Characterization of Cost Excess in Cloud Applications

by

Kevin Buell

A Dissertation Presented in Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy

Approved November 2012 by the
Graduate Supervisory Committee:

James Collofello, Chair
Hasan Davulcu
Timothy Lindquist
Arunabha Sen

ARIZONA STATE UNIVERSITY

December 2012

ABSTRACT

The pay-as-you-go economic model of cloud computing increases the visibility, traceability, and verifiability of software costs. Application developers must understand how their software uses resources when running in the cloud in order to stay within budgeted costs and/or produce expected profits. Cloud computing's unique economic model also leads naturally to an earn-as-you-go profit model for many cloud based applications. These applications can benefit from low level analyses for cost optimization and verification. Testing cloud applications to ensure they meet monetary cost objectives has not been well explored in the current literature.

When considering revenues and costs for cloud applications, the resource economic model can be scaled down to the transaction level in order to associate source code with costs incurred while running in the cloud. Both static and dynamic analysis techniques can be developed and applied to understand how and where cloud applications incur costs. Such analyses can help optimize (i.e. minimize) costs and verify that they stay within expected tolerances.

An adaptation of Worst Case Execution Time (WCET) analysis is presented here to statically determine worst case monetary costs of cloud applications. This analysis is used to produce an algorithm for determining control flow paths within an application that can exceed a given cost threshold. The corresponding results are used to identify path sections that contribute most to cost excess.

A hybrid approach for determining cost excesses is also presented that is comprised mostly of dynamic measurements but that also incorporates calculations that are based on the static analysis approach. This approach uses operational profiles to increase the precision and usefulness of the calculations.

DEDICATION

To Arabeth, who has put up with all of it, and still does.

ACKNOWLEDGEMENTS

I would first like to acknowledge the support, patience, and guidance of my advisor Jim Collofello. He is very skillful in knowing how hard to push, what is important and what is not, when to step out of the way and when to intervene, and in general how best to help. He was definitely the right advisor for me and this work would not have been possible without him.

I would also like to acknowledge my committee members–Tim Lindquist, Hasan Davulcu, and Arun Sen–for sacrificing their time to review and discuss my research. They offered several important suggestions and this work is better because of their help.

I cannot leave out Martha Vanderberg who is probably the most helpful person in public service that I've ever met. Her skillful and caring leadership in the computer science advising department at Arizona State was immensely helpful to ensure this work moved through the process toward completion.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Table 1: Key Terms

Key Term	**Brief Definition**
Cloud Application	Software targeted for deployment in a cloud computing environment. In particular, the cloud applications discussed here assume abstraction at the platform level (Platform as a Service or PaaS) and make use of outside software services (Software as a Service or SaaS).
Transaction	Usage of a cloud application for which the application provider charges a specific amount or for which a specific cost is budgeted.
Cost	Not just the usage of some resource but in particular the monetary cost incurred by running an application in the cloud.
Path	A series of statements in an application's source code that can be executed sequentially, including statements linked via procedure calls (an interprocedural control flow path). Some of the algorithms in this dissertation also consider intraprocedural paths.
Path Section	A subset of the statements in a path. For simplicity, entire methods or the contents of decision blocks are used to delineate path sections in this dissertation.
Characterization	The identification and description of cost excess. Paths and methods are shown to be cost excessive and are described in terms of their association with operational profiles and/or particularly important (costly) path sections.
Annotation	A note inserted into source code by an application developer that conveys some information to the algorithms in this dissertation which they cannot determine independently. The term is borrowed from the WCET literature where annotations are sometimes required for WCET systems to determine loop and recursion bounds.
Conceptual Model	A description of how cloud computing costs are scaled down to the transaction level and associated with source code. This forms the basis for calculations of maximum cost and cost excess.

Chapter 1

Introduction

1.1 Background

Traditionally, organizations providing remotely accessible software services have hosted software on in-house servers. Under these conditions, economics are generally considered at a relatively high level. Costs that must be accounted for up front include software development, infrastructure investment, third party software purchases or contracts, and bulk bandwidth service agreements. A certain level of revenue is forecast for a certain time period, and the organization attempts to recoup initial costs over time as well as pay for ongoing costs. Furthermore, system planning would account for various levels of resource usage by allocating infrastructure with slack in reserve.

Cloud computing obviates the need for application developers to concern themselves with the logistics and economics of large, upfront infrastructure investments [7]. Instead, cloud application developers pay only for the processing, bandwidth, and storage their applications require. Likewise, application developers may use other software services for which they pay a fee, according to their Service Level Agreement (SLA). It is not unlikely that this fee will take a pay-as-you-go approach similar to the cost structure of cloud infrastructure, realizing the vision of computing as a utility (see [15]).

Application developers using cloud infrastructure and services are not concerned with resource or system unavailability. Indeed, the promise of the cloud is that they need not be [7]. Cloud application developers may not be as concerned with the economics of large initial investments (though they are certainly not freed from recovering initial development costs).

Many cloud applications are offered in the form of Software as a Service (SaaS) and users are charged on a per use basis. For these types of cloud applications,

not only is this a pay-as-you-go cost model, but this revenue model could similarly be termed ‘collect-as-you-go’, which then leads to a profit model could be called ‘earn-as-you-go’. Understanding exactly how much the software costs as it runs is essential for these types of applications. The more willing cloud application developers are to understand the fine grain economic details of their software (i.e. at the level of transactions and ultimately source code costs), the more they will be able to maximize profit and/or minimize the costs they pass on to customers.

However, different paths in software may produce significantly different costs [12]. See figure 1.1 for an illustration of what various transactions in a software service might cost. Those that exceed the cost objective or threshold are marked with an ‘x’. The rate that customers are charged is also noted. This is a generic distribution that might represent any number of pay-per-use software services. This dissertation is particularly beneficial to transaction oriented applications and services, especially those that might aggregate other services.

For example, a transaction with stock trading middleware running as a service in the cloud might consist of the fulfillment of a single purchase of stock. The middleware provider might charge $0.10 for this transaction which must cover processing and bandwidth costs plus the cost of any outside services used to complete the stock purchase. Other possible examples include middleware for processing credit card purchases, online wire transfer and transaction software, and real estate research engines.

As low level economic concerns of cloud applications are considered, the following types of issues become particularly interesting:

- How much a transaction costs
  - How much do common transactions cost?

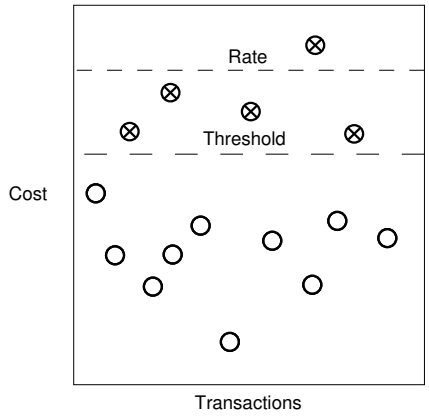

Figure 1.1: Cost of some sample transactions

  - How much do transaction costs vary?
  - What causes cost variation in transactions?

- How much to charge for a transaction
  - Fixed rate or variable?
  - Based on an average transaction cost, or closer to the max transaction cost?

Based on business case analysis and intuition, developers may have an idea of some of the answers to these questions. Furthermore, they may fix a rate that they charge customers for the different transactions available with their cloud application. When fixing this rate, they have in mind a cost threshold such that most of the transactions they service will cost less than the threshold. In this way, they expect to make a profit roughly equal to the rate they charge customers minus the cost threshold, multiplied by the number of transactions they expect to service.

During the verification phase of a software development process, developers test certain nonfunctional attributes of their software. One of these attributes that is especially visible and traceable in cloud computing is monetary cost. As part of a verification process for earn-as-you-go cloud applications, it would be desirable to ensure at least one of the following conditions holds:

- The cost of each transaction will always fall below the threshold
- Transactions that cost more than the threshold will be sufficiently rare
- The cost or frequency of transactions above the threshold can be reduced

Note that monetary costs in cloud applications probably do not have hard constraints. As in most business situations, developers can ensure they turn an overall profit even if they service some transactions at a loss, as long as most transactions cost somewhat less than developers charge for them.

The ultimate business goal for cloud applications is to either produce profit or stay at or below budgeted cost. This dissertation focuses on technologies that allow application developers to understand costs and in particular, to understand how and when costs fall outside of tolerances.

Other types of resource usage verification problems operate within hard limitations. For example, real time systems often require that certain execution times always stay within tolerances, even in the worst case [79]. The use of monetary resources is likely to have slightly softer constraints. Of course, overall costs should not exceed revenue, but some individual transactions may exceed the price charged for them. Note that in the context of this dissertation, a transaction is defined as a distinct usage of the cloud application, generally one for which an associated cost/price can be set. When the cloud application is used internally (not to derive direct revenue), a transaction is a usage for which a specific cost is budgeted.

Cloud application cost analysis is a somewhat novel research area not well represented in the literature to date. Two main branches of research immediately present themselves within this area: optimization and verification. Cost analysis for optimization is concerned with determining how cloud applications can perform their intended function more efficiently such that costs are minimized. It is possible that

some general optimizations can be made, and guidelines may be specified to address this problem [61]. However, optimization is likely to be most effective with intimate domain knowledge and application specific information.

Cost analysis for verification may require access to an application's source code but not necessarily an understanding of the application's functionality. This is similar to how memory management can be analyzed from a program's source code. These resource usage problems are concerned with where resources are used and to what extent but not whether resource usage fulfills promised functionality. The work developed in this dissertation falls within the general area of cost verification, while the problem of cost optimization is not directly addressed.

