency impacts both TCP throughput [87] and end-to-end response time. Together, these metrics provide insight into how a provider’s intra-cloud network is provisioned.

None of the providers charge for traffic within their data centers. Inter-datacenter traffic is charged based on the volume crossing the data center boundary. Since all providers charge similar amounts, comparing cost of network transfers becomes a moot point.

Wide-area Network

The wide-area network is defined as the collection of network paths between a cloud’s data centers and external hosts on the Internet. All the providers we study offer multiple locations to host customer applications. Requests from an end user can be served by an instance close to that user to reduce latency. Uniquely, AppEngine offers a DNS-based service to automatically map requests to close-by locations. The others require manual configuration.

We use the optimal wide-area network latency to compare providers’ wide-area networks. The optimal wide-area network latency is defined as the minimum latency between a vantage point and any data center owned by a provider. We use locations of PlanetLab nodes as vantage points. The more the data centers offered by a provider and the closer they are to population centers, the smaller the optimal network latency. The metric is useful for customers because it corresponds to the network latency an application may experience given an ideal mapping. For AppEngine, which provides automatic mapping of
requests to locations, we also measure how close its automatic mapping is to the optimal mapping.

3.4 Implementation

In this section, we describe the implementation details of CloudCmp and highlight the practical challenges we address.

3.4.1 Computation Metrics

**Benchmark tasks.** As described above, we would like a suite of benchmark tasks that stresses various aspects of the compute infrastructure offered by cloud providers. In traditional computation performance measurement, any benchmark suite, such as the SPEC CPU2006 benchmarks [41], would fit this bill. However, the context of cloud computing poses new constraints. For example, AppEngine only provides sand-boxed environments for a few cross-platform programming languages, and applications have to be single-threaded and finish within limited time.

To satisfy these constraints and be fair across different providers, we modified a set of Java-based benchmark tasks from SPECjvm2008 [42], a standard benchmark suite for Java virtual machines. We choose Java because it is supported by all cloud providers. The benchmark suite includes several CPU intensive tasks such as cryptographic operations and scientific computations. We augment it with memory and I/O intensive tasks. Each benchmark task runs in a single thread and finishes within 30 seconds so as to be compatible with all providers.

**Benchmark finishing time.** We run the benchmark tasks on each of the virtual instance types provided by the clouds, and measure their finishing time. Some instances offer multiple CPU cores for better parallel processing capability. For these instances, we also evaluate their multi-threading performance by running instances of the same benchmark.
task in multiple threads simultaneously, and measuring the amortized finishing time of the task. The number of threads is set to be equivalent to the number of available CPU cores.

**Cost per benchmark.** For cloud providers that charge based on time, we compute the cost of each benchmark task using the task’s finishing time and the published per hour price. For AppEngine that charges by CPU cycles, we use its billing API to directly obtain the cost.

**Scaling latency.** We write our own scripts to repeatedly request new virtual instances and record the time from when the instance is requested to when it is available to use. We further divide the latency into two segments: a provisioning latency and a booting latency. The former measures the latency from when an instance is requested to when the instance is powered on. The latter measures the latency from the powered-on time to when the instance is ready to use. The separation of the two is useful for a cloud provider to pinpoint the performance bottleneck during instance allocation.

### 3.4.2 Storage Metrics

**Benchmark tasks.** Along with each storage service, comes an API to get, put or query data from the service. Most APIs are based on HTTP. To use the APIs, we wrote our own Java-based client based on the reference implementations from the providers [12, 23, 51]. The client has a few modifications over the reference implementations to improve latency. It uses persistent HTTP connections to avoid SSL and other connection set up overheads. It also skips a step in some implementations in which a client first sends a request header and waits an RTT until the server returns an HTTP 100 (Continue) message before proceeding with the request body. For comparison, we also tested other non-Java-based clients such as **wget** and C#-based clients. To avoid the potential impact of memory or disk bottlenecks at the client’s instance, our clients mimic streaming workload that processes data as it arrives without retaining it in memory or writing it to disk.
Regarding the benchmark workload, we vary the size of the data fetched to understand the latency vs. throughput bottlenecks of the storage service. We vary the number of simultaneous requests to obtain maximum achievable throughput as well as measuring performance at scale. We vary the size of the working sets to observe both in- and out-of-cache performance. Because performance will be impacted by load on the client and in the network, we repeat each experiment at different times across different locations to get representative results. We also study the impact of the different client implementations described above.

**Response time.** The response time for an operation is the time from when the client instance begins the operation to when the last byte reaches the client.

**Throughput.** The throughput for an operation is the maximum rate that a client instance obtains from the storage service.

**Time to Consistency.** We implement a simple test to estimate the time to consistency. We first write an object to a storage service (the object can be a row in a table, a blob, or a message in a queue). We then repeatedly read the object and measure how long it takes before the read returns correct result.

**Cost per operation.** Similar to cost per benchmark task, we use the published prices and billing APIs to obtain the cost per storage operation.

### 3.4.3 Network Metrics

We use standard tools such as *iperf* [22] and *ping* to measure the network throughput and path latency of a provider. To measure intra-cloud throughput and latency, we allocate a pair of instances (in the same or different data centers), and run those tools between the two instances. Some providers further divide instances within the same data center into zones for a finer-grained control of instance location (e.g., not placing all instances in the same failure domain). In this case, we also deploy inter-zone and intra-zone instances
respectively to measure their throughput and latency.

To prevent TCP throughput from being bottlenecked by flow control, we control the sizes of the TCP send and receive windows. Our measurements show that with a 16MB window, a single TCP flow is able to use up the available capacity along the paths measured in this paper. Larger window sizes do not result in higher throughput. For comparison, we also measure the throughput obtained by TCP clients that use the default window size configured by the instance’s operating system.

To measure the optimal wide-area network latency, we instantiate an instance in each data center owned by the provider and ping these instances from over 200 vantage points on PlanetLab [37]. For each vantage point, the optimal latency to a provider is the smallest RTT to a data center of the provider. AppEngine automatically replicates our application to multiple instances at different data centers. By querying the DNS name corresponding to our application from each of the PlanetLab vantage points, we collect the IP addresses of the instance that AppEngine’s automatic mapping service maps the request from each vantage point. Each of these IP addresses, we conjecture, corresponds to the virtual IP address of the front-end load balancer at a data center. We then ping all of these IP addresses from each of the PlanetLab vantage points to identify the best mapping that AppEngine might have achieved.

3.5 Results

In this section, we present the comparison results between the four providers: AWS, Azure, AppEngine, and CloudServers. Due to legal concerns, we anonymize the identities of the providers in our results, and refer to them as $C_1$ to $C_4$. The data was collected over a two-month period from March to May 2010. We have made the data available for download from the project website [82].
Figure 3.1: The finishing time of benchmark tasks on various cloud instances. The time values are normalized using the longest finishing time to emphasize each provider’s relative performance. We show both single-threaded and multi-threaded results. The multi-threaded and I/O results are missing for $C_3$ because it does not support multi-threading or accessing the local disk.

3.5.1 Elastic Compute Cluster

We first measure the computation performance of different types of instances offered by cloud providers. Our naming convention refers to instance types as $\text{provider}.i$ where $i$ denotes the tier of service with lower numerals corresponding to lower cost and computational speed. For instance, $C_1.1$ is the cheapest and slowest instance type offered by $C_1$.

Table 3.3 summarizes the instances we measure. We test all instance types offered by $C_2$ and $C_4$, and the general-purpose instances from $C_1$. $C_1$ also provides specialized instances with more memory or CPU or high bandwidth network, which we do not include in this study, as they have no counterparts from other providers. $C_3$ does not offer different instance types, so we use its default environment to run the benchmark tasks.
Table 3.3: Information of the cloud instances we benchmark. With $C_3$, the first six CPU hours per day per application are free.

<table>
<thead>
<tr>
<th>Cloud Provider</th>
<th>Instance Type</th>
<th>Number of Cores</th>
<th>Price</th>
</tr>
</thead>
<tbody>
<tr>
<td>$C_1$</td>
<td>$C_{1.1}$</td>
<td>$&lt; 1$</td>
<td>$0.085 / hr$</td>
</tr>
<tr>
<td></td>
<td>$C_{1.2}$</td>
<td>2</td>
<td>$0.34 / hr$</td>
</tr>
<tr>
<td></td>
<td>$C_{1.3}$</td>
<td>4</td>
<td>$0.68 / hr$</td>
</tr>
<tr>
<td>$C_2$</td>
<td>$C_{2.1}$</td>
<td>4</td>
<td>$0.015 / hr$</td>
</tr>
<tr>
<td></td>
<td>$C_{2.2}$</td>
<td>4</td>
<td>$0.03 / hr$</td>
</tr>
<tr>
<td></td>
<td>$C_{2.3}$</td>
<td>4</td>
<td>$0.06 / hr$</td>
</tr>
<tr>
<td></td>
<td>$C_{2.4}$</td>
<td>4</td>
<td>$0.12 / hr$</td>
</tr>
<tr>
<td></td>
<td>$C_{2.5}$</td>
<td>4</td>
<td>$0.24 / hr$</td>
</tr>
<tr>
<td></td>
<td>$C_{2.6}$</td>
<td>4</td>
<td>$0.48 / hr$</td>
</tr>
<tr>
<td></td>
<td>$C_{2.7}$</td>
<td>4</td>
<td>$0.96 / hr$</td>
</tr>
<tr>
<td>$C_3$</td>
<td>default</td>
<td>N/A</td>
<td>$0.10 / CPU hr$</td>
</tr>
<tr>
<td>$C_4$</td>
<td>$C_{4.1}$</td>
<td>1</td>
<td>$0.12 / hr$</td>
</tr>
<tr>
<td></td>
<td>$C_{4.2}$</td>
<td>2</td>
<td>$0.24 / hr$</td>
</tr>
<tr>
<td></td>
<td>$C_{4.3}$</td>
<td>4</td>
<td>$0.48 / hr$</td>
</tr>
<tr>
<td></td>
<td>$C_{4.4}$</td>
<td>8</td>
<td>$0.96 / hr$</td>
</tr>
</tbody>
</table>

Some providers offer both Linux and Windows instances with the latter being slightly more expensive due to licensing fees. For experiments that depend on the type of OS, we compare instances from both OSes. For others, we choose Linux instances to reduce the cost of our experiment.

Figure 3.1 shows the finishing time of a CPU intensive task, a memory intensive task, and a disk I/O intensive task. Each bar shows the median and the 5th/95th percentiles of the measured samples. The same convention is used for all other figures without special notice. We omit the results of other benchmark tasks that show similar trends. For each instance type, we instantiate 10 instances and repeat each task 20 times per instance, i.e., a total of 200 samples per task per instance type. We only show the first four instance types of $C_2$ because the others have similar performance for the reason we soon describe. The I/O and multiple-threaded results are not available for $C_3$ because it does not support local disk access or multi-threading.
From the results, we can see that price-comparable instances offered by different providers have widely different CPU and memory performance. For example, $C_{4.1}$ and $C_{1.1}$ are in the same pricing tier with the former being only 30% more expensive per hour per per instance, but twice as fast as the latter. Alternatively, $C_{1.2}$ and $C_{1.3}$ offer better performance than their counterparts from $C_4$ and are on average 50% more expensive.

Note that across providers the instance types appear to be constructed in different ways. For $C_1$, the high-end instances ($C_{1.2}$ and $C_{1.3}$) have shorter finishing times in both single and multiple threaded CPU/memory tests. This is perhaps due to two reasons. First, besides having more CPU cores, the high-end instances may have faster CPUs. Second, the low-end instance may suffer from higher resource contention, due to high load and poor resource multiplexing techniques (e.g., CPU time sharing) that open a tenant to interference from other colocated tenants.

In contrast, for $C_4$, the finishing times do not improve significantly for the single
threaded tests when we alter the instances from low-end to high-end, while the amortized running times of the multi-threaded tests are greatly reduced. This suggests that all the $C_4$ instances might share the same type of physical CPU and they either have similar levels of resource contention or are better at avoiding interference.

Interestingly, instances of $C_2$ have the same performance regardless of their prices. This might be explained by the work-conserving CPU sharing policy of $C_2$, where a virtual instance can fully use all physical CPUs on a machine if there is no contention, and when colocated instances compete for CPUs, the high-end instances are given larger weight in the competition. Under such policy, we expect to observe interference and poor performance at times of high load. However, we found this to never happen in our experiments, suggesting that $C_2$’s data centers were lightly loaded throughout our experiment period.