Cloud application costs can broadly be defined to fall into one of four categories: processing, storage, bandwidth, or services [48], [83]. Processing costs are all those items associated with the CPU. Storage costs are the items associated with persistent data. Bandwidth costs are incurred when sending or receiving data to or from outside entities. Services are other applications that are provided inputs and produce outputs required by the client application. The cost of a transaction with a cloud application is the cost of each of these individual parts for the transaction added together.

The amount of processing for a transaction is measured in terms of time. The clock time required for a transaction is determined and multiplied by the cost per nanosecond that the cloud provider charges for processing. It is also important to know the number of bytes of memory required for the duration of the transaction, or perhaps whether the transaction has average or high memory requirements.

The storage cost of a transaction is calculated based on the number of bytes stored, the cost per byte for interfacing with the storage medium (though current providers generally charge a flat fee for each interface [4]), the cost per second per

Table 1.1: Current Rates Charged by Major Cloud Providers

Provider	**CPU (hour)**	**Storage (GB/Month)**	**Bandwidth (GB)**
Amazon EC2 [4]	$0.08	$0.10	$0.12
Microsoft Azure [56]	$0.12	$0.125	$0.12
Google AppEngine [31]	$0.08	$0.13	$0.12

byte for persistent storage, and the amount of time the data stored during the transaction is retained on the storage medium.

Bandwidth costs are separated into incoming and outgoing transfers. The bandwidth cost calculation is simply the amount of bandwidth in bytes requested multiplied by the amount the vendor charges per byte. Sometimes ingoing and outgoing transfers are billed at different rates. See Table 1.1 for an example of current rates cloud providers charge for resource usage.

When developers use outside services they incur costs according to pricing established by the service vendors. Their total cost for using outside services for a transaction is simply the sum of all the individual transactions they make with outside services. (This, of course, assumes that they can determine which services are invoked and how many times, which will be discussed later in this dissertation.)

As a very brief example, consider a real estate related cloud application that takes as input a street address and produces as output an image of the property at that address. The application is focused mostly on real estate recognition within images, and it relies on outside services for data (the initial rough images) from which it determines an appropriate angle, zoom, and bounds. Each time a client requests an image for an address, the cloud application requests and receives rough images from one or more data providers, analyzes the images making decisions on how to present the real estate and possibly requesting more images in the process, and sends a final high quality image to the client. The cloud application developer has run through a

Table 1.2: Average measured costs in a sample cloud application

Resource	**Cost**
Bandwidth for data transferred to and from the client as well as outside services	$0.0005
Outside services which provide rough images	$0.002
Processing time to analyze the images and perform the business logic of the application	$0.00001
Storage of the final image so that a subsequent identical request can be handled with a simple database retrieval of the predetermined image	$0.0002
Total	$0.00271

few common scenarios and has measured average costs (per transaction) as shown in Table 1.2.

Since these numbers are only estimates, and since there are other overhead charges to recover (personnel costs are probably the largest, but also tools, rent, etc.), the cloud application developer decides to charge a flat fee of $0.005 for each client transaction with the application. However, as part of the verification phase of the software, the developer would like to determine if the cost of a transaction will ever exceed the fee being charged to clients for that transaction (or perhaps the fee minus a minimally acceptable profit margin). If there are instances where this can happen, the developer would like to understand how it can happen to determine whether the business plan is infeasible or whether the instances are so rare as to be acceptable.

It should be noted that cloud providers generally do not (at present) charge for cloud infrastructure at the fine level of granularity as outlined here [2], [30], [55]. As shown in Table 1.1, rates are currently on a per hour, per GB basis. Charges are artificially scaled down to a much lower level of granularity. Even if cloud providers never reduce their cost granularity, the verification in this dissertation can be scaled back up as the costs of many transactions are added together. However, the pay-as-you-go economic model is tightly coupled with the elastic resource usage

model of cloud computing [7], and a good case can be made for a downward trend in cloud cost granularity. For example, recent work has covered low level billing and resource tracking which helps enable the downward trend [62].

A further mitigating factor is this dissertation's ready application to additional fields of research. The significant base of WCET research is used by this dissertation, but the results of this dissertation may in turn be useful to the WCET field, particular for soft timing constrains. Low power computing may also benefit from this dissertation, especially since it may have softer constraints. Some limited research already exists to analyze software and compute a bound on the amount of power used by an application [37].

Both WCET and low power computing have very little research in analyzing cases where constraint thresholds are exceeded, which can be valid when constraints are soft (e.g. when constraints are not required to be met for safety or correctness). WCET focuses on finding the single highest execution time since anything over a certain threshold is generally unacceptable. A low power system may allow some limited cases when power budget is exceeded, as long as those cases are understood ahead of time and known to happen infrequently.

For cloud costs, it is likely that constraints could be softer so that some cost excesses are OK as long as developers are making good money on most of their transactions.

1.2 Hypothesis

The hypothesis of this dissertation is as follows:

Cost excesses in cloud applications can be characterized before deployment.

Characterization of cost excesses means that they can be identified and described. The identification of cost excess is against a given cost threshold. The

description of the cost excess includes identifying methods and/or path sections which are particularly important (costly and/or frequent). It also includes an association of the excess with operational profiles.

It is important to emphasize that this dissertation is particularly focused on analyzing applications before deployment. This is an entirely different matter than collecting logs of applications in a production environment and analyzing actual usage to determine how software is being used and how much it is costing. Such postdeployment analysis is important and useful, but it is also reasonably straightforward. From a business perspective, it may also be too late. Rather than proactively understanding costs and profits, this would instead be reacting to possibly unforeseen costs. Of greater interest (and a much more difficult problem) is how to determine costs before deployment.

The hypothesis proposes the possibility of predeployment cost characterization. However, this dissertation is not the end of research in cost verification of cloud applications. On the contrary, it is only the beginning of such research. The discussion here and the evidence presented will argue for the possibility of this type of verification and it will show preliminary techniques for doing so. However, as will be discussed in the assumptions and future work, there is still much to be done to fully realize the possibilities.

Note that in the process of identifying cost excesses, paths and transactions can be identified that do not exceed the cost threshold. Of course, a preliminary concern may simply be whether an application will ever exceed a given cost threshold. If it can be determined that there are no cost excesses in an application, then it can asserted that the application will never exceed the threshold.

Hence, the hypothesis also implies that it can be determined whether an application might exceed a given cost threshold. (In some type of applications with

hard cost constraints, it may be important simply to verify that no paths will ever be cost excessive.) This is a special case of the general hypothesis which asserts that if there are cost excesses, they can be found and described in a helpful way.

Chapter 2

Conceptual Model

This chapter presents a low-level cost accounting approach for tracking cloud infrastructure costs (see also [12]). In a traditional computing model where software is hosted on in-house servers, economics are often considered at a relatively high level. In addition to software development costs, considerations include a large up front infrastructure investment, third party software purchases or contracts, and bulk bandwidth service agreements.

Additionally, an expected overall level of revenue is forecast, and the business attempts to recoup initial costs over time. During system planning, various levels of resource usage would be accounted for by allocating resources with slack, and only in the worst case would unforeseen and excessive resource usage cause system unavailability [7].

2.1 Cloud Application Costs

This section discusses a framework for determining the cost of a single transaction with a cloud application. Four basic areas in which cloud applications incur costs are considered: processing, storage, bandwidth, and outside services [7], [48]. Hence, this dissertation is considering cloud application costs in the context of Software as a Service (SaaS) and Platform as a Service (PaaS).

The cost of a transaction with a cloud application is simply the cost of each of the constituent parts. That is, for a given transaction T with constituents T_p for processing, T_s for storage, T_b for bandwidth, and T_v for services, define the following

$$\begin{aligned} Cost(T) &= Cost(T_p) + Cost(T_s) + \\ &\quad Cost(T_b) + Cost(T_v) \end{aligned}$$

The pricing policies of Amazon Web Services [2], Google App Engine [30], and Microsoft's Windows Azure [55] have been consulted for this dissertation. Note that cloud providers may charge for bulk purchases of resources, much more than a single transaction requires. The costs are scaled down to the transaction level, but ongoing research may help overcome obstacles to more fine grain cost accounting and billing [62]. These fine grain costs can be associated with source code which enables static analysis to help understand how cloud applications incur costs.

Processing Costs

Processing is measured in terms of time [2], [30], [55]. Ultimately, the goal is to determine the clock time required for a transaction ($TIME$) and have in hand a cost per nanosecond ($Cost(ns)$). Furthermore, it is essential to know the number of bytes of memory (MEM) reserved for the duration of the transaction and a corresponding cost per byte per nanosecond ($Cost(mem)$). This yields:

$$\begin{aligned} Cost(T_p) &= Cost(ns) \times TIME(T) + \\ & \quad Cost(mem) \times MEM(T) \end{aligned}$$

Of course, cloud providers/vendors are free to produce pricing mechanisms as they see fit, and processing cost per hour appears to be common. So for example, if a cloud provider charges \$0.10 for an hour of processing, developers may obtain the cost per nanosecond by dividing that number by $60 \times 60 \times 10^9$.

Memory costs are currently not well defined by vendors or not separated from processing costs. Amazon Web Services charges more for memory intensive instances [2], so the difference in price from a standard instance can be scaled down to represent a memory unit cost.

Storage Costs

Storage costs are comprised of two main components: the cost of interfacing with the storage medium and the cost of retaining data on the storage medium [2], [55]. The

former is similar to other costs that have been calculated and can be determined by taking the vendor's rate and scaling it down to the level which is being used for transaction.

The cost of retaining data on the storage medium is more difficult to associate with a transaction. To deal with persistent storage, a variable is introduced into the calculations representing the length of time for which the data stored will be persisted in the storage medium.

The storage cost of a transaction is based on the number of bytes stored (B), the cost per byte for interfacing with the storage medium ($Cost(intf\_store)$), the cost per second per byte for persistent storage ($Cost(store)$), and the persistence duration of the data stored during the transaction ($STAY$). That is,

$$\begin{aligned} Cost(T_s) &= Cost(intf\_store) \times B(T) + \\ & \quad Cost(store) \times B(T) \times STAY(T) \end{aligned}$$

Bandwidth Costs

Bandwidth costs are generally separated into those for incoming and outgoing transfers [2], [30], [55]. If a transaction uses a certain bandwidth in bytes (IN, OUT) and a vendor charges a certain amount per byte ($Cost(in), Cost(out)$), or if the charges can be scaled down to this level, this yields

$$\begin{aligned} Cost(T_b) &= Cost(in) \times IN(T) + \\ & \quad Cost(out) \times OUT(T) \end{aligned}$$

Cost of Services

As developers make use of outside services in the form of SaaS, they incur costs according to the pricing mechanisms established by the service vendors and/or an individual service level agreement (SLA). Assume that these costs are known and that they have been scaled down to a single use. Further, consider for a transaction the set

of service calls made is $\{sc_0, \ldots, sc_n\}$. The total cost for using outside services for a transaction is

$$Cost(T_v) = \sum_{i=0}^{n} Cost(sc_i)$$

Hence, a characterization of total costs for processing, bandwidth, storage, and services has been derived.

2.2 Cloud Application Revenue

Consider two types of revenue for cloud applications: direct and indirect. This dissertation is more aligned with applications and transactions that produce direct revenue, but indirect revenue should be accounted for as well.

Direct Revenue

This work is more naturally applied to cloud applications that directly produce revenue and are made up of one or more transactions which perform a service that can be associated directly with some amount of revenue. For example, if an application is provided to clients as SaaS, developers may charge a specific amount per use of the service.

Note that a cloud application may be SaaS, but that same application may also use SaaS (a sort of aggregator). Of course, clients may instead purchase the services directly from providers rather than indirectly through an aggregating service. However, the aggregator ostensibly is providing some added value in composing the services in such a way that would be more difficult for the clients to do themselves. Clients decide instead to pay for use of the aggregator service which provides a complete package of logic and outside services that fits what clients require.

Paying for a single use is not the only pricing scheme that produces direct revenue. Developers may charge clients only after some number of uses, or they may allow unlimited usage for a single fee. However, developers can attempt to estimate

the per use (transaction) revenue derived from customers, perhaps based on usage data previously collected or usage patterns previously observed.

Indirect Revenue or Budgeted Cost

Many products and services may use a cloud application in part, though it may not be the direct source of revenue. For example, when an individual purchases a book online, she may use a vendor's website to read about the book, access reviews, and find associated products. The software to provide all of these ancillary services may run on the cloud, but the book is the actual product that is purchased and provides the direct revenue. The services were important and certainly supported the purchase. Since the services do not produce a specific amount of revenue, they instead may be allocated a certain budget.

Other products and particularly services may not produce any revenue at all, but may use cloud applications and may benefit from cost calculations. These include information technology infrastructure for private sector companies as well as services provided by governmental and non profit organizations. For example, a local municipality may allow residents to pay their bills online. The services to do this may run on the cloud. Though the municipality may not generate revenue from this service, it is likely saving money since they do not have to pay the cost of opening envelopes and cashing checks. Therefore, they allocate a certain budget to the services and as long as the services stay within the budget, the municipality is pleased with the results.

2.3 Cloud Application Profit

For developers, the ultimate goal is generally to produce cloud applications to either produce profit or stay at or below budgeted cost. This work focuses on technologies that allow developers to understand and minimize costs and/or ensure costs stay within tolerances. Maximizing revenue is not part of this dissertation.

Developers may attempt to fix a single fee for a single usage of a cloud application. However, it should be recognized that for all but the simplest of applications, actual costs will vary to service a transaction, sometimes widely. As developers strive to ensure cloud applications turn a profit, they look for answers to the following questions:

- How much does it cost to service a transaction?
- Does the cost per transaction vary?
- What does the cost of a transaction depend on?
- What is the maximum cost of a transaction?
- What is the average cost of a transaction?
- Could the average cost change significantly?

In addition, it would be interesting to answer the following important questions:

- Could changes be made to software to make it cheaper to service a transaction?
- How can developers determine (before deployment) if their software will produce more revenue per transaction than it will cost per transaction?

The first set of questions fall broadly under the topic of cloud application cost analysis, while the last two questions are specifically optimization and verification. It is likely that the cloud computing optimization problem is largely dependent on the particular application. However, some types of general optimizations may be possible. For example, an approach to workflow cost minimization is given in [61].

Cloud application cost verification, on the other hand, might be accomplished in a largely application independent way. The problem can be stated in this way: given a cost tolerance for a specific transaction, can developers verify that the software completes the transaction (under various conditions) using resources with a total cost less than or equal to the tolerance? Furthermore, can developers characterize cost excessive paths through their software to convince themselves that they are so unlikely as to be negligible? These questions position cloud application monetary costs as a quality attribute or part of a requirement that should be verified during the software development process.

One method for determining the amount of processing resources used is to borrow from the field of Worst-Case Execution Time (WCET) analysis. WCET research makes very precise calculations about cycle time, taking into account method caching and even using data flow analysis to give a tight upper bound on the cycles used for a given function [79].

In addition to WCET, there has also been significant interest in verifying and optimizing an application's energy consumption. For example, an approach for determining worst case energy consumption is presented in [37] which, like this dissertation, builds on WCET analysis.

For cloud application cost analysis, this could provide direct results or promising tools for calculating processing cost. However, cloud computing generally abstracts away from the user (hosted application) the details of the resources it is using. WCET analysis generally relies on intimate knowledge of the host processor to determine a tight and accurate bound. Therefore, in order to adapt WCET to cloud application cost analysis, it may be necessary to use some measured results to find averages and/or make some assumptions about host hardware.

On the other hand, it may be decided that measured results and a dynamic approach to cost verification are altogether easier and more effective. WCET has certain limitations that may make it impracticable for immediate use in this context.

Some aspects of WCET analysis are difficult to calculate through static analysis (e.g. loop and recursion bounds). In these cases, the analysis may rely on user inserted annotations into source code [79]. Such annotations might also be used for calculating a bound on monetary costs using tools borrowed from WCET analysis. Annotations could account for fees charged for the use of outside services. Cloud application developers could also plan for specific bounds on the amount of time an outside service will take to return results as well as the amount of data received (and possibly stored later in a database). These annotations would help a tool perform a static analysis of source code and estimate an upper bound on monetary costs for a transaction.

2.4 An Example

As an example of calculating transaction level cloud application costs, consider the real estate image service mentioned earlier. Suppose an organization has developed software that analyzes images and identifies real estate (homes, apartments, lots) in the image either from a street or aerial view. The developers decide to use this software to provide a service that accepts an address as input and produces an image of the real estate at that address as output. They charge a flat fee per address and deliver a high quality image of the real estate, one that uses the latest imagery and frames the real estate accurately no matter what type of property it is or where it is located.

The developers don't have expertise in hosting web services, and they don't want to incur the initial cost of the necessary infrastructure, so they decide to turn to the cloud for a pay-as-you-go platform. Furthermore, since their expertise is in real estate recognition software, the developers rely on outside services for other parts of their system.

First, the service uses an outside service to convert an address to a geo location (latitude and longitude). It uses a second outside service to find a street level image for the location. If the service has such an image, it may be post processed with an algorithm and sent to customers. Or, the service may request another image at a different scale/width. If the service is not satisfied with the street view, or if the service does not have an image for the location, it uses a third service to retrieve an aerial image for the location. This service allows a fine zoom level to be specified, and depending on the type or extent of the real estate detected, the image may be requested at multiple zoom levels to obtain an accurate image. It may be compared to the street view to determine which would give the best detail in a single image.

After producing an image for a given location, the service stores it in a database for one month. If it is requested again, the service accesses the database and provides it directly. After one month, it is removed from the database.

Now consider the cost calculations for a sample transaction. This particular transaction is processed as follows:

1. Receive street address from customer
2. Determine if there is a stored image for this address and find that there isn't
3. Request geo location for street address
4. Receive geo location.
5. Request street level image
6. Receive street level image
7. Run real estate recognition algorithm and determine closer image is needed
8. Request closer street level image

9. Receive closer street level image
10. Run real estate recognition algorithm and determine size and rotation for final image
11. Send final image to customer
12. Store image in database

Consider now the total cost for this transaction. The following calculations use $\simeq$ to denote rounded values. Unrounded values are used to achieve exact values when possible, and compounding imprecisions due to rounding is avoided.