Unlike CPU and memory intensive tasks, the disk I/O intensive task exhibits high variation on some $C_1$ and $C_4$ instances, probably due to interference from other colocated instances [55]. Further, the multi-threaded I/O performance is worse than the single-threaded performance, perhaps because interleaved requests from multiple threads are harder to optimize than requests from the same thread. On the contrary, instances from $C_2$ are much more stable perhaps due to better I/O scheduling techniques or lightly loaded physical machines.

*Performance at Cost*

Figure 3.2 shows the monetary cost to run each task. We see that for single-threaded tests, the smallest instances of most providers are the most cost-effective compared to other instances of the same providers. The only exception is $C_{1.1}$, which is not as cost-effective as $C_{1.2}$, because the latter has much higher performance due to faster CPU or lower contention.

Surprisingly, for multi-threaded tests the high-end instances such as $C_{1.3}$ and $C_{4.4}$ with more CPU cores are not more cost-effective than the low-end ones. There are two
possible reasons. First, the prices of high-end instances are proportional to the number of CPU cores, and thus do not provide any cost advantage per core. Second, although high-end instances are assigned more CPU cores, they still share other system resources such as memory bus and I/O bandwidth. Therefore, memory or I/O intensive applications do not gain much by using high-end instances as long as the applications do not run out of memory or disk space. This suggests that for parallel applications it might be more cost-effective to use more low-end instances rather than fewer high-end ones.

Scaling Latency

Finally, we compare the scaling latency of various providers’ instances. To save cost, we only measure the scaling latency for the smallest instance of each provider. For providers that support both Linux and Windows instances, we test both choices to understand how different OSes affect the scaling latency, especially the booting latency. We run Ubuntu 9.04 for Linux instances and Windows Server 2008 for Windows ones. For each cloud and each OS type, we sequentially allocate 20 instances and measure the time between the request for a new instance and when that instance becomes reachable. We attribute the kernel uptime of the instance once it becomes available to be the time to boot and the

FIGURE 3.3: The scaling latencies of the lowest end instance of each cloud provider.
Figure 3.4: The cumulative distribution of the response time when using the large table with 100K entries. Note that for the query operation, the x-axis is in a logarithmic scale, due to the significant performance gaps between different services.

remaining latency as the time to provision or otherwise set up the VM. We drop $C_3$ here because it does not allow manual requests for instances.

Figure 3.3 shows the scaling latencies for three clouds and different OS types. All cloud providers can allocate new instances quickly with the average scaling latency below 10 minutes. $C_1$ and $C_2$ can even achieve latency within 100 seconds for Linux instances. The latency of $C_4$ is larger. We see that across providers, Windows instances appear to take longer time to create than Linux ones, but for different reasons. For $C_1$, the provisioning latency is similar for both instances, but the Windows ones have larger booting latency, possibly due to slower CPUs of the smallest instances. Windows Server 2008 R2 instances based on the Win7 code base can boot faster. For $C_2$, the booting latency is similar, while the provisioning latency of the Windows instances is much larger. It is unclear but likely that $C_2$ may have different infrastructures to provision Linux and Windows instances.
We note that a few other factors impacting scaling agility have not been considered here. Rather than focusing just on allocating instances quickly, some providers make it easy to update active instances or provide extensive monitoring and automatic recovery from faults during running. Another common goal is to create instances consistently within an SLA deadline even when they are brought up in large batches.

3.5.2 Persistent Storage

We measure and compare three types of storage services: table, blob, and queue, offered by various cloud providers.

Table Storage

We first compare three table storage services offered by $C_1$, $C_3$, and $C_4$. $C_2$ does not provide a table service. Here we only show the results obtained using our Java-based client, as other non-Java clients achieve similar performance, because the table operations are lightweight on the client side. For each table service, we test the performance of three operations: get, put, and query (see Table 3.2). Each operation runs against two pre-defined data tables: a small one with 1K entries, and a large one with 100K entries. The get/put operations operate on one table entry, while the query operation returns on average 10 entries. For storage benchmarks, unless otherwise specified, we use instance types that occupy at least one physical core to avoid excessive variation due to CPU time sharing.

Figure 3.4 shows the distributions of response time for each type of operation on the large table. We repeat each operation several hundred times. The results of using the small table show similar trends and are omitted to save space. From the figure, we can see that all table services exhibit high variations in response time. For example, the median response time of the get operation is less than 50ms, while the 95th-percentile is over 100ms for all services. The three services perform similarly for both get and put operations, with
Figure 3.5: The cumulative distribution of the time to consistency for $C_1$’s table service.

$C_3$ slightly slower than the other two providers. However, for the query operation, $C_1$’s service has significantly shorter response time than the other two. $C_4$’s service has a very long response time, because unlike the other providers, it does not appear to maintain indexes over the non-key fields in a table. In contrast, $C_1$ appears to have an indexing strategy that is better than the others.

We also measure how well each table service scales by launching multiple concurrent operations. None of the services show noticeable performance degradation when up to 32 operations are issued at the same time. This suggests that all these table storage services appear to be reasonably well provisioned. Testing at higher scale is deferred to future work.

We then evaluate the time to reach consistency for the table services by using the mechanism described in §3.4.2. We discover that around 40% of the get operations in $C_1$ see inconsistency when triggered right after a put, and do not return the entry just inserted by the put. Other providers exhibit no such inconsistency. Figure 3.5 shows the distribution of the time to reach consistency for $C_1$. From the figure, we see that over 99% of the inconsistencies are resolved within 500ms, with the median resolved within 80ms. The long duration of the inconsistency opens up a sizable window for race conditions.
Figure 3.6: The cumulative distribution of the response time to download or upload a blob using Java-based clients.

Table 3.4: The average cost per operation for all three table services. The cost is in the unit of milli-cent, i.e., one-thousandth of a cent.

<table>
<thead>
<tr>
<th>Provider</th>
<th>get</th>
<th>put</th>
<th>query</th>
</tr>
</thead>
<tbody>
<tr>
<td>$C_1$</td>
<td>0.13</td>
<td>0.31</td>
<td>1.47</td>
</tr>
<tr>
<td>$C_3$</td>
<td>0.02</td>
<td>0.23</td>
<td>0.29</td>
</tr>
<tr>
<td>$C_4$</td>
<td>0.10</td>
<td>0.10</td>
<td>0.10</td>
</tr>
</tbody>
</table>

$C_1$ does provide an API option to request strong consistency but disables it by default. We confirm that turning on this option eliminates inconsistencies and surprisingly does so without much extra latency for get or put. We conjecture that the performance overhead of enforcing consistency might be visible only when more users request it or during failure cases because otherwise $C_1$ would have enabled it by default.

Finally, we compare the cost per operation for different table services in Table 3.4.
We can see that, although the charging models of the providers are different, the costs are comparable. Both $C_1$ and $C_3$ charge lower cost for get/put than query, because the former operations are simpler and consume less CPU cycles to serve. $C_4$ charges the same across operations and can improve its charging model by accounting for the complexity of the operation. This is an example of how CloudCmp’s benchmarking results can help the providers make better choices.

A comparison with the compute costs in Table 3.3 indicates that the cost of table storage is comparable to that of compute instances for workloads that trigger about 1000 operations per instance per hour. Applications that use storage at a lower rate can choose their provider based on their computation costs or performance.

**Blob Storage**

We compare the blob storage services provided by $C_1$, $C_2$, and $C_4$. $C_3$ does not offer a blob store.

Figure 3.6 shows the response time distributions for uploading and downloading one blob measured by our Java-based clients. We consider two blob sizes, 1KB and 10MB, to measure both latency and throughput of the blob store. The system clock of $C_2$’s instances have a resolution of 10ms, and thus its response time is rounded to multiples of 10ms. We see that the performance of blob services depends on the blob size. When the blob is small, $C_4$ has the best performance among the three providers. When the blob is large, $C_1$’s average performance is better than the others. This is because blobs of different sizes may stress different bottlenecks – the latency for small blobs can be dominated by one-off costs whereas that for large blobs can be determined by service throughput, network bandwidth, or client-side contention. Uniquely, $C_2$’s storage service exhibits a 2X lower performance for uploads compared to downloads from the blob store. It appears that $C_2$’s store may be tuned for read heavy workload.

Figure 3.7 illustrates the time to download a 10MB blob measured by non-Java clients.
Figure 3.7: The cumulative distribution of the time to download a 10MB blob using non-Java clients.

Figure 3.8: The blob downloading time from each blob service under multiple concurrent operations. The number of concurrent requests ranges from 1 to 32. Note that the x-axes are on a logarithmic scale.

Compared to Figure 3.6(c), in every provider, non-Java clients perform much better. Markedly $C_4$’s performance improves by nearly 5 times because it turns out that the Java implementation of their API is particularly inefficient.

We then compare the scalability of the blob services by sending multiple concurrent operations. As per above, we use non-Java clients to eliminate overheads due to Java. Figure 3.8 shows the downloading time with the number of concurrent operations ranging from 1 to 32. We omit the results for uploading because they are similar in trend. When
Figure 3.9: The cumulative distributions of the response time to send or retrieve a message from a queue, and the propagation delay of a queue.

Table 3.5: The maximum throughput an instance obtains from each blob service when downloading many 10MB blobs concurrently.

<table>
<thead>
<tr>
<th>Provider</th>
<th>Smallest instance with at least one core (Mbps)</th>
<th>Largest instance from a provider</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Maximum throughput achieved by one instance</td>
<td></td>
</tr>
<tr>
<td>C₁</td>
<td>773.4</td>
<td>782.3</td>
</tr>
<tr>
<td>C₂</td>
<td>235.5</td>
<td>265.7</td>
</tr>
<tr>
<td>C₄</td>
<td>327.2</td>
<td>763.3</td>
</tr>
</tbody>
</table>

The blob size is small, all services except for C₂ show good scaling performance. This suggests that C₁ and C₄’s blob services are well-provisioned to handle many concurrent operations. When the blob is large, C₄ and C₁ continue to scale better though all three providers show certain scaling bottlenecks. The average time to download increases with the number of concurrent operations, illustrating a throughput bottleneck.
We compute the maximum throughput of the blob service that one instance can obtain by increasing the number of simultaneous operations until the throughput stabilizes. Table 3.5 shows the results for two types of instances – the smallest instance that has at least one full core and the largest instance from each provider. The former data point eliminates CPU time sharing effects [103] while the latter minimizes contention from colocated VMs. We report results with non-Java clients to eliminate overheads due to client side inefficiency. As we will soon show, \( C_4 \)’s blob service throughput is close to their intra-datacenter network bandwidth (see Figure 3.10), suggesting that the bottleneck in throughput is unlikely within the blob service itself. This is also true for \( C_4 \)’s blob service throughput of a large instance, which more than doubles that of the single core instance that we were using earlier. It appears that the impact on an instance’s throughput due to other VMs colocated on the machine is non trivial in \( C_4 \).

In summary, we observe that client implementation and contention from other VMs or along network paths significantly impact the perceived storage service performance (across the three cloud platforms we study). Therefore, a more efficient client implementation or less contention may improve the results in this paper.

We tested the consistency property of the blob services, and did not find inconsistency problems in the three providers. The charging models are similar for all three providers and are based on the number of operations and the size of the blob. We omit these results for brevity.

**Queue Storage**

We compare the queue services of \( C_1 \) and \( C_4 \). Similar to table services, we only show the results from our Java-based client. Figure 3.9(a) and 3.9(b) show the distributions of the response time of sending and retrieving a message. The queue services are designed to transfer only small messages up to 8KB. Thus, we choose a message size of 50B in our measurement. The results show that both services have large variations in response time.
$C_4$ is slightly faster at sending messages while $C_1$ is faster at retrieving messages. We test the scalability of the queue services, up to 32 concurrent messages, and no significant performance degradation is found, mostly due to the small message size.

It is interesting to note that the response time of the queue service is on the same order of magnitude as that of the table and blob services. This suggests that although the queue service is simpler and designed to be more efficient compared to the other storage services, the performance gain is insignificant. One may use the table or blob service to implement a simple signaling framework with similar functionality as the queue service without much performance degradation.

We then measure the propagation delay of a queue. The delay is defined as the time between when a message is sent to an empty queue and when it is available to be retrieved. A queue with shorter propagation delay can improve the responsiveness of an application. Figure 3.9(c) shows the distribution of the propagation delay of the two services. We see that roughly 20% of the messages for $C_1$ take a long time (>200ms) to propagate through the queue, while $C_4$’s queue service has a similar median propagation delay but lower variation. Finally, both services charge similarly—1 cent per 10K operations.