Suppose the cloud provider charges \$0.10 per hour for processing costs and provides a standard set of memory resources. Therefore, the cost per nanosecond comes to about \$0.000000000000027778. The processing duration for the transaction is 500 milliseconds (500000000 ns), so the processing cost of the transaction is

$$\begin{aligned} Cost(T_p) &\simeq \$0.000000000000027778 \times \\ &\quad 500000000 \\ &\simeq \$0.0000138889 \end{aligned}$$

Suppose the cloud provider charges \$0.01 to interface with the storage medium 10,000 times, and \$0.15 per gigabyte per month for persistent storage. Therefore, the cost per interface is \$0.000001 and the cost to store one byte per month is about \$0.0000000001397. The transaction interfaces with the database twice and the image is stored for one month that is 2 megabytes (2097152 bytes), so the storage cost is

$$\begin{aligned} Cost(T_s) &\simeq \$0.000001 \times 2 + \\ &\quad \$0.0000000001397 \times 2097152 \\ &= \$0.00029496875 \end{aligned}$$

The cloud provider charges $0.10 per gigabyte transferred for both incoming and outgoing data. Therefore, the bandwidth cost for each direction is about $0.00000000009313 per byte. The transaction receives 4 megabytes (4194304 bytes) of data and sends 2 megabytes (2097152 bytes) of data, so the bandwidth cost for the transaction is

$$\begin{aligned} Cost(T_b) &\simeq \$0.00000000009313 \times 2097152 + \\ &\quad \$0.00000000009313 \times 4194304 \\ &= \$0.0005859375 \end{aligned}$$

The service converting a street address to a geo location is free. The service to get street level images costs $0.01 for 10 images ($0.001 per image). The transaction gets two images, so total service costs are

$$\begin{aligned} Cost(T_v) &= \$0.001 + \$0.001 \\ &= \$0.002 \end{aligned}$$

The total cost for the entire transaction comes to

$$\begin{aligned} Cost(T) &= Cost(T_p) + Cost(T_s) + \\ &\quad Cost(T_b) + Cost(T_v) \\ &\simeq \$0.0028948 \end{aligned}$$

If the costs of all transactions were close to this one, the developers might decide to charge $0.01 per transaction to recoup costs and make some profit on each transaction. A viable business plan built on this cloud application would likely anticipate several million transactions per month.

Note, however, that the application could handle transactions with significant cost variations based on the paths taken in the algorithm and the associated service,

bandwidth, and storage variations. For example, in the real estate image service, the service may have found just the right image initially, in which case bandwidth costs would be much lower. On the other hand, if the service had to retrieve data from a specialized aerial image provider for some real estate (e.g. large lots), the costs could be significantly different based on the (probably higher) price charged by the aerial image provider.

If the developers attempt to charge users a single price per transaction, what amount should they choose? Can they determine, before deployment, if/when their own costs will exceed the amount they charge? The answers to these questions will be explored further in this dissertation.

Chapter 3

Research Context

3.1 Overview

This dissertation touches on several important themes which are general research areas that have existed for some time. Its contribution is unique but can be situated within a broader set of current works. Topics that intersect with the ideas in this dissertation include cloud computing, software services, software economics, cost optimization, software testing and analysis, and software engineering.

Much of the literature (though certainly not all) on cloud computing economic analysis is from the cloud provider's perspective (e.g. [51], [77], and [34]). This is natural given the evolution of cloud computing, which sprang largely out of research in grid computing and virtualization [78]. Furthermore, fully functioning cloud infrastructure is a necessary prerequisite for cloud applications, so it is important that the problems encountered by cloud providers are understood and addressed first. As this research matures, cloud users (those writing software targeting the cloud) will benefit from research geared toward cloud applications.

From a verification perspective, there are at least two unique aspects of cloud computing. First is that the user does not have immediate/direct control of infrastructure. Although this is largely a benefit, it also raises verification concerns in the areas of performance, reliability, and security [7]. Second, the pay-as-you-go economic model of cloud computing opens a relatively new area of verification: monetary cost, which is tracked at a lower scale (e.g. an hour of CPU time) than under current economic models (e.g. the purchase of a server).

Concerning cloud application costs, the two most obvious problems are optimization and verification. While cost optimization has received considerable attention in the literature (e.g. [61], [76], and [41]), verification has not. There has been some attention given to measuring the costs consumed by running applications in

the cloud (e.g. [74] and [75]). However, unless it is put in the context of dynamic analysis, measurements only allow for reactive decisions rather than proactive planning.

Of course, the overall business plan of a system has always been a concern. However, the new economic model that cloud computing has established with its pay-as-you-go pricing and complete scalability [7] leads to thinking more in terms of a rate of return that applications produce as they run (i.e. margin). In a traditional environment, a poorly planned system may result in unexpected demand that causes unavailability for some users and perhaps even system wide unavailability. If developers also had poorly planned margins, their losses may be contained within a specific limit. However, with cloud computing the availability of virtually unlimited resources means that services can continue losing money on an even grander scale! Therefore, the understanding of profit margin in the cloud is of even greater importance than in a traditional computing environment.

3.2 Review of Related Literature

This section briefly describes areas of research related to this dissertation as well as representative works from the literature. Subsections delineate groups of related works.

Cloud Cost Estimation and Verification

Verification of cloud applications costs is the main research area in which this dissertation focuses. Cost estimation is closely related since it assumes that some sort of analysis is performed to predict cloud resource usage and/or the resulting monetary costs.

Some existing work has covered estimation and verification of cloud application costs. However, these topics have generally been covered only indirectly or in very specific cases. For example, Chen discusses cost verification but specifically

for grid workflows and proposes cost decomposition as an approach toward understanding and verifying costs [22].

Kudtarkar presents a case study using a comparative genomics tool which shows that running in the cloud and optimizing for the cloud can result in significant cost savings [44]. The cost estimates performed are very domain and tool specific as are the optimizations.

Lu presents work on running the BLAST algorithm in the cloud and providing cost estimation. The estimation is based on the values of tunable parameters specific to that algorithm [52].

Tosic presents a nice tool for estimating monthly cloud resource usage (and associated costs), breaking down cost by type and showing usage over time [73]. Costs are not brought down any lower than the per-month level.

Truong's work is probably the most relevant [74]. It provides extensive measurement based calculations for a scientific workflow, but it focuses on runtime estimation, optimization, and decision making rather than predeployment analysis and verification in particular. Specifically, it does not cover in depth the types and methods of measurements, their tradeoffs, and the associated implications on cost verification. Furthermore, it doesn't take into account a threshold and does not attempt to determine which parts of the application are (or are likely to be) cost excessive.

In another work, Truong discusses cost modeling and calls for cost estimation and monitoring tools from cloud providers and research communities [75]. Although these themes come close to the themes of this dissertation, the specifics are somewhat divergent.

Cloud Cost Optimization

Cost optimization is a sort of sister work to this dissertation. Perhaps a more obvious concern and one garnering more immediate attention, cost optimization attempts to

minimize the amount of money or resources consumed by a cloud application or by the cloud provider.

There are several existing works on cloud cost optimization, but most deal only with cloud providers. Liu covers how cost goals can be met while optimizing execution time [51]. Tsakalozos discusses how client cost goals can be met while maximizing profit for the cloud provider [77]. Henzinger analyzes tradeoffs between price and execution time from the cloud provider's perspective [34].

Lee covers service request scheduling for profit optimization, also from the cloud provider's perspective [48].

Nallur uses a market mechanism such that cloud applications evaluate tradeoffs between quality attributes and price [58]. In this sense, costs are optimized from the cloud application's perspective.

Pandey covers cost minimization specifically for workflows [61]. Costs are measured and in some instances are scaled down from provider prices. This is of particular interest since costs are massively scaled down in this dissertation.

Truong Huu presents an approach for cost estimation and optimization specifically for cloud based workflows [76]. The cost calculations there are similar to those presented in this dissertation and are at the per second and per Mbps scale (well below the common levels cloud providers use but also well above the finer grain levels in this dissertation).

Kllapi has developed a method of optimizing cloud dataflows for completion time given a fixed budget, and cost given a fixed completion time [41]. Also discussed is the approximation of cost as well as costs of fragmentation (paying for the use of CPU even when, during some quanta, no computational resources are required). Fragmentation is not discussed in this dissertation as it assumes a fairly consistent

stream of transactions consuming cloud resources. In practice, fragmentation would have to be accounted for to some extent.

Agarwala covers cost optimization for data storage in cloud applications [1].

Ishii researched cost optimizations for stream computing applications in the cloud [36].

Fine-Grain Cloud Economics and Measurements

Another unique feature of this dissertation is the scaling of costs down to very low levels. Although this is not an entirely new idea, it has not generally been associated with cost estimation for verification. Measuring these lower level costs is basically equivalent to the dynamic analysis approach discussed later. Some of these themes are touched on in related literature.

Amazon's CloudWatch feature is an example of a cloud provider offering utilities to track resource usage below the standard charging granularity [3]. CloudWatch gives statistics in terms of minutes instead of hours.

Deelman measures costs on a per transaction basis for a scientific application running on the cloud [26]. Data related costs are normalized to the byte level, although processing costs are scaled down only to a per second level. Though the work is focused primarily on one specific scientific application, it does contain some discussion relevant to general business concerns for cloud applications.

Sekar offers an approach for measuring (through approximations) the amount of billable resource usage consumed by a cloud application at a fine-grain level [69].

Park's work is also from the provider's perspective and deals with how cloud costs can be recorded and billed in a way that is verifiable and minimally invasive, even at a very fine granularity [62].

Cloud and Software Economics

There are many important general works on cloud and software economics. These works help to place the topics discussed here into a broader context. The following are some of those general works that are most closely related to this dissertation.

Lehmann presents an in-depth discussion of pricing for software in general and services in particular [49].

Bala provides a nice overview of usage-based pricing. Although many types of cloud applications may benefit from this dissertation, those that assume a usage-based pricing scheme will likely benefit the most [9].

Kossman provides a detailed cost analysis for an application involving data intensive transaction processing, but the figures are at a much higher level than the fine grain level discussed here [43].

Yeo discusses highly variable pricing schemes to maximize not only resource usage and provider revenue, but also resource accessibility for cloud users [82].

Lampe accounts for arbitrary SaaS begin and end times. The goal is to work toward resource distribution optimization for the SaaS provider (which is an IaaS user) [45].

Cusumano wrote a short piece in 2007 (just before the explosion of literature on cloud computing) claiming that although usage based pricing schemes were rare at the time, a healthy percentage of customers would prefer that pricing scheme [24].

Cheng discusses a variety of two-part pricing schemes involving a single fee component plus a pay as you go or usage based component [23]. These can be tweaked to optimize profit for the service provider.

Khajeh-Hosseini discusses the details of migrating existing applications to the cloud [38].

Klems provides a framework for determining whether the use of cloud resources makes good business sense for a given application [40]. This is more of a high level planning tool and not a low level technology.

Martens presents a fairly comprehensive list of cloud costs (organizational and technological). He also includes a theoretical model for calculating Total Cost of Ownership (TCO) associated with cloud computing [53].

Nurmi offers a research oriented cloud implementation called Eucalyptus (e.g. [59]) which could be modified to provide the metrics required to enable cost verification from the cloud user's perspective.

Cost has been identified as an important QoS attribute for workflows in very early work [84]. Not unlike many software applications, workflows involve several, often disparate systems or components which individually analyze input data, make specific calculations, and produce output data which is often then used by another component in the workflow [32].

Shibata presents of a study of five workflow applications and their associated costs [70].

Dun performed profiling on data intensive workflows. The approach discussed there is similar to the detection approach for measurement covered in this dissertation [27].

Testing and the Cloud

This dissertation is concerned with testing cloud applications specifically for cost attributes. The following works cover various topics related to cloud application testing. They are helpful for understanding what the realization of the technology

developed in this dissertation might look like and what sorts of implementation issues could arise.

Chan gives an overview of modeling and testing cloud applications [20].

Robinson provides an in-depth discussion of strategies for and levels of testing in the cloud [65].

King has devised an approach for integration testing of remotely hosted cloud services (services outside the user controlled cloud application) [39].

Bai presents an excellent and comprehensive overview of testing tools for cloud applications [8]. Some features include cloud simulation and cost related testing (though not at the fine grain, transaction level discussed in this dissertation).

Zhang also presents an approach to model the cloud for testing cloud applications [85]. The work simulates cloud states in order to present a realistic representation.

Riungu presents research needs based on interviews with industry practitioners [64]. Specifically mentioned is the need for a way to estimate costs and price services accordingly.

In addition to presenting a cloud testing framework, Wu enumerates some key features of a robust cloud application testing framework [80].

There is also a wealth of literature related to testing clouds and cloud infrastructure (not cloud applications). One of the more popular example is CloudSim which is a fully featured cloud modeling and simulation suite use extensively in cloud infrastructure research [19]. However, that work is not closely related to this dissertation.

General Software Engineering and the Cloud

In a very broad sense, this dissertation is concerned with software engineering for the cloud. The following works present a summary of important works in this field.

Tai sets the stage for the software engineering of cloud services by defining and describing the basic characteristics of cloud services as well as their business and value context [72].

Yau points out that the cloud fundamentally changes the delivery, deployment, and maintenance of the software lifecycle. This affects the engineering of software services [81].

Rellermeyer emphasizes the importance of loosely coupled, highly cohesive modules to form the basis of distributed cloud based software applications where modules are a first class entity [63].

Singh describes the software development lifecycle (SDLC) in terms of the cloud [71]. The cloud specific pieces of each phase make up the cloud development lifecycle (CDLC).

Mei compares key aspects (I/O, storage, and calculations) of cloud computing to service and pervasive computing and presents important questions that cloud application developers should keep in mind [54].

Boehm's well-known work on value based software engineering is both directly related to this dissertation and in some ways parallel [11]. Boehm advocates incorporating value into the entire software engineering process, much as this dissertation advocates incorporating cost into much of the process. Furthermore, Boehm discusses operational costs as part of business case analysis which corresponds exactly to the costs discussed here.

Cloud Based Services

Many of the applications that are likely to find this research useful will be cloud based services, particularly those embracing the pay-per-use economic model. Following are descriptions of cloud based services that are particularly relevant to this dissertation.

Rostrom provides a prototype of a cloud based service for medical imaging which could operate on a pay-as-you-go basis [66].

Lau presents an analysis of IP-based video on demand hosted in the cloud [46]. This service is pay-per-use.

Cai emphasizes the differences between cloud infrastructure services and cloud application services in meeting customers' needs, as well as providing billing policies that align with customer objectives [17].

Lee points out that a quality attribute for SaaS hosted on the cloud is efficiency, and that efficiency is associated with the pay-per-use aspect of cloud computing (and often SaaS which may not require up front purchases) [47].

Koehler found in a survey that customers strongly prefer subscription based pricing of cloud based SaaS over pure pay-per-use pricing, and even slightly prefer one-time based pricing over pure pay-per-use [42].

Hou discusses application software running on the cloud for use in high performance scientific domains [35]. These cloud based services are offered on a pay-per-use basis, and the associated cost model is analyzed to show that the result is likely to be favorable for the customer.

Gmach analyzes several sample cloud workflows and provides important characteristics such as memory and CPU usage plus associated costs as part of the results of the analysis [29].

Table 3.1: Comparison of related literature

	Perspective		Analysis		Costs		Contribution	
Name	**Provider**	**User**	**Measure**	**Predict**	**Full**	**Scaled**	**Optimize**	**Verify**
Buell		X	X	X		X		X
Truong Huu		X		X		X	X	
Chen		X		X				X
Kudtarkar		X		X	X		X	
Lu		X		X	X			
Pandey		X	X			X	X	
Deelman		X	X			X		
Sekar		X	X			X		
Tosic		X	X		X		X	
Truong		X	X		X		X	
Kossman		X	X		X			
Nallur		X					X	
Agarwala		X					X	
Kllapi		X					X	
Ishii		X					X	
Tsakalozos	X						X	
Liu	X						X	
Henzinger	X						X	
Lee	X						X	
Park	X							

3.3 Contribution of the research

Table 3.1 situates this research within the body of existing related work on cloud computing economic analysis. The table attempts to show a concise description of related works based on categories particularly important to this research. Not all of the categories are applicable to all of the related works.

Only the most closely related works are shown in the table. For example, there are several works on general cloud economics and specifically in the area of migrating to the cloud (see [50], [40], and [38]), but these are not considered in the table as they are not closely enough related.

Categories listed in the table are:

- Perspective - Whether the work considers the cloud provider's perspective or the cloud developer's perspective (user perspective). Other perspective could be interesting (governmental regulator, cloud application user, etc.) but are not considered in this dissertation.
- Analysis - The type of cost analysis performed in the work. Measurement is for dynamic analysis performed during and after the cloud application runs (which may occur before or during deployment of the application). Prediction is for static analysis and estimates an upper bound on costs before the cloud application is deployed.
- Costs - Whether the work considers costs directly from the cloud provider's pricing scheme (generally a relatively high level such as per hour or per GB) or whether costs are scaled down to lower levels (e.g. cost per second, cost per transaction).
- Contribution - The contribution of the work relevant to cost analysis. The types here are limited to optimization and verification since these are most relevant to this dissertation, but there are certainly many other diverse contributions of the works listed.

To use the concise terminology from the table, this research is from the cloud user's perspective and considers both prediction and measurement of scaled costs for the purpose of verification. Also note that the columns in the table corresponding to this dissertation are shaded. This clearly demonstrates that this dissertation is particularly unique in its consideration of the prediction approach for cost analysis along with its ultimate goal of cost verification.

Furthermore, the entire conceptual model of verification of low level costs is unique to this dissertation. This dissertation advocates a novel approach to verification of cloud transactions by first setting a transaction level cost threshold then verifying that it is met.

But this dissertation goes even further by determining where cost excesses originate–whether in particular methods, sections of code, or perhaps operational profiles. Hence, this dissertation explores approaches to providing cloud application developers guidance as to how they can meet cost thresholds and under which conditions cost thresholds may not be met.

Chapter 4

Overview of Research Approaches

4.1 Introduction

Monetary cost has always been an important attribute of software systems, but the pay-as-you-go economic model of cloud computing changes how costs are planned for and incurred. Cloud computing provides elastic resources that scale on demand and mimic the consumption and charging patterns of a traditional utility [15].

It is important to understand how much applications cost to run for at least two reasons. First, there may be a specific budgeted amount for each transaction within an application and it is essential that the budget is not exceeded. For example, scientific workflows are not concerned with profit but are often conscious of cost. Second, an application may be running as a direct part of an activity for which a fee is charged and from which some amount of profit should be derived. In this case, it is important to ensure that the software stays within expected costs in order to ensure that profit objectives are met.

4.2 Cost as a Software Quality Attribute

Since cloud computing costs are incurred at a much finer scale (e.g. server usage per hour) than traditional computing, cost verification can also be scaled down to a much lower level [12]. Indeed, this dissertation advocates tracking cost as a quality attribute starting at the use case level and flowing down even to the unit and function level.

As with most software quality attributes, the cost to run an application is one that should be understood well before the application is deployed. An obvious consequence of having a cost excessive use case within an application would be a cost overrun. Precisely because of the cloud's resource elasticity, the extent of cost overruns is possibly much higher than in a resource limited traditional computing environment, particularly for customer facing applications where customer behavior may be forecast but not entirely known ahead of time.

Fixing a budgeted cost for a use case may be quite natural for many cloud applications, but particularly for services operating on a pay-per-use model. If developers charge customers a certain amount per transaction to use their service, they would also want to account for the cost of servicing the transaction. For example, if developers are running credit card transaction processing middleware in the cloud, then it is only natural that they can and would account for the cost of each transaction when deciding how much to charge customers per transaction.

On the other hand, some cloud applications provide only ancillary services to an ultimate customer transaction. Others have no direct revenue, but are still likely to be under some sort of budget constraints. Many types of cloud applications could potentially benefit from monetary cost verification.

Applications involving large amounts of data–and particularly large amounts of data variability–require special attention for cost verification. In particular, cost verification methods involving only static analysis and/or control flow analysis may not work well for data-intensive applications. A dynamic, measurement based approach is more likely to produce adequate and accurate cost verifications for these types of applications.