3.5.3 Intra-cloud Network

In this section, we compare the intra-cloud network performance of different providers. As of April 27th, when we conducted this measurement, we found a total of 11 data center locations from three providers: 6 for $C_4$, 3 for $C_1$, and 2 for $C_2$. Similar to instance types, we name each data center as provider $D_{Ci}$. Table 3.6 summarizes the geographic locations of the data centers. We do not consider $C_3$’s intra-cloud network performance because it does not allow direct communication between instances.

When measuring the intra-cloud bandwidth and latency, we choose the instance types that can at least fully occupy one CPU core. This is to avoid the network performance degradation due to CPU time sharing introduced by virtualization [103].
Table 3.6: The geographical location of the cloud data centers.

<table>
<thead>
<tr>
<th>Cloud Provider</th>
<th>Data Center Name</th>
<th>Location</th>
<th>Region</th>
</tr>
</thead>
<tbody>
<tr>
<td>$C_1$</td>
<td>$C_1.DC_1$</td>
<td>North Virginia</td>
<td>US</td>
</tr>
<tr>
<td></td>
<td>$C_1.DC_2$</td>
<td>North California</td>
<td>US</td>
</tr>
<tr>
<td></td>
<td>$C_1.DC_3$</td>
<td>Ireland</td>
<td>Europe</td>
</tr>
<tr>
<td>$C_2$</td>
<td>$C_2.DC_1$</td>
<td>Dallas/Fort Worth, Texas</td>
<td>US</td>
</tr>
<tr>
<td></td>
<td>$C_2.DC_2$</td>
<td>Chicago, Illinois</td>
<td>US</td>
</tr>
<tr>
<td>$C_4$</td>
<td>$C_4.DC_1$</td>
<td>Chicago, Illinois</td>
<td>US</td>
</tr>
<tr>
<td></td>
<td>$C_4.DC_2$</td>
<td>Amsterdam, Netherlands</td>
<td>Europe</td>
</tr>
<tr>
<td></td>
<td>$C_4.DC_3$</td>
<td>San Antonio, Texas</td>
<td>US</td>
</tr>
<tr>
<td></td>
<td>$C_4.DC_4$</td>
<td>Singapore</td>
<td>Asia</td>
</tr>
<tr>
<td></td>
<td>$C_4.DC_5$</td>
<td>Dublin, Ireland</td>
<td>Europe</td>
</tr>
<tr>
<td></td>
<td>$C_4.DC_6$</td>
<td>Hong Kong</td>
<td>Asia</td>
</tr>
</tbody>
</table>

**Figure 3.10:** The intra-datacenter TCP throughput between two instances in all data centers we measure.

_Intra-datacenter Network_

Figure 3.10 shows the TCP throughput between two instances in the same data center. In rare cases, a pair of instances are colocated on the same machine and obtain a throughput larger than 1Gbps which is the speed of the NIC. We filter out such pairs as they do not measure actual network performance. We combine the results for intra-zone and inter-zone cases, because the difference between them is not significant. From the figure, we see that
the network bandwidth of providers differs significantly. $C_1$ and $C_4$ provide very high TCP throughput which is close to the limit of the NIC (1Gbps) and the variation is low. $C_2$ has much lower throughput, probably due to throttling or under-provisioned network.

In terms of latency, all data centers achieve low round trip time (< 2ms) for all pairs of instances we test. This result is not surprising, because the instances located in the same data center are physically proximate. We omit the results to save space.

**Inter-datacenter Network**

Next, we show the performance of network paths between data centers of the same provider. Because most providers focus on the US market (and some only have US data centers), we only show the results for data centers within the US. Figure 3.11 shows that the throughput across datacenters is much smaller than that within the datacenter for all providers. Both $C_1$ and $C_4$ have their median inter-datacenter TCP throughput higher than 200Mbps, while $C_2$’s throughput is much lower. Further, the variation in throughput across datacenters is higher since the wide-area traffic may have to compete with much other traffic.

We also compare the above result with the inter-datacenter throughput obtained by a vanilla client that uses one TCP flow with the default send and receive window sizes con-

![Figure 3.11: The TCP throughput between two different US data centers of a cloud provider.](image)
Figure 3.12: This figure shows the cumulative distribution of the optimal round trip time (RTT) to the instances deployed on a cloud provider from 260 global vantage points. For $C_3$ we also show the actual RTT from a vantage point to the instance returned by the cloud’s DNS load balancing.

figured by the provider. All providers see a smaller throughput with this vanilla client. This is because the network paths across data centers have a high bandwidth-delay product (e.g., $50\text{ms} \times 800\text{Mbps} = 5\text{MB}$) and the default window sizes are not configured appropriately. We find that the degradation is less with Linux instances, since modern Linux kernels auto-tune the size of the TCP receive buffer [30] which can grow up to 4MB in the default setting. However the degradation is far worse in the non-Linux instances since either auto-tuning is not turned on or is configured poorly.

We find that the latencies between data centers largely correspond to the geographical distance between the data centers. The latencies across providers are incomparable because their data centers are at different locations. Hence, we omit these results.

3.5.4 Wide-area Network

In this section, we compare the wide area network performance of different providers. Figure 3.12 shows the distribution of the optimal wide-area latency observed from a diverse set of vantage points. We also show the actual latency of $C_3$ which is the latency to the
instance returned by its DNS load balancing system. From the figure, we can see that both the optimal and actual latency distributions of \( C_3 \) are lower than that of other providers. This could be explained by the provider’s widely dispersed presence – we observe 48 unique IP addresses simultaneously serving our test application. These IP addresses likely correspond to the virtual IPs of front-end load balancers at distinct locations. Furthermore, the gap between the optimal and actual latency of \( C_3 \) is less than 20ms, suggesting that its load balancing algorithm works very well in practice.

\( C_1 \) and \( C_4 \) have similar latency distributions: they are worse than \( C_3 \), but much better than \( C_2 \). A main difference is that \( C_1 \) has a larger fraction of vantage points that have an optimal latency higher than 100ms. Closer examination of our data reveals that these high latency vantage points are mostly in Asia and South America, where \( C_1 \) does not have a presence (Table 3.6).

\( C_2 \) has the worst latency distribution because it has the smallest number of data centers. The flat curve between 50ms and 100ms corresponds to the latency differences between two groups of vantage points: the North American nodes and those in other continents. Because all \( C_2 \)’s data centers reside within the US, there is a latency jump for vantage points outside North America.

3.6 Using CloudCmp: Case Studies

In this section, we deploy three simple applications on the cloud to check whether the benchmark results from CloudCmp are consistent with the performance experienced by real applications. If they are, it validates our conjecture that CloudCmp’s results can be used by customers to choose cloud providers in lieu of porting, deploying, and measuring their applications on each cloud. The applications include a storage intensive e-commerce website, a computation intensive application for DNA alignment, and a latency sensitive website that serves static objects. This study is a first step. We consider the case for
Fig. 3.13: The page generation time of TPC-W when deployed on all three cloud providers that support it. The y-axis is in a logarithm scale.

arbitrary applications in Chapter 4.

3.6.1 E-commerce Website

The first application we choose is TPC-W [47], a standard benchmark for transactional web services. TPC-W itself is a fully functional e-commerce website where customers can browse and purchase books online. It is advantageous for such applications to migrate to a cloud computing utility, because the cloud provides highly scalable and available storage services and free intra-datacenter bandwidth.

We use a Java implementation of TPC-W [24] and port it to various cloud providers by redirecting the database operations to use each cloud’s table storage APIs. However, not all database operations used in TPC-W map directly to the APIs offered by the cloud. Specifically, there is no equivalent for JOIN and GROUP BY. We disable pages that use these operations. Out of the original 16 pages in TPC-W, we had to disable four pages. The three operations we benchmarked (put, get, and query) account for over 90% of TPC-W’s storage operations. Other operations include delete and count.

One major performance goal of TPC-W, similar to other dynamic web applications, is to minimize the page generation time. The performance bottleneck lies in accessing
table storage. From CloudCmp’s comparison results of table storage service, shown in Figure 3.4, we see that cloud $C_1$ offers the lowest table service response time among all providers. That benchmark appears relevant because the table size it uses (100K) is on the same order as that of the TPC-W tables (100K - 400K), and because it covers most of the storage operations used by the ported version of TPC-W. Therefore, from the benchmarking results, a customer may guess that $C_1$ will offer the best performance for TPC-W.

To verify this, we deploy TPC-W on the three providers that offer table storage service: $C_1$, $C_3$, and $C_4$. We use instances that occupy a physical core to avoid interference due to CPU sharing. Figure 3.13 shows the page generation time for all twelve pages. We see that $C_1$ indeed has the lowest page generation time among all three providers, consistent with our benchmarking result. Furthermore, $C_4$ has lower generation time than $C_3$ for most pages except for pages 9 and 10. These pages contain many query operations and are consistent with CloudCmp’s results in Figure 3.4, where $C_4$ has a much higher query response time but lower get and put response time than $C_3$.

3.6.2 Parallel Scientific Computation

We then test Blast, a parallel computation application for DNA alignment. We choose Blast because it represents computation-intensive applications that can take advantage of the cloud computing utility. The application is written in C#, with one instance running a web service to accept job input and return job output, and multiple worker instances responsible for executing the jobs in parallel. Blast instances communicate with each other through the queue storage service, which serves as a global messaging system. The application also leverages the blob storage service to store computation results.

We consider two cloud providers for Blast: $C_1$ and $C_4$. The others do not support queue service. The performance goal of Blast is to reduce job execution time given a budget on number of instances. CloudCmp’s computational benchmark results shown in Figure 3.1
suggest that at a similar price point, $C_{4.1}$ performs better than $C_{1.1}$.

To check this prediction, we deploy Blast on both types of instance ($C_{1.1}$ and $C_{4.1}$) and compare the real execution time of five example jobs. Figure 3.14 shows the results. For all five jobs, Blast running on $C_{4.1}$ takes only a portion of the time it takes when running on $C_{1.1}$. This suggests that $C_{4.1}$ indeed offers better performance than $C_{1.1}$ for real applications at similar price point, consistent with CloudCmp’s benchmarks.

### 3.6.3 Latency Sensitive Website

We choose a latency sensitive website for our third case study. We configure a simple web server to serve only static pages, and download the pages from PlanetLab nodes around the world. The performance goal is to minimize the page downloading time from many vantage points, and the main performance bottleneck is the wide area network latency.

We choose this application because many existing online services such as web search and online gaming depend critically on network latency [78]. Due to TCP semantics, shorter latency also often leads to higher throughput [87].

We deploy our website on all providers and use `wget` to fetch web pages from PlanetLab
nodes. Each vantage point fetches from the instance with the minimum network latency to emulate a perfect load balancing scheme. Figure 3.15 shows the distributions of the page downloading time for various providers. We download two web pages of sizes 1KB and 100KB. In both cases, $C_3$ has the smallest page downloading time, consistent with our benchmarking results in Figure 3.12 that show $C_3$ having the lowest wide-area network latency distribution.

3.7 Discussion

In this section, we discuss some CloudCmp’s limitations.

**Breadth vs. depth trade-off.** As our main goal is to perform a comprehensive comparison among cloud providers, in several occasions, we sacrifice depth for breadth. For instance, an in-depth study that focuses on the storage service of a particular provider could have used more storage system benchmarks [63] in addition to the metrics we use, to examine factors such as pre-fetching and other query optimization techniques. The results presented in this paper show only the first-order differences among various providers, and can be complemented with more in-depth measurement results on each individual node.
provider.

**Snapshot vs. continuous measurement.** The results in this paper should be viewed as a snapshot comparison among cloud providers in today’s market. As time goes by, providers may upgrade their hardware and software infrastructure, and new providers may enter the market. It is our future work to use CloudCmp to continually update those results.

### 3.8 Summary

In this chapter, we have introduced which we believe to be the first systematic comparison tool for public cloud providers. We address some key challenges to scope the problem to one that is manageable given bounded money and time and yet is meaningful to predict the performance of real applications. We have then built a novel collection of tools that together enable effective cloud comparison.

Our measurement results represent the first comprehensive view of the performance and cost characteristics of production cloud providers. They are relevant to application performance, and also reveal a few under-performing regions for almost every provider we measure. More importantly, we observe dramatic performance and cost variations across providers in their virtual instances, storage services, and network transfers, with no single provider winning in all aspects. This underscores the need for careful cloud environment selection, especially for complex applications that rely on multiple cloud services.
4

CloudProphet: Estimating Application Behavior in the Cloud

4.1 Introduction

In the previous chapter, we showed that the cloud environments can have dramatically different performance and cost characteristics across many application-related dimensions (CPU speed, storage response time, network bandwidth, etc.) Therefore, it is critical to understand how an application would behave in every candidate cloud environment before picking the most suitable one.