Furthermore, dynamic measurement is probably the verification approach most likely to be used in the near term (and possibly currently in use) in production systems. Given the absence of other approaches to verifying monetary costs in cloud applications, a very simple, very high level measurement of the costs incurred by cloud applications are the costs reported by the cloud provider during billing. These are tracked in terms of the associated resources consumed, as shown in Figure 4.1. Breaking this down further, a cloud application developer might measure the types of applications that were run during a billing cycle to get an idea of which applications (perhaps even which components and use cases) contributed to the total cost in specific ways.

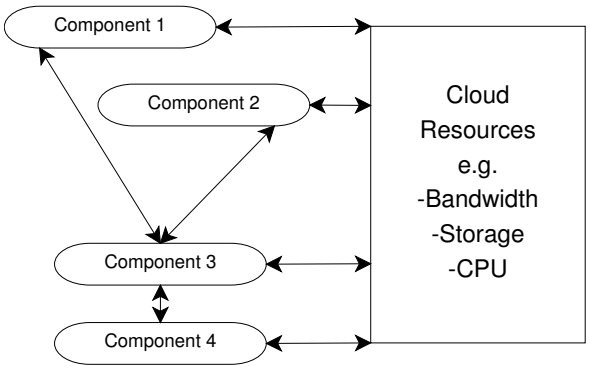

Figure 4.1: Sample cloud application

Just as the development of certain units can be associated with use cases, the budgeted cost of a use case can be broken down into constituent pieces. For example, a particular component, unit, or even function in a piece of software could be allotted 75% of the cost of a transaction while another method gets 20% and another 5%.

Associating a cost constraint with components, units, and functions is probably the ultimate decomposition of the problem and would promote cost tracking down to the unit test level. Though budgeted costs at this level would likely be extremely small, they would scale back up as the usage of the software scales up to service thousand and millions of transactions, or even more.

Of course, this would only naturally lead to an attempt by a cloud application developer to independently measure resource usage in order to determine where costs are coming from and how they vary. Hence, the following work on dynamic approaches to cloud application cost verification is not only important for data-intensive cloud applications and workflows, it also provides approaches to cloud cost verification that might be immediately applicable to many types of cloud applications. Transaction oriented services using a pay-per-use cost model will likely find this dissertation particularly beneficial.

4.3 Dynamic Verification

A dynamic verification approach would involve taking measurements on applications as they run and calculating costs from these measurements. The application under test might be run off the cloud during this dynamic verification which could be easier, less costly, and quite accurate except for a possible difference from cloud hardware in CPU time. The application might be run on the cloud so that measurement could be more accurate, but the testing process on the cloud could also be more costly.

As with any black box testing approach, a dynamic cost verification approach would not guarantee that cost objectives are met. However, using standard software testing techniques like boundary values and equivalence partitions, this approach could provide a reasonable and practical estimate.

This section covers three basic approaches to dynamic measurement for cost verification (also see [14]). First, an instrumentation approach takes measurements by modifying the software under test, essentially wrapping it with a measurement layer. Second, a detection approach requires no modification to the software under test but instead intercepts the requests made by the software for resources that incur cost.

These first two approaches are specifically designed so that they can be used off the cloud. Indeed, the measurements they produce are independent of the cloud provider. This is important because they have the advantage that their measurements can be applied to several cost profiles of different cloud providers and the various resulting costs can be compared.

On the other hand, the third approach would be delivered by a cloud provider. It would require no modification to the software and relies on a sort of measurement layer built into the provider's infrastructure.

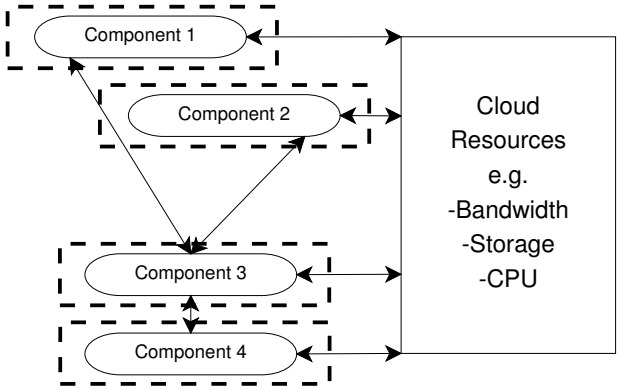

Figure 4.2: Sample cloud application showing measurement via instrumentation

Though these three approaches may seem similar, their differences have non trivial implications for the realization of a cost verification strategy.

Instrumentation Approach

The instrumentation approach is characterized by 'surrounding' each component with a measurement layer, as shown in Figure 4.2. This layer provides an interface to cloud resources such as bandwidth and storage, but it is only a pass through to the actual resource layer. The measurement layer simply keeps track of the amount of resources requested before forwarding requests to the resource layer. A very basic measurement layer could be very easy to implement and an example of a thin measurement layer is found later in this dissertation.

The potential downside here is that the user may need to update the software under test to use this measurement layer. If the software is developed using the measurement layer from the start, the burden for such an approach could be minimal. As an alternative, the user may only need to run the source code through a preprocessor or special compiler to automate the task of inserting the measurement layer where appropriate. This would be more complicated to implement but would ease the burden on the user considerably.

Note that the contribution of processing time toward the overall cost of running a component would not be instrumented. It is a fairly simple calculation and would be determined much as described below for other resources in the detection approach. However, any processing time spent in the measurement layer would be subtracted from the overall time as it will not be used during a production run of the software.

An instrumentation approach could be used to analyze and test software locally (off the cloud). The measurement layer could forward requests to local resource providers (locally hosted DB, sockets, etc.). But the measurement layer could also forward requests to the cloud provider and so the instrumentation approach could be used for verification directly in the cloud as well.

The instrumentation approach is very much compatible with (and similar to) unit testing. A special implementation of the measurement layer could be made for unit tests to interface with mock resource objects. This would allow for extensive testing of various control flow paths in an application and would track the corresponding resources that are requested by individual units. These could then be rolled up into higher level component tests, and developers could more easily track how resources costs are accrued in their applications. The ability to ensure extensive test coverage (likely through unit and component testing) for the purposes of and from the perspective of resource cost tracking would generally be well handled by an instrumentation approach.

Consider a concrete example that could use an instrumentation approach, a corporate wide accounting related application for a very large, multinational corporation with several divisions/companies and ultimately hundreds of separate accounting units. This software is internal to the corporation and its users/customers are employees in the various payroll departments of the different accounting units.

The application's components are the payroll applications for each of the corporation's accounting units, and each component is comprised of subcomponents that query relevant databases for pay scale, timecards, personnel data, etc. All of the components and subcomponents have been migrated to the cloud, and it is essential that the components run in a timely manner and that they stay within a budgeted execution cost.

The payroll application is a fairly extensive end to end solution for running the components from the different accounting units, aggregating data, interfacing with outside entities (i.e. banks) for direct deposits, etc.

Under the instrumentation approach, enterprise level developers would either modify the individual components to make use of instrumentation hooks, or they would require that the different IT departments for each accounting unit make the change so that data can be collected for the overall application. Although this could represent a significant retrofitting of existing applications, it allows different organizations to manage and verify their own components as long as they use a common instrumentation approach.

Detection Approach

The detection approach is less invasive than the measurement approach. It relies on an outside entity to monitor requests for storage and bandwidth resources, intercepting requests for resources as shown in Figure 4.3. Such an entity would be running apart from the actual application under test and would not interfere with its execution. This approach would likely require no modification to the software under test, so it would be easy to use.

The monitoring application is responsible for measuring resource usage using whatever mechanisms are made available to it. For example, a socket library may allow for logging of bandwidth usage. Likewise, a database provider may have

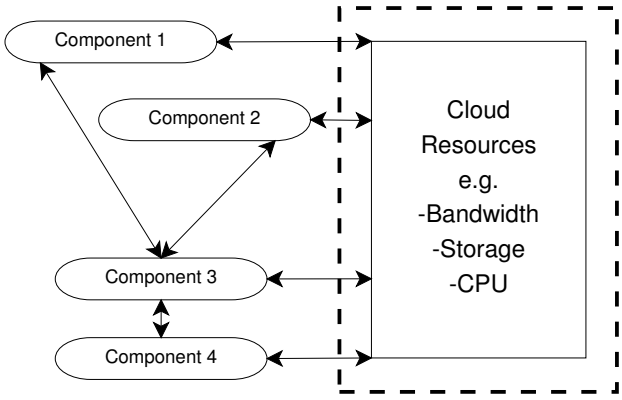

Figure 4.3: Sample cloud application showing measurement via detection

mechanisms for triggering detection software when updates are made or queries are run.

The detection approach is assumed to run off the cloud. (A similar approach that runs on the cloud is discussed in the next section.) It is important to verify the cost usage of cloud applications off the cloud for several reasons. A cost profile (or resource usage profile) could be developed independent of cloud provider that can help developers decide which cloud provider's pricing scheme would result in the lowest cost for cloud applications. Furthermore, certain levels of testing (e.g. unit level) or harnesses (isolating certain components) may not be compatible with running on the cloud. Testing needs may also be immediate, isolated, and/or sporadic which may decrease the advantages of running in the cloud.

A significant problem with the detection approach is that its implementation could be quite difficult. Depending on the different computing environments (DB, OS, etc.) supported by the implementation or test framework, the work could be substantial. Since there is no standard for making logging information available for storage and bandwidth resources, the monitoring applications would have to include some individual strategy for each database provider and bandwidth provisioning mechanism. Furthermore, the capabilities for performing detection and logging are

likely to vary from vendor to vendor, so achieving a common set of functionality in this space may be difficult or impossible.

This is different than the instrumentation approach, which requires a well defined interface for requesting resources. The instrumentation approach is simply inserting a layer to take measurements. Unlike that approach, the detection approach relies on the resource providers' functionality for detecting and logging resource usage.

Returning to the accounting application example, the detection approach on the surface seems very promising since the various IT departments would not have to update their components. In practice, the detection approach may be quite challenging since the different departments may use disparate and dissimilar systems. Either enterprise level developers would require each department to be responsible for providing the detection functionality for their own unique systems (not ideal, to say the least), or they would have to ensure that the solution they provide accounts for all of the different systems in use (if that is even possible).

Provider Based Approach

Another possible approach is for the cloud provider to make available an option to enable fine grain cost and/or resource measurements, as shown in Figure 4.4. This would be similar from a user's perspective to the detection approach since it requires no update to the cloud application. An approach to fine grain billing can be found in [62], and could form the basis for a provider based measurement option.

To clarify what this approach entails, consider what a cloud provider would need to add beyond what a basic cloud billing system would already record and make available to users. Currently, cloud providers generally charge on the scale of per hour for processing costs, per gigabyte for bandwidth costs, and per gigabyte per month for storage costs.

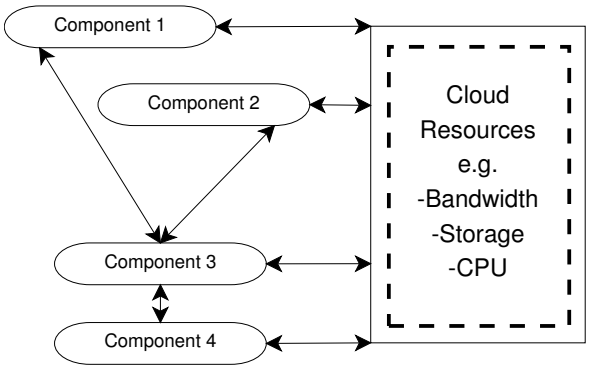

Figure 4.4: Sample cloud application showing measurement within the cloud provider

Ultimately, it would be desirable to record resource usage down to the unit level of cloud applications. (In practice, this could be the subcomponent level or the interface level of components.) This would entail recording CPU usage down to the millisecond or even nanosecond, bandwidth down to the kilobyte or even byte, and storage usage also to the kilobyte or even byte level. To reiterate, the reason for these fine grain metrics would be for developers to understand and project costs as the number of transactions is dramatically scaled up, transactions are combined, and/or users iterate through large amounts of data for a given transaction.

Though some cloud providers make available some amount of data below the basic billing scale, Amazon's CloudWatch feature is currently the closest to enabling the types of metrics that would be required from the cloud provider for fine grain cost verification of cloud applications [3]. Though not yet at the level proposed here, its features at least demonstrate the demand for and usefulness of cloud resource usage metrics far below the per hour, per gigabyte scale.

Note that an interesting aspect of the cloud provider approach is that the software would likely have to be tested in the cloud. In many ways, this is ideal since it more closely simulates the production environment. However, it could also incur unwanted costs. For an in-depth discussion of strategies for and levels of testing in the cloud, see [65].

Furthermore, this approach by definition is tied to a specific cloud provider. The results of the measurements will only be applicable to that particular provider, and the user may not be able to derive enough information from the results to determine what the cost would be using a different provider. Although this kind of comparison shopping is not the main focus of this dissertation, it is certainly a related concern.

Allowing users to comparison shop other cloud providers based on resource usage profiles is probably not an incentive for cloud providers to make low level cost and resource usage data available to users. However, there could be other incentives. Amazon's CloudWatch is offered for a fee [3], so deriving revenue from these metrics is one option. If the demand for such metrics increases, it may become an assumption that a cloud provider will make them available as part of a value added package expected by cloud users.

Returning once again to the enterprise accounting and payroll example, this system would fit very well with a cloud provider based system. Each of the separate divisions could provide their own components to the application, none of which would have to be updated to use the system. All the cost verification data could be aggregated in one place and used through one interface.

On the other hand, developers would be completely reliant on the cloud provider for the kinds of metrics that would allow cost verification. Furthermore, this cost verification approach may not be portable to other providers since the metrics and gathering would likely be provider specific.

4.4 Summary and Limitations of Dynamic Approaches

Cost verification through measurement is practical for practitioners and could provide reasonable results. This section has presented three measurement based approaches for cloud application cost verification. Given the advantages and limitations of the

various approaches, it is likely that an instrumentation based approach would make a good first step in measuring cloud costs.

The detection approach has significant barriers to implementation and/or realization. Because this approach assumes the usage of local resources (testing locally, not on the cloud), it is complicated by the various resource providers' capabilities and interfaces. The cloud provider approach is interesting and may ultimately be the easiest and most accurate, but it relies on the willingness of cloud providers to make metrics available at a very fine scale.

An instrumentation based approach could start very simply and eventually be made more complex (with the intent of making it easier for the user). It can be used for cost verification strategies that run on or off the cloud. The instrumentation approach does not rely on metrics made available by cloud providers, so it can both compare cost profiles from different providers and it can stand independent of various levels of cloud providers' measurement features. Though each of the three approaches deserves more analysis, the instrumentation approach would likely be the best for independent research on measurement based cost verification.

Note, however, that dynamic approaches to cloud application cost verification are inherently imprecise. They are only as good as the testers understanding of the software, possibly even its internals. Dynamic methods are theoretically hit and miss–whatever gets executed for the measured test runs is what is represented as the overall behavior of the system. Of course, the test cases are ostensibly selected wisely and with extensive domain experience by experts, but there is always room for something to be missed.

4.5 Static Verification

Another approach to verifying software costs is to perform static analysis of the constituent components of an application. The goal of this kind of analysis would be

to predict, without actually running the software, what sorts of costs would be incurred for the various kinds and amounts of inputs for each component. This would be more of a white box testing approach which would necessarily make use of full access to the source code.

Static analysis has many limitations which will be further discussed later. However, it has been selected as a research approach for investigation in this dissertation for two main reasons. First, since static methods may uncover cost excessive in source code that dynamic verification misses, it is important that an exploration of the tools for a static analysis be undertaken. Second, static analysis methods of cost verification for cloud applications are non existent, and it would be interesting to explore the benefits and limitations there.

Ultimately, a dynamic or perhaps a hybrid method may prove the most immediately available if not the most successful. However, from the current state of research and practice, it would seem that the development and evaluation of a static analysis approach to cost verification for cloud applications would be beneficial and unique as a contribution to the body of related research.

Chapter 5

Static Research Approach

5.1 Overview

This section describes an approach for calculating the maximum cost of a transaction in a cloud application through static analysis. A method is also described for determining the control flow paths that can cause cost excessive transactions, and ranking methods and decisions based on the amount they can contribute to the cost excess. This allows for verifying cost in cloud applications, identifying control flow paths that can lead to cost excess, and pointing out areas in which to concentrate efforts in order to reduce costs (also see [13]).

5.2 WCET Adaptation

The approach for determining cost excessive paths begins with an adaptation of WCET analysis (which has already been identified as applicable to cloud computing in [12] and [34]). The intermediate goal is to determine the maximum monetary cost of a given method within the application. A given transaction with the application begins with some method and follows a control flow path through an interprocedural graph. Note that the term 'method' is used here but this might be used interchangeably with 'function' or 'procedure' depending on the programming language in use.

The processing component of a method's maximum cost can be derived directly from the WCET. This cost component is simply the WCET (converted to nanoseconds) multiplied by the cost per nanosecond that the cloud provider charges (the per hour cost scaled down to the nanosecond level).

For the other cost components (bandwidth, storage, and services), user inserted annotations are used. WCET analysis already embraces annotations for information that cannot be (or cannot fully be) calculated through static analysis [79]. For example, loop bounds are often annotated. In order to calculate worst case cost,

annotations can be inserted in source code directly where bandwidth or storage are used, or where services are invoked.

The annotations are realized as standard Java Annotations on method definitions. These are available to clients during introspection, and the WCET adaptation looks for these annotations when the existing introspection logic is carried out for standard WCET analysis. An annotation is defined for each resource and is given a value of an appropriate type. Following is an example of an annotated method:

```
@ServiceCost(0.005)
@ServiceTimeBound(500000000)
@ServiceBandwidthBound(4096)
public Byte[] retrieveOutsideData()
{
   ...
}
```

These annotations indicate that the method invokes an outside service costing $0.005. The bound on the amount of time required to wait for the results of the service invocation is 500 milliseconds. (Waiting for an outside service is assumed to be synchronous here and the cost of that wait time is accounted for in the CPU cost.) The amount of bandwidth required for data transfer with the service is 4KB. The service cost is used directly in max cost calculations. The time and bandwidth bounds are converted to monetary cost based on the provider's rates, and those contributions are added to the max monetary cost.

It should be noted that annotations do not place a bound on a certain cost type for an entire transaction. User inserted annotations are required only at those points of the call graph where resources are used. The WCET adaptation must account for all the annotations within the context of loops and decisions while traversing the call

graph to arrive at a final bound on the cost of a method. So the WCET adaptation is still important even for annotated costs. An attempt to calculate a bound or derive such a bound through analysis, simulation, or measurement is outside the scope of this dissertation. For now, annotations are used.

Also note that the scaling and tracking of costs down to the transaction level, though discussed to some extent in the literature (see [12] and [26]), is not immediately compatible with how cloud providers charge for cloud resources. Cloud providers generally charge at a higher scale (e.g. per hour for bandwidth and per GB for bandwidth). It is implicit in these verification techniques that the cloud provider's costs are scaled down to the transaction level, but that as the number of transactions increases, the actual usage of resources rises to the level tracked and charged by cloud providers. Also note that some cloud providers are offering usage statistics at a scale much lower than per hour, as in Amazon's CloudWatch [3].