A straightforward approach to assess an application’s performance is to port the application to each candidate environment. This approach can accurately assess an application’s behavior, but requires onerous manual porting efforts and can be challenging in practice. This is because applications are complex. They may have many different components (e.g., front end load balancers, application servers, database back-ends), dependencies between these components, and often rely on third party software that may also need to be ported along with the applications. All of this is true for applications that we use in our evaluation. Furthermore, there are many clouds, and many feature sets to choose from in each cloud.
Although cloud platforms usually support virtual machine images for fast startup, these images are often not transferable across platforms. As a result, the application owner will have to port and re-configure the application for each of the cloud platforms and feature sets that she intends to evaluate.

Another common approach is modeling. A plethora of work [96, 95, 94, 100] has emerged to estimate the application performance after varying different environment and application parameters, such as the component placement [96], physical CPUs [95], workload mix [94], and the number of replicas [96, 100]. Though the modeling approach allows efficient exploration of the parameter space without onerous application porting effort, it requires extensive expert effort to build and calibrate these models [110], while each of them only captures one or few aspects of the environment and the application. A holistic model that can accurately predict an application’s behavior by capturing all dimensions of a complex networked application running in a completely new environment (e.g., a cloud platform) is still missing, probably because of the sheer complexity to describe everything analytically.

In this chapter, we aim to achieve the best of both worlds by exploring a middle approach to predict application behavior. We ask the question: can we estimate the performance of a networked application in a new environment without application porting yet without abstracting away the essential details of the application and its environment? Our answer is both yes and no. We discover that for generic applications, behavior prediction is extremely challenging (if not intractable), as an application can behave as an adversary to defeat a prediction method. For instance, an application can actively probe the available network bandwidth in its running environment, and change the number of TCP flows based on the probing result. In this case, one cannot predict an application’s behavior without running it in a new environment.

Despite the negative finding for generic applications, we find that we can predict the performance of a broad range of practical applications in different cloud environments.
without porting or analytically modeling them, as long as these applications do not vary their application logic when environments change and the behavior of the application follows well-known patterns, such as the dispatcher-worker pattern commonly seen in server applications.

Our approach is CloudProphet. Instead of porting or trying to capture the characteristics of both application and environment through abstract models, we estimate the application’s behavior by running a set of high-fidelity shadows inside a target environment, in place of the real application components. The shadows mimic the resource usage of their corresponding components with easy-to-port code agnostic to the actual application logic. For instance, a shadow will read or write the same number of bytes through disk or network I/O as its corresponding component. Also, CloudProphet enforces the same inter-component dependencies among the shadows as the real application. For instance, a shadow for a front-end server will contact the back-end shadow in the same way as a real server.

Our evaluation shows that mimicking just the resource usage and the dependencies suffices to estimate a variety of practical applications’ behavior in a new environment with high accuracy. This result holds for the applications and cloud platforms we tested. The cloud platforms we tested differ from each other along many dimensions, including CPU types, network conditions, and interference levels. This is our primary contribution. We can estimate the performance seen by users, the scalability of the service, and the expected cost all without migrating the actual application.

There are a couple reasons why this method works. First, at the level of individual components, we find that the performance obtained by a component in a new environment and the monetary cost depend exclusively on resource usage. Cloud billing policies measure and bill based on the usage of a few key resources (§ 3.3.3). Similarly, performance is a function of the infrastructure (hardware and software) and the competition from others using the same infrastructure. We find that both of these can be exercised by replaying
the actual resource demands. Second, at the level of an entire networked application, we find inter-component dependency is critical to the end-to-end performance, and it can be extracted and enforced precisely.

The key challenge of CloudProphet is how to create high-fidelity shadows at low cost. Manually modifying an application’s source code and abstracting away the unnecessary details is not an option, because the process can be extremely time-consuming and even impractical for proprietary applications. We discover that one can create high-fidelity shadows from traces of the application running in its local environment. This is because the traces succinctly encode the application’s resource demands. Furthermore, with some knowledge of the programming patterns, one can also accurately extract the application dependencies from the traces. The shadows then replay the traced resource demand in the cloud following the extracted dependencies. Although relying on programming patterns limits the generality of the system, we argue it is a necessary compromise because accurately and efficiently inferring the dependency of generic applications can be very challenging [58, 97]. The current version of CloudProphet focuses primarily on the dispatcher-worker pattern widely adopted by many server implementations. In the future, we plan to extend CloudProphet to cover other popular patterns as well, such as the event-driven pattern and the staged event-driven architecture (SEDA) [105].

We implemented a prototype of CloudProphet in C++ and evaluated it with a variety of applications inside three popular public clouds: Amazon’s AWS [9], Rackspace’s Cloud-Servers [38], and Joyent [27]. In our realistic case studies, we found that CloudProphet can help customers identify the best-performing and the most cost-effective cloud environment for two multi-tiered web applications RUBiS [40] and MediaWiki [32]. We also applied CloudProphet to a number of practical applications ranging from video encoding to source version control, and found it can achieve high prediction accuracy across all cases (relative error < 18%). Finally, we found CloudProphet has reasonable tracing and low deployment overhead. All the evidences suggest CloudProphet are likely to be useful
The remaining sections in this chapter are organized as follows. In § 4.2 we present an overview of the system and the key challenges. In § 4.3 and § 4.4 we describe the detailed design and implementation of the system. In § 4.5 we present our experience of using the system with practical applications and real clouds. Finally, we conclude in § 4.6.

4.2 Overview

How can we predict the performance of an application in a new environment without going through the pain of porting it? The basic idea is rather simple: replace each component of the application with a shadow that mimics the component’s behavior, i.e., takes the same inputs, generates the same outputs, and makes the same demands on all resources (computation, memory, storage, network, . . .), but is made up of very simple code and hence is trivial to port. If we can create these shadows, predicting behavior is easier – deploy the shadow in the candidate cloud environments and measure the observed performance, cost, and scaling behavior.

One way to generate the shadows is to refactor the code and remove the logic that is unrelated to the application’s behavior. However, code refactoring usually requires manual work and thorough understanding of the application logic. Further, it might not be even possible to simplify code beyond a point. Therefore, we look for an approximation that can achieve similar accuracy as manual code refactoring, but is much simpler and requires little human-intervention.

In CloudProphet, we propose to trace the behavior of each component, i.e., its inputs, outputs and resource usage. The shadow then replays the collected trace while exerting the same demands on every resource. This proposal differs from other uses of trace-and-replay in the goal– predicting application behavior. The new goal lets us deemphasize some aspects– we require less precision when compared to using trace-and-replay for
debugging applications [93, 71] or detecting race conditions in multi-core servers [88]. But a few novel aspects become more important. For example, to predict performance, cost, and scaling, especially in shared environments it is absolutely essential that the shadow exert the exact same demands on all resources as the original. Further, since most of the target applications have many components (e.g., front-end web servers, databases, caches) it is also important to capture and replay the inter-component dependencies accurately.

CloudProphet traces at system call boundaries and tags each trace event by the thread that generates it. System call parameters and return values let us capture inputs and outputs. Some resource use is implicitly captured in the call parameters, e.g., network transfers. CloudProphet explicitly captures the usage of resources between every pair of successive trace events – e.g., the number of CPU cycles used.

4.2.1 Challenges

**Resource-Mimic’ing Replay:** CloudProphet sets up a shadow for each component (application thread) and replays the trace events such that between each event pair the shadow uses the same resources as the original.

A primary challenge is to translate the resources used in the observed environment to those that will be used in the new environment. This is easy for some resources— network transfer sizes are likely to remain the same, but not for others— how to translate a CPU use of x cycles (or y seconds) to the different types of CPUs in the new environments. We will show in §4.3.1 how CloudProphet mimics each of the various resource types.

A subsidiary challenge is the choice of resources to mimic since there is a trade-off between fidelity and overhead. CloudProphet mimics all the major resource types— computation, storage, network, and some Inter-Process Communication (IPC) such as locks. Our results show that these resource types are necessary— there are applications and environments that can bottleneck at each of these resources, and hence, not mimic’ing them leads to poor predictions. We note also that CloudProphet does not trace some other
Table 4.1: Popular servers in different domains that follow the dispatcher-worker pattern.

<table>
<thead>
<tr>
<th>Domain</th>
<th>Servers</th>
</tr>
</thead>
<tbody>
<tr>
<td>Web</td>
<td>Apache, IIS, Tomcat</td>
</tr>
<tr>
<td>RPC</td>
<td>Java RMI, ASP.NET, Hadoop RPC</td>
</tr>
<tr>
<td>Middleware</td>
<td>JBoss, PHP fastCGI, JOnAS, WebSphere</td>
</tr>
<tr>
<td>Authentication</td>
<td>Kerberos, OpenLDAP</td>
</tr>
</tbody>
</table>

Inter-component Dependencies: So far, we worried about replaying individual trace events and ensuring that the resource use is consistent. On the other hand, to mimic the application behavior end-to-end, it is also important to infer the inter-component dependencies of the application, because the dependencies determine the correct order to replay the events.

We use two examples to illustrate why it is critical to infer and replay the right inter-component dependencies. In Figure 4.1, two threads are synchronized with a pair of network messages. Though the events on the threads seem parallel, they should not be
FIGURE 4.2: This figure shows how CloudProphet replays the behavior of each application component leveraging the dispatcher-worker pattern. CloudProphet uses its own dispatcher to mimic the real dispatcher’s behavior, and replays the events in each request’s handler.

replayed simultaneously, because it will violate the inter-component dependencies introduced by the messages. Such violation can change the running time of the threads, which in turn affects the prediction accuracy. In some cases, it is also important to avoid any fake ordering between events. Consider a more complex example in Figure 4.2. A dispatcher thread distributes incoming requests among worker threads. In the trace (see left), it happens that requests A and B are served by the leftmost thread. From the events on that thread, it may seem that request B must be handled after request A. However, in the new environment, request A may take longer to process causing request B to be served concurrently by a different thread (see right). The underlying problem is that the trace collects events in chronological order, but the only order that will be preserved across environments is that between dependent work. But, in general, it is hard to infer dependencies.

CloudProphet focuses on what we call the worker-dispatcher pattern that makes it easy to infer dependencies. Akin to the Figure 4.2, one or more dispatcher threads issue work to a pool of worker threads. We choose this pattern as a starting point, because it is simple
and widely used by a variety of server software. Table 4.1 shows some popular software in diverse domains that employ the pattern. A key reason for the popularity of this pattern is that these workers are easier to write compared to event-driven handlers. For example, workers can be arbitrary programs that may block for any reason, yet the OS can remain efficient by scheduling other workers.

Details on how CloudProphet leverages this pattern are in §4.3.2. In short, CloudProphet uses a special shadow to mimic the dispatcher. Unlike other shadows that simply stress the desired amount of resources (e.g., via busy loops), this shadow mimics the functionality of the dispatcher, i.e., assigns incoming requests to the idle shadow workers. Extending to similar design patterns, such as producer-consumer, is doable but we do not yet know how to cope with arbitrary dependencies.

4.2.2 What CloudProphet skips (for now)

CloudProphet is only the first step in aiding cloud environment selection for general applications. A few problems remain. First, there may be more effective ways of searching through the space of potential environments instead of deploying shadows on every one. It may be possible to interpolate (e.g., predict behavior on a medium instance in AWS based on that in the small and large instances). Inherently, such short-cuts in search are an extra source of prediction error and so we ignore them for now. Second, CloudProphet assumes that applications do not adapt their behavior to the environment; for example, applications that infer resource availability and auto-tune themselves can behave differently than their traces. Also, adaptations that change the amount of work to be done (resources used) for the same input/output are harder to deal with. Third, applications, whose behavior crucially depends on resources that CloudProphet’s shadows do not mimic (e.g., cache accesses), will receive poor predictions. Finally, CloudProphet does not capture dependencies other than worker-dispatcher. While these last few assumptions reduce the scope of CloudProphet, we surprisingly found that CloudProphet suffices to accurately predict
the behavior of a variety of practical applications in real cloud environments.

4.3 Design Details

Here, we describe how CloudProphet traces and replays to mimic the resource usage of the actual application with much simpler shadows. We also describe how dependencies between the components are enforced.

4.3.1 Resource Mimic’ing Replay

**Computation:** How to faithfully replay the computation done between two trace events? There are many potential characteristics— instruction mix, memory access pattern, and instruction-level parallelism— that affect the runtime in the new environment. Collecting these requires instruction-level instrumentation which is expensive. For instance, just tracing the instruction-level parallelism slows down an application by over 600X [33]. Reproducing these characteristics with simple code is also hard.

To understand the problem better, we ran the industry standard SPEC CPU2006 benchmark suite [41] on nine server processors, including those used at three popular public
Figure 4.4: Example of the linear relationship between the benchmark running times on two different processors (AMD Opteron 2374HE vs. Intel Xeon E5507). There is one point per benchmark. The dashed line shows the best linear fit.

Table 4.2: The processor models we observed inside three cloud platforms (AWS, Rackspace, and Joyent). We found these models by launching 10 instances for each instance type offered by the clouds.

<table>
<thead>
<tr>
<th>Vendor</th>
<th>Model</th>
<th>Micro-arch</th>
<th>Year</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intel</td>
<td>Xeon E5430</td>
<td>Penryn</td>
<td>2007</td>
</tr>
<tr>
<td>Intel</td>
<td>Xeon E5507</td>
<td>Nehalem</td>
<td>2009</td>
</tr>
<tr>
<td>Intel</td>
<td>Xeon E5410</td>
<td>Penryn</td>
<td>2007</td>
</tr>
<tr>
<td>Intel</td>
<td>Xeon E5507</td>
<td>Nehalem</td>
<td>2010</td>
</tr>
<tr>
<td>Intel</td>
<td>Xeon X5550</td>
<td>Nehalem</td>
<td>2009</td>
</tr>
<tr>
<td>AMD</td>
<td>Opteron 2374</td>
<td>Shanghai</td>
<td>2009</td>
</tr>
<tr>
<td>Intel</td>
<td>Core i5 750</td>
<td>Nehalem</td>
<td>2009</td>
</tr>
<tr>
<td>Intel</td>
<td>Core i5 760</td>
<td>Nehalem</td>
<td>2010</td>
</tr>
<tr>
<td>Intel</td>
<td>Xeon X3440</td>
<td>Nehalem</td>
<td>2009</td>
</tr>
</tbody>
</table>

cloud (AWS, Rackspace, and Joyent). The suite includes a number of benchmark applications with diverse workloads (e.g., compression, compiling, multimedia en-/decoding, and floating point computation). Detailed information of the processors can be found in Table 4.2 (obtained from /proc/cpuinfo). As can be seen, these CPUs vary in frequencies,
vendors, architectures (Intel’s Core and Nehalem, AMD’s Shanghai), cache configurations, and the year of manufacture ranges from 2007 to 2010.

Figure 4.4 shows the results for a processor pair. It compares the runtime of the SPEC benchmarks between Intel’s Xeon E5507 (Nehalem) and AMD’s Opteron 2374HE (Shanghai). But for two outliers, the runtime of the various benchmarks conforms to a linear fit. This observation holds for all the processor pairs tested— the smallest pearson correlation coefficient was 0.95.

But why should such a linear fit exist? We have two conjectures. First, unlike highly optimized code, general purpose code tends to rely less on the esoterics of processor architecture. Second, that competition has led to significant uniformity across architectures.

CloudProphet exploits this linear fit. Given the runtime on one processor, it predicts the runtime on another processor by (linearly) scaling with the appropriate factor. The shadow runs in a busy loop for this duration. We note that while we could create adversarial workload (low cache hit rate and large working set) on which this simple approach yields poor predictions, we are surprised that the approach works for real apps (see §4.5).

Storage: CloudProphet mimic’s the use of two types of storage – raw storage, by which we mean accesses to files and blob, and managed database services offered by popular clouds, such as Amazon’s Relational Database Service (RDS) [6] and Microsoft’s SQL Azure [34]. We ignore a few others that are less commonly used by migrated legacy apps (key-value [8] and queue storage [7]).

Raw storage is relatively easy. By intercepting relevant system calls (e.g., read, write, fsync), CloudProphet records the amount of data read or written and the context such as the file name and the position of the file pointer. The actual content is silently dropped. Shadows mimic the calls but with fake content. CloudProphet performs bookkeeping when necessary, e.g., opening files.

As before, there is adversarial workload that circumvents this approach, e.g., opera-
tions on memory mapped files. We do not know how to trace such operations cheaply; raising a page fault upon every access to a mapped page incurs high tracing overhead. In our evaluated applications (§4.5), we find that memory-mapped files are small and fit in memory. Therefore, missing those I/O operations did not introduce significant error.

**Managed Databases:** It is a trend for cloud providers to offer managed database services since the regular VMs on offer cannot satisfy the high availability requirements of databases. To resource mimic the accesses to these databases, CloudProphet special-cases all database related system calls during tracing. Prior to replay, CloudProphet migrates the application data to the chosen database service. During replay, the shadows make the same queries. For data privacy, we note that the actual content in the database does not matter and can be anonymized as long as the schemas and the data properties, such as the key distributions, that impact performance are unchanged.

**Network:** By tracing network system calls, CloudProphet records the size and the four-tuple \((\text{src addr}, \text{src port}, \text{dest addr}, \text{dest port})\) identifying the connection for each network transfer. We show later how this helps to infer the dependency between network messages and handlers (§4.3.2). CloudProphet also records socket-level parameters that may be set by the application (e.g., via \texttt{setsockopt} calls) since some of these options, such as TCP\_NODELAY, can impact network performance considerably [107]. Shadows replay the same send/recv calls, and perform \texttt{setsockopt} when needed.

Due to the state maintained in TCP and the network stack, which connection a transfer uses impacts performance. For instance, a transfer sending through a connection whose window size has been reset due to idleness would take longer than sending through a “hot” connection. Moreover, since TCP provides reliable in-order delivery, network losses in an earlier transfer can delay when the later transfers in that connection are delivered to the application.

CloudProphet currently supports two policies—“no re-use”, in which each transfer uses
a new connection and “no overlapping”, in which a transfer can use any of the idle connections towards the destination/port. We empirically found these to be the most common patterns though others can be built as well (e.g., having some special affinity between connections and transfers). As with event ordering, we note that the choice of connection to use for a transfer, need not be the same as observed in the trace, because when a connection becomes idle depends on environmental factors such as network latencies. CloudProphet leverages patterns, such as worker-dispatcher, to identify dependencies.

4.3.2 Inter-component Dependencies

So far, we have looked at mimicking individual resource use effectively. That is, the shadows replay events in the trace and make the corresponding resource demands. Now, we focus on issues that impact the order in which events occur in the new environment.

Locks

Tracing locks is important, because they encode dependency between work, and the lock blocking time can affect performance. Literally replaying the order in which the threads gain (or block at) a lock would not work for reasons similar to those described in §4.2.1—in the new environment work may take different amounts of time, causing different thread orderings through locks.

CloudProphet explicitly traces the synchronous lock primitives provided by common libraries such as pthread locks, file locks, and Java monitors. For each lock operation, CloudProphet records its type (lock or unlock), the lock type (shared or exclusive), and a unique identifier for the lock (e.g., its memory address). During replay, the shadows issue the same lock operations against pre-initialized lock objects. For simplicity, we ignore the performance difference across different lock implementations, and use pthread locks to replay all of them.

Some applications optimized for high concurrency conditions (e.g., databases) use
low-level synchronization primitives to implement more complex locking schemes. For instance, an application might use the simple pthread locks to implement a read-write lock with priority for writers. To reproduce the behavior of such locks, it is insufficient to only trace and replay the primitives, because they do not encode the actual locking logic (e.g., the priority queue in the read-write lock). Instead, we need to annotate the application’s lock library, and enforce the same locking logic during replay. Although this is doable, because such locks are rare, CloudProphet ignores them for simplicity.

**Applying dispatcher-worker pattern**

Recall the challenge in §4.2.1 which requires knowing dependencies between the work (events) in order to replay them well. CloudProphet makes the simplifying assumption that most of the hardness in dependencies comes due to the dispatcher-worker pattern. A dispatcher issues *requests* to workers who in the course of processing the request can make calls to other components across the network, invoke storage, or interact with the database. All of these latter dependencies can be captured easily based on the ID of the worker thread and the time-period between receiving the request and completing the response. This leaves us with two main challenges: 1) How to identify the time boundaries when a worker is serving a request? 2) How to emulate the dispatching of work? We tackle these in order.

**Separating the work due to each request:** By definition of the dispatcher-worker pattern, we know that a request conceptually starts when it is passed by the dispatcher to one of the idle workers. However, the dispatcher might pass the requests in different ways, which require different considerations. We describe three popular cases in the following and present our solution for each of them.

In the first case, the dispatcher only listens for new connections. When a connection is established, the dispatcher immediately passes it to one idle worker thread, which will then serve all the requests sent through that connection. Many traditional multi-process/thread servers follow this pattern, such as Apache running in the prefork or worker mode,
Tomcat, Java RMI, PHP fastcgi, and Subversion [11]. Such pattern is easy to implement, but it can cause a waste of memory if there are many idle connections, because at least one worker must exist for every live connection.

We use inference to extract the work for each request. The idea is to exploit the pattern of the network events on the worker threads. Here we only look at the network events triggered by clients, but not those between the server and the backend. For each thread, an uninterrupted sequence of \texttt{recv}s followed by an uninterrupted sequence of \texttt{send}s is identified as a request-response pair. This is the common case. The thread first uses a \texttt{recv} loop to obtain the entire request body, does some work, and then returns the response with a \texttt{send} loop. All events between the request and the response are treated as the work triggered by the request.

In some other cases, there may be no response, which can happen for some fire-and-forget requests. To extract such requests, we also end a request-response pair when the same thread receives a message after a long lag (time $T$) from the previous receive. There is a trade-off in choosing the right $T$: if $T$ is too small (e.g., less than one round-trip-time), the \texttt{recv}s that belong to the same request would be erroneously inferred as a number of individual fire-and-forget requests; if $T$ is too large, some lightweight fire-and-forget requests would be inferred as part of their following requests. In our experiments, we found that, unless the network is badly congested, the time gaps between the \texttt{recv}s of the same request are usually smaller than one round-trip-time (RTT), consistent with previous observations [96]. On the other hand, the processing time of a request is usually much larger (more than 10 folds) than the RTT within a data center. Therefore, we choose $T$ to be $4X$ of the expected RTT.

This inference is not perfect and leads to error in some cases. For instance, if the request-response is interrupted, that is the worker receives more data from the client in the middle of sending back the response, it will be identified as two request-response pairs that are unrelated to each other. This can happen when a request is long and needs to be
processed in multiple stages (e.g., an acknowledgement is immediately returned by the server after receiving the request header). In some cases (e.g., due to TCP retransmission timeout [89]), requests that arrive with significant inter-arrival gaps (> T) will also be considered to be separate and independent. These inference errors are rare for normal RPC-like applications under good network conditions. They are also unlikely to impact performance estimation much, because we still replay all the work between each inferred request-response pair. Hence we ignore them to keep the solution simple.

In the second case, the dispatcher uses an I/O multiplexing primitive (e.g., select or poll) to listen for new requests from both existing and new connections. When a new request arrives, it passes the parent connection to one idle worker thread, which will then process the request, send the response, and return the connection back to the dispatcher. Some newer web servers use this pattern, such as Apache under the experimental event mode and the Java NIO-supported Jetty [26] and Glassfish [19]. Because this pattern does not require one worker thread per connection, it can save memory when there are many concurrent but idle connections. The inference method is still applicable here, because the worker threads still perform the network calls to receive/send the request/response.

In the third case, the dispatcher reads the incoming requests from the connections, stores them in memory, and uses IPC (e.g., a producer-consumer queue) to pass the requests to workers. Some applications with short requests follow this pattern, such as Hadoop [10]’s RPC server. This is the most complicated case, because the worker threads do not directly perform the network calls to receive the requests, and we cannot use the calls to infer the work boundaries. Our solution is to explicitly instrument the IPC mechanism used to pass the requests. We require the application to be modified such that a unique ID is attached to a request when it is passed from the dispatcher to a worker thread. In addition, both the work boundaries of the request and its network calls must be annotated with the ID. From our experience, adding such instrumentation only requires a few lines of code change, because the requests are typically well packaged into objects and
while true do 
  fd = poll(fd_set)
  if is_connection_new(fd) then 
    add_fd(fd_set, fd)
  else if is_connection_closed(fd) then 
    remove_fd(fd_set, fd)
  else 
    req_ID = recv_id(fd)
    enqueue(req_ID)
  end if 
end while 

FIGURE 4.5: The pseudo-code of CloudProphet’s dispatcher.

structures, and the work boundaries are usually marked by special handler functions (e.g., Server.call() in Hadoop RPC). We believe this is a reasonable price to pay to cover a broader set of applications.

Emulating Dispatchers: CloudProphet implements its own dispatcher to emulate the behavior of the real application dispatcher. Figure 4.5 shows the pseudo-code of the dispatcher. The algorithm is similar to the second dispatching pattern described above. The dispatcher thread listens for incoming requests from a set of connections. When a request arrives, it first extracts a request ID from the connection, which is inserted by the client to uniquely identify the request, and then adds the ID into a queue. The worker threads repeatedly poll request IDs from the queue (not shown in the pseudo-code), and replay the events to serve the requests.

Two tunable parameters are available to make the dispatcher’s behavior better match that of the real dispatcher. The first is the maximum number of worker threads, which limits the maximum number of concurrent requests can be replayed. This is akin to real application parameters such as ThreadLimit in Apache and maxThreads in Tomcat. The second is the maximum request ID queue length, which controls how much burstiness the application can tolerate. With these parameters, a customer can test her application’s performance under different settings to find the best one.
4.4 Implementation

We have implemented a Linux prototype of CloudProphet, which includes three main components: tracing engine, trace analyzer, and replayer. In addition, CloudProphet also comes with a benchmark tool to estimate the appropriate scaling factor for computation time (§4.3.1). The total lines of code are around 9.5K. In the following we describe the implementation details of each component.

**Tracing engine** The tracing engine is implemented as a lightweight library call interceptor through LD_PRELOAD. It transparently records the resource usage events from all application threads and write them to disk. Each recorded event contains four fields: a thread ID, a start timestamp, an event type, and a data field whose semantics depend on the event type.

We first intercept the libc wrappers for the I/O system calls to trace the I/O events. When intercepting these calls, the tracing engine also obtains the call context through helper functions. For instance, for a network I/O call, it would call `getsockname` over the intercepted file descriptor to obtain the local address and port. To collect database events, the engine intercepts library calls inside MySQL’s native driver `libmysqlclient`. Currently, CloudProphet only supports MySQL, but it is easy to extend CloudProphet to other databases. Finally, we intercept the pthread lock, the file lock, and the Java monitor functions to trace the lock events. Note that the Java monitor functions are not strictly library calls: they are Java bytecode instructions. Therefore, we use a bytecode binary instrumentation technique [13] to intercept these Java monitor functions.

To capture computation events, the tracing engine also measures the running time of a thread between two non-computation events. This is done through the system call `clock_gettime`, which returns the virtual time of the current thread with high precision.

To reduce tracing overhead, we also build an efficient, thread-safe, library to log the events to disk. The library uses per-thread event buffers to avoid global locks, and has
a background thread to flush the filled buffers to disk asynchronously. We also keep all event buffers in shared memory to minimize copies.

**Trace analyzer** The trace analyzer is an offline program that takes the per-thread event traces as input and outputs the events for each request-response pair, following the algorithm in §4.3.2. The trace is stored as one file per component to simplify event distribution, as all replayers of the same component can be configured to use the same trace file. In addition, an index of the handlers based on their request IDs is added at the beginning of the file for efficient lookup.

**Replayer** The replayer is implemented as a lightweight multithreaded server, following the design in §4.3. It takes a processed trace file and a configuration file as input. The configuration file includes the connectivity information towards the other replayers and the replay options. The replayer also prefetches the future events to reduce event loading overhead.

To drive the replayers, CloudProphet also includes a client emulator that replays the requests sent by the real clients during tracing. To let a customer test her application under different inputs, we make the emulator very flexible. A customer can increase or decrease the request sending rate to test the application under different load levels. She can also group the requests into user sessions (with some external information) to emulate each user’s behavior more accurately. In the future, we plan to support emulating other interesting client behaviors, such as automatic load-balancing across multiple sites.

**Benchmark tool** In the current implementation of CloudProphet, we choose the SPEC CPU benchmarks to estimate the runtime scaling ratio. We run all benchmarks inside both the cloud and local machines, and compute the average runtime ratio. We also tested other benchmark suites (e.g., the Java-based computation benchmarks in CloudCmp), and found the results are similar.

When running the benchmarks in the cloud, one needs to be careful with time mea-
surement, because the runtime measured by typical `gettimeofday` is the wall clock time, which might not equal to the actual CPU time spent at the physical processors. This is because part of the measured time can be actually spent when our VM is suspended while other co-located VMs are scheduled to run on the same set of physical processors, a case called CPU core-sharing. The extra suspension time essentially makes the measured runtime non-deterministic, because how long a VM is suspended depends on a lot of variable factors, such as the scheduler algorithm, the workload level inside our VM, and the workload inside the co-located ones.

One straightforward way to address this is to exclude the suspension time from the measured runtime. However, this is difficult to do in practice because many hypervisors and guest OSes do not expose the suspension time information to the applications running inside a VM. CloudProphet, on the other hand, adopts a clever strategy to address the problem without measuring the suspension time. Our observation is that to replay a piece of workload one does not actually need the accurate CPU time spent by the workload; all we need is how many times a busy loop needs to execute so that it takes the same CPU time as the workload and is therefore “equivalent” in terms of computation load. CloudProphet measures the equivalent looping times for every benchmark by running the busy loop and the benchmark side-by-side on the same virtual processor. If a benchmark’s running time is much longer than each scheduling time slice (a few tens of milliseconds), we can reasonably assume that the benchmark and the busy loop take the same CPU time, regardless of how much time the virtual processor is suspended. CloudProphet then estimates the scaling factor directly from the local runtime to the looping times in the cloud. This also simplifies the replayer, which learns immediately from the trace and the scaling factor how many times a busy loop should run to replay a piece of workload.
4.5 Evaluation

We evaluate CloudProphet in three steps. First, we show the usefulness of CloudProphet with two realistic case studies. Next, to understand why it works, we evaluate CloudProphet’s accuracy over a variety of applications, and compare it with simpler alternative designs. Finally, we evaluate the overheads of using CloudProphet.

4.5.1 Experiment Settings

Application Table 4.3 summarizes the applications we choose to evaluate CloudProphet. The applications have diverse characteristics. They range from realistic web applications such as RUBiS and MediaWiki which have multiple components and consume different types of resource, to applications such as IOZone that stress one particular resource. They are also implemented in different technologies: some are native applications running directly on the bare-metal machines, while others require interpreters and/or just-in-time compilers.

About half of the applications are distributed, which means they involve at least a client and a server and conform to the dispatcher-worker pattern. RUBiS and MediaWiki further have three tiers on the server side: a web tier, an application tier, and a database tier. The web tier serves the incoming HTTP requests; the application tier executes business logic (RUBiS) or performs full-text search (MediaWiki); the database tier stores the application data. We further generate representative client requests to drive the distributed applications. For some applications that are originally designed as benchmarks, such as RUBiS, MediaWiki, and TPC-C, we generate requests using their official benchmark workload generators.

Cloud environments We evaluate CloudProphet inside three popular IaaS cloud platforms: Amazon AWS [9], Rackspace CloudServers [38], and Joyent [27]. We choose AWS and Rackspace because they are the largest and most well-known cloud platforms,
Table 4.3: The applications we use to evaluate CloudProphet and the main resource types they consume.

<table>
<thead>
<tr>
<th>App</th>
<th>Description</th>
<th>Resources used</th>
</tr>
</thead>
<tbody>
<tr>
<td>RUBiS [40]</td>
<td>J2EE auctioning app</td>
<td>cpu, net, db</td>
</tr>
<tr>
<td>MediaWiki [32]</td>
<td>PHP collaborative app</td>
<td>cpu, net, disk, db</td>
</tr>
<tr>
<td>HTTP file server</td>
<td>File serving website</td>
<td>net, disk</td>
</tr>
<tr>
<td>Subversion [11]</td>
<td>Source version control</td>
<td>cpu, net, disk, db</td>
</tr>
<tr>
<td>TPC-C [48]</td>
<td>OLTP app</td>
<td>db</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>App</th>
<th>Description</th>
<th>Resources used</th>
</tr>
</thead>
<tbody>
<tr>
<td>FFmpeg [17]</td>
<td>Video transcoding</td>
<td>cpu</td>
</tr>
<tr>
<td>DCraw [16]</td>
<td>Photo transcoding</td>
<td>cpu</td>
</tr>
<tr>
<td>gcrypt [46]</td>
<td>Cryptography suite</td>
<td>cpu</td>
</tr>
<tr>
<td>IOzone [21]</td>
<td>Disk I/O benchmark</td>
<td>disk</td>
</tr>
</tbody>
</table>

Table 4.4: The instance types used in our evaluation. We discovered two common types of CPUs used in the AWS instances, but only one CPU type for Joyent and Rackspace instances. The prices are quoted in March 2012.

<table>
<thead>
<tr>
<th>Cloud</th>
<th>Instance</th>
<th>CPU Model</th>
<th>Cores</th>
<th>Mem (GB)</th>
<th>Price $/hour</th>
</tr>
</thead>
<tbody>
<tr>
<td>AWS</td>
<td>small</td>
<td>Intel E5507</td>
<td>1</td>
<td>1.7</td>
<td>0.08</td>
</tr>
<tr>
<td></td>
<td>medium</td>
<td>Intel E5430</td>
<td>1</td>
<td>3.75</td>
<td>0.16</td>
</tr>
<tr>
<td></td>
<td>large</td>
<td></td>
<td>2</td>
<td>7.5</td>
<td>0.32</td>
</tr>
<tr>
<td>Rackspace</td>
<td>small</td>
<td>AMD 2374HE</td>
<td>4</td>
<td>2</td>
<td>0.12</td>
</tr>
<tr>
<td></td>
<td>medium</td>
<td></td>
<td>4</td>
<td>4</td>
<td>0.24</td>
</tr>
<tr>
<td></td>
<td>large</td>
<td></td>
<td>4</td>
<td>8</td>
<td>0.48</td>
</tr>
<tr>
<td>Joyent</td>
<td>small</td>
<td>Intel E5620</td>
<td>1</td>
<td>2</td>
<td>0.17</td>
</tr>
<tr>
<td></td>
<td>medium</td>
<td></td>
<td>1</td>
<td>4</td>
<td>0.24</td>
</tr>
<tr>
<td></td>
<td>large</td>
<td></td>
<td>2</td>
<td>8</td>
<td>0.36</td>
</tr>
</tbody>
</table>

and we choose Joyent because it is a relative new platform with comparable service tiers as AWS. Different from the previous chapter, we did not choose Microsoft’s Azure [35] because it does not offer a Linux VM option, which is required to run CloudProphet.

Each of the providers we choose offers a full spectrum of instance types with different performance and cost. Due to budget and time constraint, we cannot cover all of them.
Hence, we select three instance types from each provider in the low and medium price range, as shown in Table 4.4. We name the three instance types from each platform according to AWS’ convention as small, medium, and large. These instances have different configurations (CPU type, memory size, etc.) and prices. Further, according to existing measurement results [103], they also have different interference levels. Each provider also offers multiple data center locations, and by default we allocate the instances from its default location.

**On-premise environment** We collect all application traces in a homogeneous on-premise cluster with 11 machines. Each machine has a quad-core Intel Xeon X3210 CPU and 4GB of memory. The machines are connected with a 1Gbps LAN.

### 4.5.2 Metrics

We use CloudProphet to estimate both the performance and cost of an application. We consider two commonly used performance metrics: the end-to-end response time and the maximum application throughput. The end-to-end response time is defined as the time from when a request is sent by the end-user to when the response is received. For web applications, we also consider the maximum application throughput, which is defined as the number of requests the application can process per second without exceeding a response time threshold. We set the threshold to be two seconds according to a recent study on e-commerce applications [2].

To estimate an application’s performance in a target cloud environment, we first collect the traces while the application is serving representative workload inside our on-premise environment. We then replay the traces inside the target cloud environment with real instances. We further adjust the replay speed to obtain the response time at different incoming request rates and to measure when the replayers have reached their maximum throughput.

An application’s cost is computed as the total monetary cost incurred by running the
application inside a cloud environment while serving an expected workload. In the context of our applications, the cost includes three main parts: instance cost $C_I$, bandwidth cost $C_B$, and database service cost $C_{db}$. The instance cost equals to how many instances are needed times the price for each instance, that is $C_I = N \times P_I$. The bandwidth cost can be calculated as $C_B = f_b(T)$, where function $f_b()$ captures the cloud’s bandwidth pricing strategy (usually a step-function), and $T$ is the amount of traffic sent across the cloud’s boundary. Finally, the database cost $C_{db}$ is the hourly cost of the chosen database service tier.

Because a cloud deployment is scalable, a customer would want to know what is the smallest deployment and hence the minimum cost that can support her expected application throughput. We estimate these in the following steps. First, we use CloudProphet to estimate the maximum application throughput achieved per instance in each application tier. This can be done by intentionally under-provisioning the tier. Then, by dividing the expected throughput by the throughput per instance, we get the number of instances required per tier, and the sum of them over all tiers becomes the total number of instances $N$. Similarly, we use CloudProphet to benchmark the throughput of each database service tier, and find out the lowest tier that can support the expected workload. Further, we measure the cross-cloud traffic sent by the replayers, and scale it to obtain $T$ under the expected workload. Finally, we compute the overall cost by combining the estimated parameters ($e.g., N$ and $T$) with the cost-specific pricing information ($e.g., P_I$ and $f_b()$).

4.5.3 How Can CloudProphet Help?

In this section, we use two case studies to show how CloudProphet can help customers migrate applications into the cloud.
Table 4.5: Three ways to split the application over two data centers. The ratio corresponds to the portion of budget spent in each data center. The database component cannot be split, and always remains in Joyent’s cloud for policy reason. We use the small instance type in both data centers.

<table>
<thead>
<tr>
<th>Hybrid Case</th>
<th>AWS DC</th>
<th>Joyent DC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Balanced</td>
<td>1/2 web and app</td>
<td>1/2 web and app, db</td>
</tr>
<tr>
<td>Biased</td>
<td>2/3 web and app</td>
<td>1/3 web and app, db</td>
</tr>
<tr>
<td>Asymmetric</td>
<td>2/3 web, 1/3 app</td>
<td>1/3 web, 2/3 app, db</td>
</tr>
</tbody>
</table>

Choosing Best-Performing Cloud Environment

As described in Chapter 1, a customer who plans to migrate applications into the cloud often faces the dilemma of choosing from a wide range of candidate cloud environments. One typical scenario is the customer wants to pick the best-performing cloud environments under certain budget constraint. In this section, we study whether CloudProphet can help customers make the correct choice using the two realistic multi-tiered web applications, RUBiS and MediaWiki. As we will see later, both applications are complex and difficult to model or migrate.

We first select a set of candidate environments. We assume the customer has a fixed budget limit on instance and database, and selects the candidates from the various instance types and database services. The budget does not include bandwidth cost which depends on the workload. The candidates include the six instance types from AWS and Joyent in Table 3.3, and depending on the instance prices. To make the study more realistic, we choose a mid-range budget of $3.2/hr ($2304/month) which allows the customer to provision a a few high-end instances or many (up to 40) low-end instances. Further, in both clouds we choose the database service with four dedicated CPU cores (XL 15GB RDS instance in AWS and XL 8GB MySQL Appliance in Joyent). We did not include Rackspace in this study because the cloud imposes a stringent limit on how many instances one can allocate (maximum 50GB memory in total).
In addition to the environments where all components of an application are deployed inside the same data center, we also consider hybrid environments, where the components are split over two adjacent AWS and Joyent data centers. A customer might wish to look into hybrid environments, because they enable mix-and-matching the services from both clouds and can also improve the application’s availability. Table 4.5 shows the three hybrid candidates we consider with different split ratios. All splitting is done between two west
Figure 4.7: The real and estimated end-to-end response time for both applications. In comparison, we also show the time estimated with the modeling approach.

cost data centers with a 15ms inter-datacenter network latency. The ratios are chosen to cover several common scenarios in hybrid deployment: the balanced case is suitable when two clouds share similar characteristics; the biased case is suitable when one cloud is more powerful than the other; the asymmetric case might be useful if different application components work best in different clouds.

We first choose application throughput as the selection criterion. Figure 4.6 shows the
actual throughput (y-axis on the left) of running RUBiS and MediaWiki under different
candidate environments and those predicted by CloudProphet. In comparison, we try to
predict the best-performing cloud environment through benchmarking, similar to what we
did in §3.6. There are a number of benchmarks related to the applications, including CPU,
network, and database. We choose the CPU benchmark because it predicts the ranking
most accurately. In particular, we run the multi-threaded CPU benchmark in CloudCmp
(§3.4.1) on each instance with enough threads to saturate the CPUs and measure the ag-
gregate throughput across all instances, i.e., the total number of benchmark tasks finished
within a time period. We further normalize the benchmarking throughput to between 0 and
1 (y-axis on the right) because it is not directly comparable to the application throughput.

The figures show that CloudProphet accurately predicts the throughput for both appli-
cations under all candidates (with relative error < 16%). As a result, it can easily help
the customer pick the best candidate (AWS.S for RUBiS and Joyent.L for MediaWiki).
On the other hand, if the customer follows the benchmarking numbers, she would have
picked Joyent.L for both applications which offers only roughly half of the throughput un-
der the best instance type. We can further observe that the candidate ranking according to
the benchmark results is much different from the real one, while CloudProphet accurately
predicts the entire ranking.

The main reason behind the ranking differences between CloudProphet and bench-
marking is because of the unpredictability of performance bottlenecks under different
cloud environments. For example, in most Joyent instances the CPU utilization increases
significantly under heavy network activity, possibly due to an inefficient network driver
implementation. This interference creates a convoluted bottleneck that involves both CPU
and network, which cannot be captured by CPU benchmarking alone. It is also difficult
to come up with the right mix of CPU and network workloads to stress such bottleneck.
As another example, both the web and business tiers have high CPU utilization when run-
ning RUBiS in Joyent.L instances. When multiple bottlenecks exist, it is hard to combine
the benchmarking results of individual tiers into an aggregate number that reflects the end-to-end application performance. Finally, the extreme low throughput when running MediaWiki in the hybrid environments is caused by a new synchronization bottleneck inside the database. Because in MediaWiki each database transaction sent by a web server includes many queries, splitting the web server from the database significantly increases each transaction’s execution time, which in turns greatly reduces the transaction throughput. Without being able to predict such bottleneck shift, the CPU benchmarking result unavoidably deviates from the actual application performance, and it is very difficult to capture such application-specific bottleneck. In contrast, the trace-and-replay approach of CloudProphet can naturally capture the real bottlenecks of an application running under different cloud environments.

Secondly, we look at the candidate ranking by end-to-end response time. We assume the customer has an expected workload in mind, and wants to find the candidate with the lowest end-to-end response time. The expected workload is chosen to be 1400 reqs/s for RUBiS and 50 reqs/s for MediaWiki, about half of the maximum throughput averaged over all non-hybrid candidates. Figure 4.7 shows the real and predicted response times. We omit the results of MediaWiki under the hybrid cases because the application cannot handle the test workload for the reason we described above. In comparison, we also estimate the response times through a modeling approach [95]. The approach models each resource at every application tier as an M/M/1 queue, and sums both the regular service time (including network latency) and the queue waiting times to estimate the total response time. It further estimates the per tier service time in the new environment through a simple CPU model that is similar to the one we used to scale the computation time (§4.3.1).

From Figure 4.7 we can find, again, CloudProphet attains high prediction accuracy. In contrast, the modeling approach significantly under-estimates the response times, and is only accurate for the lightly loaded AWS candidates. The main underlying reason is that the model over-simplifies both the applications and the environments. For instance, in
the Joyent cloud, the model fails to capture the convoluted bottleneck described above, and hence under-estimates the utilization of the CPU resource. For MediaWiki, the model estimates the resource utilizations with reasonable accuracy, but its M/M/1 queueing assumption does not fit the application. This is because each application tier includes multiple servers and queues instead of just one, and the service time at each tier is not exponentially distributed. More complex models targeted specifically at the clouds/applications would probably work better, but building such models can take significant expert effort. In contrast, CloudProphet does not abstract away the critical application and environment characteristics.

Estimate Cost in Cloud

Now we look at another important use case of CloudProphet, that is to help customers estimate how much their applications would cost if hosted in cloud. We first apply the CloudProphet-based approach described in § 4.5.2. In comparison, we also obtain the real application cost in cloud by searching for the cheapest cloud deployment that can satisfy the expected application throughput.

Figure 4.8 shows the estimated and real costs of the two applications in different candidate environments. We choose the expected workload to be half of the averaged maximum throughput, the same as used in the response time estimation. We found that the CloudProphet-based approach can accurately estimate the cost over all cases, and the customer can easily compare the cost-effectiveness of different cloud environments. One can also notice that the Joyent instance types are more cost effective than the AWS ones, despite being less efficient. This is because the overall cost of an application depends on many factors. For instance, the AWS instances are more efficient and therefore require less cost. However, Joyent does not charge for bandwidth for the first 20TB transfer every month, while AWS charges around 12cents/GB for outgoing traffic from the data centers. The additional bandwidth charge offsets the cost saving from instances.
Without CloudProphet, such accurate cost estimation will only be possible if we migrate the application to the target cloud environment and calibrate the model parameters through benchmarking. With CloudProphet, because the replay mimics the real application behavior, we can calibrate the model parameters such as the maximum throughput per instance through replaying rather than through benchmarking the actual application. Further, as the replay consumes the same amount of resource as the real application, it also
incurs the same resource usage cost as the real application, such as bandwidth cost.

### 4.5.4 Why CloudProphet Works?

In the previous section we have shown case studies where CloudProphet can help customers pick the best-performing cloud environment and accurately estimate their applications’ cost. In this section, we use controlled experiments to gain insight on why CloudProphet works. We first evaluate its accuracy with a variety of applications, and then study our design choices in dependency extraction and computation scaling.

**CloudProphet Accuracy**

In the case studies, we primarily focus on the estimation accuracy of the two web applications RUBiS and MediaWiki inside two clouds AWS and Joyent. In this section, we expand our focus to the rest of the applications listed in Table 4.3 and all three clouds in Table 4.4. Figure 4.9 shows the real and estimated response time (or running time, if the application is not distributed) of the applications in each cloud instance type. For the database-intensive TPC-C, we test it over the different database services available inside AWS. For AWS which uses two different CPU models (Intel Xeon E5507 and E5430) for the instances, we cover both of them by selecting E5507 for AWS.S and AWS.M and E5430 for AWS.L.

The figure shows that CloudProphet’s estimation accuracy is high throughout all combinations of instance type and application. The largest relative error is 17%. This finding is surprising because the applications have very different workload characteristics: some are bottlenecked by CPUs, while the others are bottlenecked by storage or network. This suggests that CloudProphet’s accuracy is not sensitive to application type and workload.

There are several interesting observations from the results. First, we found that the simple scaling approach we applied to the computation time between events (§4.3.1) works very well for practical CPU-intensive applications such as FFmpeg, DCraw, and gcrypt.
Figure 4.9: The real and estimated response times of the applications we evaluate. The times are averaged over 10 runs and normalized to fit between 0 and 1.
Figure 4.10: The real and estimated response time distribution for the custom lock benchmark with log4j. The lock blocking time of each request during the replay run is also shown. To focus on lock contention, we run the experiments on a 8-core instance type c1.xlarge offered by AWS.

This again confirms our observation that a linear runtime relationship is likely to exist for general purpose applications.

Second, we found that CloudProphet can reproduce the storage caching effect, both for file system and database query caches. This is because the system replays the disk I/O calls and database queries under the same context as the original calls (same file and location, same query content).

Third, all applications we tested are not bottlenecked by locking. This is probably because they are well optimized to reduce lock contention. To evaluate how well CloudProphet reproduces lock contention, we specifically build a simple web application where each request uses the log4j logging library [31] to write a large log message. log4j uses Java monitors to protect the log file and avoid corruption. Figure 4.10 shows the real and estimated response time distributions when we send 30 logging requests per second. As can be seen, lock contention occurs for around 40% of the requests, because the lock blocking time (and the response time) starts to increase at the 60th percentile point. CloudProphet can accurately reproduce the contention effect, as the distribution of the predicted
response time closely matches that of the real one.

**Impact of Inter-Component Dependency**

A significant part of CloudProphet focuses on how to extract and enforce the right inter-component dependencies. In this section, we study how important it is to follow the correct dependencies. We compare CloudProphet with a naive approach that simply replays the traces thread-by-thread, which is adopted by a number of existing trace replay work such as [83, 76, 111].

Figure 4.11 shows the response time estimation results for RUBiS and MediaWiki.

![Graph](image-url)

**Figure 4.11**: Comparison between CloudProphet and the naive approach that replays the traces thread-by-thread. We run the experiments under different workload levels.
Both applications involve many inter-component dependencies between tiers. To exclude any inaccuracy caused by resource contention, the workloads we use are well below the maximum throughputs. As can be seen, the response times estimated by the naive approach are much greater than the real ones. We found this is because the applications frequently block at false intra-thread dependencies, as illustrated in Figure 4.12. Further, the estimation error increases with the workload level, because higher workload reduces the gap between requests, which in turn causes more out-of-order arrivals. In contrast, CloudProphet can break the false dependencies leveraging the dispatcher-worker pattern. As long as the applications follow the pattern, CloudProphet can extract and enforce the correct inter-component dependencies, which further results in high estimation accuracy.

**Impact of CPU Core Sharing**

In §4.4 we describe the practical challenge and solution on handling CPU core sharing in the cloud. In this section, we study the importance of the problem and the effectiveness of our solution. We first implement a naive system that is completely oblivious to virtualiza-
Figure 4.13: Comparison between CloudProphet and the naive approach that treats virtual cores as physical ones. We run the experiments under different workload levels.

The naive approach and treats every VM processor as a physical one. It simply computes the computation time scaling factor using the wall clock running time of the benchmarks, and uses busy loops to consume the scaled computation times in the cloud. We then compare it with CloudProphet inside AWS’ small instances, where each virtual CPU core only gets less than 50% share of a physical one.

Figure 4.13 shows the results for RUBiS and MediaWiki. The response times estimated by the naive approach is much greater than the real ones. We found that the estimation error comes from the workload-dependent nature of a virtual core’s speed. For instance,
under the latest Credit scheduler of Xen [61], if a VM’s workload is low, its virtual core may never run out of credit, and can always run as fast as the underlying physical core; however, when the workload is high, the virtual core is more likely to be preempted by co-located ones, causing it to slow down. Because the benchmarks are highly CPU-intensive, the naive system (falsely) concludes that the virtual core is always very slow. It then overscales the computation times of the applications, because they are not as CPU-intensive as the benchmarks. The over-scaled computation times increase the response times and further introduce higher resource contention.

CloudProphet does not suffer from this problem, because the mechanism we use to measure the equivalent “looping times” of a benchmark is independent of any CPU core sharing policy. In fact, we found that the looping times of a benchmark only depend on the model of the underlying physical processor. The technique might be generalized to other scenarios, such as application profiling in a virtualized environment.

4.5.5 Overhead

Finally, we evaluate the computation and storage overhead of CloudProphet while collecting the trace, and compare its deployment overhead with the migration overhead of the real application.

Computation Overhead

To evaluate the computation overhead of CloudProphet, we measure the slowdown ratio of the application when tracing is enabled. Figure 4.14 shows the results. We found that the slowdown ratio is close to 1 in all cases, which means the computation overhead is low and the tracing engine has little impact to the application performance. This is because CloudProphet does not collect very fine-grained events that require frequent instrumentation. Furthermore, the asynchronous tracing engine can effectively hide the trace-writing overhead by overlapping it with the application’s other non-I/O workload.
We measure both the memory overhead of the tracing engine and the persistent storage overhead to store the traces. The memory footprint of the tracing engine is around 2MB per application thread (for the event buffer) plus a global bookkeeping overhead less than 20MB. Figure 4.15 shows the average trace size per second per application thread when the thread is actively running. The network and disk-intensive applications, such as RUBiS and IOzone, generate more events which result in larger traces. We believe such overhead is acceptable, because performance prediction typically does not require a very long trace (*e.g.*, several hours is usually good enough), in which case the trace size is on the order of several GBs to tens of GBs. Further, the incoming bandwidth of a cloud is usually free [50, 27] to encourage migration. Therefore, CloudProphet will not incur any additional cost for trace uploading.
Figure 4.15: The average trace size generated per second by an actively running application thread.

Figure 4.16: The comparison of the migration overhead of the real applications and the deployment overhead of CloudProphet. We use CMP points [99] as our metric.

Deployment Overhead

It is difficult to quantify the effort spent in deploying an application in the cloud. Nevertheless, we can still try to quantify the complexity of the deployment/migration project
by examining the tasks one needs to perform. Here we leverage an existing model called Cloud Migration Point (CMP) [99] that assigns empirically determined CMP points to each type of task to quantify its complexity. We choose the model because it is based on the famous Function Points (FP) model in software engineering [53], and statistics from real migration projects suggest that the metric correlates well with the person-hours needed to perform the migration.

We calculate the CMP points following the rules in [99] for deploying CloudProphet and migrating the applications. Figure 4.16 shows the results. We find that overall the complexity of deploying CloudProphet is only 30% of the complexity of migrating the actual application. We also break down the CMP points into several categories. Deploying CloudProphet has similar network and database migration complexity as migrating the applications. Such complexity is low, because we only need to change the endpoint addresses and copy the database content without changing its schema. On the other hand, CloudProphet has much lower complexity in software installation and configuration. This is because each application has many dependent software packages with non-trivial configuration, while CloudProphet only requires one package (the replayer) with minimum configuration. Further, the complexity of deploying CloudProphet does not depend on the complexity of the actual application, and is only determined by the number of components in the application.

4.6 Summary

Due to the diverse performance and cost characteristics of cloud platforms, a tool that easily estimates the behavior (performance, cost, scaling) of an application in a candidate cloud environment can be very useful in practice. Such tool can help application owners choose, from among the plethora of available cloud platforms, the one that is best suited for their application. It can also help application architects quickly examine the trade-offs
with various design choices.

In this chapter, we presented CloudProphet, a system that achieves this goal by replaying the work with application-specific shadow programs. CloudProphet generates these shadows by tracing the application, ensures that the shadows exercise the same resources as the original application, and leverages the popular dispatcher-worker pattern to preserve dependencies across components. Shadows are simple code that can trivially be ported across environments.

We demonstrated the usefulness of CloudProphet in helping customers migrate their applications to the cloud and show CloudProphet to be accurate for a variety of applications. While CloudProphet achieves the stated goal – predicting behavior in a new environment– for several real applications, much work remains to be done, however, in designing a general solution to this problem.
5

Future Work

In this chapter we discuss the ongoing and future work to improve CloudCmp and Cloud-Prophet.

5.1 Temporal Comparison of Cloud Providers

In chapter 3, we use the CloudCmp tool to launch a spatial comparison study across multiple cloud providers during the same period of time. On the other hand, it is also interesting to compare a cloud provider temporarily with itself over a relatively long period (e.g., a few years). Such study can offer us insights on how public cloud computing platforms evolve. In particular, it can help us answer the following questions:

- How frequently does a cloud provider update its infrastructure?
- Does an application’s performance bottleneck in cloud changes over time?
- Does the increased popularity of a cloud introduce higher interference and cause performance degradation?
- Do the prices of different cloud platforms reach a market equilibrium over time?
• How available are the different cloud services?

To conduct such a study, we can simply run the CloudCmp tool continuously inside the chosen cloud providers. Because the benchmark workload of CloudCmp is light, we can probably utilize the free service tiers to save cost.

5.2 Generalize CloudProphet

In its current form, CloudProphet works with applications implemented following the popular dispatcher-worker pattern. Although it can already support a variety of realistic applications, there are still opportunities to generalize CloudProphet to cover a broader set. Here we identify two possible directions.

5.2.1 Cover Other Popular Patterns

The most straightforward way to extend CloudProphet is to cover other popular programming patterns, such as the event-driven pattern [86] and the SEDA pattern [105]. Arguably, these patterns are more complex than the dispatcher-worker pattern, because they involve new inter-component dependencies that are difficult to track down. For instance, in an event-driven application, two threads can synchronize with each other through a custom implemented event queue in memory. How to extract and enforce such new inter-component dependencies is a key challenge to address.

Based on our experience, we identify two potential solutions. The first is to leverage a few popular implementation frameworks such as libevent [29] and Java NIO [25]. These frameworks have well-defined interfaces, which can be instrumented to extract the dependencies. Such approach does not require any changes to the applications, but it only applies to applications using those frameworks. Another more general approach is to design a set of annotation primitives for developers to explicitly annotate the dependencies, similar in spirit to X-trace [65] and Magpie [75]. Such annotations essentially decouple an
application’s dependencies from its workload and other logics. Although annotating the code imposes additional overhead, we conjecture it might be doable due to good modularity in production code. Furthermore, such annotations can also be used for other purposes beyond performance prediction, such as debugging and resource usage analysis.

5.2.2 Handle Adaptive Applications

Adaptive applications that change their workload based on the resource availability inside a new cloud environment cannot be handled by the current trace replay technique. This is because the traces collected in one environment might not capture the application’s behavior in another one. It is in general very difficult to predict the behavior of any adaptive application, because it can easily defeat a prediction mechanism by behaving randomly or maliciously.

We have some early thoughts on how to handle applications that are adaptive but benign. The high-level idea is to mimic the behavior-changing factors in the local environment to observe how the application would behave. For instance, if an application’s behavior depends on the available memory size, we can feed the same memory size in the cloud to the application while running in the local environment to trick the application into believing it is running in the cloud. This is akin to deterministic replay [88, 71], with the difference that we do not need to control all non-determinism.

5.3 Efficient Search of Cloud Environments

While designing CloudProphet, we focus on estimating the performance and cost of an application inside a number of candidate cloud environments chosen by the customer. To enable a customer to automatically identify the best cloud environment for her application, one piece is still missing, that is how to efficiently search through the space of all potential environments. Due to the mix-and-matching of different cloud services in the same or different cloud platforms, there can be a combinatorial number of choices, and it is
impractical to require the customer to test every case even with the help of CloudProphet.

To address this, we propose to build an efficient search framework based on CloudProphet. The framework would use CloudProphet as a drop-in replacement of the real application to accurately estimate the application’s behavior in a few representative cases, and then use analytical models to deduce in other cases. For instance, the framework can use the simple model described in §4.5.2 to deduce how many instances are required for an expected workload.
Cloud computing is redefining how we perform computation. It has never been so easy to have access to so much computation resource in a short time frame with a pay-as-you-go billing model. As a result, application owners from large enterprises to startups and small businesses are all taking this opportunity to trim down their management and operation cost by migrating their applications “into the cloud”. However, the sheer diversity of the cloud environments and the complexity of the applications has made it very challenging to choose the right environment for an application. Although ad-hoc solutions exist, they are insufficient for the emerging cloud applications that are highly sensitive to performance, scalability, and cost.

This dissertation has made two important contributions towards systematic and accurate cloud environment selection. The first is to build tools that systematically compare different cloud providers. Such tools can help application owners better understand the strengths and weaknesses of each cloud platform. They can also help cloud providers and researchers to gain insight on the performance and cost characteristics of real clouds. The second is to build systems that accurately estimate an application’s performance in a cloud. This enables an owner to first assess her application’s performance in a number of
candidate environments before choosing the most suitable one.

6.1 Comparing Cloud Providers

We have built a novel toolset called CloudCmp that can compare the performance and cost of public cloud providers. The toolset supports both IaaS and PaaS providers, and covers the common cloud services including computation, storage, intra-cloud network, and wide-area network. While designing the toolset, we identified the comparison metrics that are relevant to application’s performance and cost, and proposed techniques to normalize the differences between cloud providers.

We also used CloudCmp to launch the first comprehensive measurement study of four popular public cloud providers: Amazon AWS, Microsoft Azure, Rackspace Cloud-Servers, and Google AppEngine. The measurement study lasted for two months (March to May 2010). The results revealed dramatic performance and cost differences across today’s clouds, without any cloud winning in all aspects. For instance, one cloud’s virtual instance can be twice as fast as that of another cloud while only being 30% more expensive. Moreover, a particular cloud provider’s instances are highly cost-effective, because they can fully utilize the underlying physical machines when there is no local resource competition. Finally, we showed that CloudCmp’s results are relevant to application performance by successfully predicting the best-performing cloud providers for three representative applications.

6.2 Estimating Application Behavior in the Cloud

We then built a system called CloudProphet that can accurately estimate an application’s behavior (performance, cost, and scalability) in a cloud environment without the need to port the application. Different from the existing modeling approaches, CloudProphet builds portable shadow programs that mimic the resource usage and dependencies of the
application components. It then deploys the shadows in the cloud to estimate the application performance.

In CloudProphet, we use the trace replay technique to automatically generate shadows for applications that conform to the popular dispatcher-worker programming pattern. A tracing engine first collects application events that encode the usage information of a number of important resource types such as CPU, disk, and network. Leveraging the programming pattern, CloudProphet then extracts the dependencies between the events. Finally, CloudProphet uses efficient replayers to replay the events in the cloud while enforcing the same dependencies.

In building CloudProphet, we have tackled several important design and implementation challenges, including how to decide what information to trace, how to extract the dependencies, how to trace at low overhead, and how to accurately replay computation workload in the face of CPU core sharing. We have demonstrated the usefulness of CloudProphet in helping application owners choose the most suitable cloud environments. Further, we have shown CloudProphet is accurate with regards to a variety of applications.
Bibliography


[59] Y. Chen and R. Sion. To cloud or not to cloud?: musings on costs and viability. In ACM SOCC, 2011.


107


Biography

Ang Li was born in Guiyang, China on July 30th, 1986. He defended his PhD thesis at Duke University in July 2012. His research interests include cloud computing, Internet routing, and network security. He received a B.S in Electronic Engineering from Tsinghua University in 2006.