The WCET adaptation is simply the sum of processing cost plus bandwidth, storage, and service costs accounted for throughout an application's call graph. Whereas WCET deals only with processing time and uses either a graph based or linear programming based approach to determine worst case cost [79], the WCET adaptation determines monetary cost and only uses a graph based approach (so that additional node level information can be saved for later calculations). In particular, a graph corresponding to the control flow nodes in a program is built and the (maximum) cost of each node is determined by adding together and finding the maximum of all the resource costs associated with the node and its children.

5.3 Decision/Call Graph Analysis

As mentioned previously, the basis of cost analysis here is a WCET adaptation that calculates maximum monetary cost of a method in a cloud application. More specifically, as the max cost algorithm runs, information is saved about each method as well as each branch of each decision, all within the interprocedural graph. The

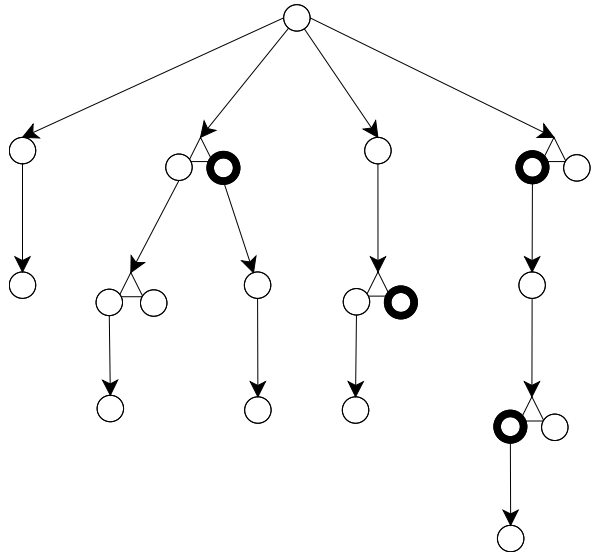

Figure 5.1: A sample Decision/Call Graph (DCG)

result is a Decision/Call Graph (DCG) where a node is considered to be either a method or a decision branch (not the decision itself). See figure 5.1 for an example of a DCG. Methods and decision branches are represented by circles. Decisions are represented by triangles. Nodes on the path of worst case cost are shown in bold.

Note that the set of unique paths through an interprocedural call graph is based on decision branches. Whether one branch or the other is taken determines the path followed. Loops complicate the matter, but a simplification is made by assuming that the same decision branch is taken for each iteration of a loop for a given path.

The DCG is defined as similar to a standard directed acyclic graph, but with a fully ordered set of nodes (i.e. a method m or branch br) encountered as a program is executed, along with the set of edges that connect the nodes. By convention, nodes are drawn in order from left to right when represented graphically. The two branches of a condition are drawn together to show that they are related, but only one of the branches of a given decision will be part of any path of execution (from each unique parent).

The DCG can be represented formally as a set of nodes and edges, as follows:

$$node \quad = \quad m \vee br$$

$$
\begin{aligned}
DCG_N &= \{node_1, \ldots, node_n\} \\
edge &= \{node_i, node_j\}, \text{ with } node_i, node_j \in DCG_N \\
DCG_E &= \{edge_1, \ldots, edge_m\}
\end{aligned}
$$

Note that an execution path will follow all invoked methods leading from a source node (assuming no early returns). However, only one execution branch will be followed when a decision is encountered. Therefore, when representing a path (P), the methods can be omitted without loss of precision.

A path is represented formally as follows:

$$P = \{br_1, \ldots, br_r\}, \text{ with } br_i \in DCG_N$$

During the calculation of worst case cost, the following pieces of information are calculated and saved for each node in the DCG:

- $MaxCost(node)$ - The full cost of the node. This includes the cost of all children nodes (accounting for loops appropriately). This is calculated as part of the WCET adaptation and can be used to find cost excessive paths, but it is not used in any of the node ranking calculations discussed later.
- $BaseCost(node)$ - The base cost of the node, or the cost of only the statements within the node. The base cost is not path specific. It is calculated by counting only 'local' statements in the node (not invoked methods or contained decision branches).
- $MaxTimes(node)$ - The maximum number of times each node can be invoked by its parent(s). This value is path specific (i.e. it may vary and depends on the particular path). For decision branches, it is calculated by starting with the *MaxTimes* value for the containing method and multiplying by any loop bounds in which the branch is contained (multiplied together if nested). For methods, it

is also calculated based on the *MaxTimes* of the invoking method and loop bounds at the site of each invocation. All invocations anywhere in the parent method are added together to determine the total number of invocations from that parent. Currently, recursion is not handled (see the limitations section of this chapter for a discussion of recursion).

For each path, the following are calculated:

- $MaxCost(path)$ - The maximum cost of the path. The *MaxCost* of a top level node will be produced by the max cost path. However, also of interest is the (max) cost for all paths, not just the path of highest cost.

A path's max cost is calculated using the worst case cost as a starting point and subtracting lower cost branches while traversing down the DCG, as follows:

```
float MaxCost(P) :-

// topLevelNode is the entry point
//   of the transaction
Set pathCost = MaxCost(topLevelNode)

For each decision branch br in P
  // brs is the sibling branch of br
  Set branchDiff = MaxCost(brs) - MaxCost(br)
  If branchDiff > 0
    pathCost -= (branchDiff * MaxTimes(br))

return pathCost
```

Of course, another approach is to simply add the base cost of each DCG node along the path multiplied by the max number of times that node can be invoked in the path, like this:

```
float MaxCost(P) :-

// Assuming that MaxTimes and BaseCost have
// already been calculated for each DCG node
Set pathCost = 0
For each DCG node along P
  pathCost += BaseCost(node) * MaxTimes(node)

return pathCost
```

A straightforward but very important definition that should be made is for cost excessive paths. Given a cost threshold THR, note that some $path$ is cost excessive if $MaxCost(path) > THR$. Note once again that the cost threshold, though not arbitrary, is not calculated as part of the algorithms in this dissertation but is instead determined by business case analysis likely involving expected revenue, usage, resource costs, and other information.

5.4 Characterizing Cost Excess

Enumerating cost excessive paths is important, but it is even more helpful to understand the degree to which nodes in the DCG contribute to the max cost. The following calculations are made to determine the magnitude to which a DCG node contributes to cost threshold. Since the magnitudes are normalized by the amount of cost excess for a given path, they can be compared between paths. Therefore, a single/maximum magnitude (or weight) for a given DCG node can be determined.

Of course, it cannot be determined for sure (without more information about how the application will be used) whether a particular node's cost excess is more or less important since it is unknown how often the cost excessive paths which include the node will be executed.

Start by calculating the raw cost excess for a particular path. This is simply the maximum cost of the path minus the cost threshold:

$$CostExcess(path) = MaxCost(path) - THR$$

Next, determine the 'weight' of the cost excess for a path. The weight is intended to measure the magnitude of the path's cost excess as it measures the percentage of excess in relation to the threshold. It is calculated as follows:

$$PathWeight(path) = \frac{CostExcess(path)}{THR}$$

Likewise, the weight of a node measures the contribution of an individual node to a path's cost (the percentage of the path cost attributed to the node). It is calculated as follows:

$$NodeWeight(path, node) = \frac{BaseCost(node) \times MaxTimes(node)}{MaxCost(path)}$$

Now determine a normalized node weight which measures the magnitude of the node's cost in terms of its path's cost excess. Since it is 'normalized' by the path weight, it can be compared to the normalized weights of nodes from other paths. It is calculated as follows:

$$NormalizedNodeWeight(path, node) = NodeWeight(path, node) \times PathWeight(path)$$

The overall normalized weight of a node, independent of path, is simply the maximum normalized weight of the node on any path and is calculated as follows:

$$\begin{aligned} & NormalizedNodeWeight(node) \\ = \; & Max(NormalizedNodeWeight(path, node)) \end{aligned}$$

5.5 An Example

Consider again the stock trading middleware example. The developers charge \$0.10 per transaction and they have set a \$0.02 cost threshold for the transaction. This margin and the vast number of transactions handled allow developers/owners to pay staff and other costs plus derive some profit on top.

Suppose the WCET adaptation has been run and that the *MaxCost* and *BaseCost* have been saved for each node in the DCG. Also, *MaxTimes* of each node has been determined for each path as well as *MaxCost* of each path.

Now, consider a path P_1 that represents the servicing of a fairly standard stock purchase with only basic database access plus average latency and bandwidth with outside service and data providers. It has been determined that $MaxCost(P_1) = \$0.024$ so clearly, this yields

$$CostExcess(P_1) = \$0.004$$

Furthermore, the path weight shows that P_1 exceeds the cost threshold by 20%.

$$PathWeight(P_1) = \frac{\$0.004}{\$0.02} = 0.2$$

Since this is a common case, this cost excess may be of concern. The node weight for each node along P_1 is calculated next. For example, suppose there is a method m_1 along P_1 (i.e. between two decision branches that define P_1). The method's base cost is \$0.00004 and the max number of times it can be called is 30. Next, it is

Table 5.1: An example of normalized node weights

Method	Normalized Weight
m_4	0.15
br_{15}	0.11
m_{31}	0.06
m_1	0.01
br_9	0.009
m_{11}	0.007
...	...
m_3	0.0005

determined that this node accounts for 5% of the path's max cost.

$$NodeWeight(P_1, m_1) = \frac{\$0.00004 \times 30}{\$0.024} = 0.05$$

In order to make useful comparisons between other nodes in the DCG, the node weight is normalized based on the path's weight. This yields

$$NormalizedNodeWeight(P_1, m_1) = 0.05 \times 0.2 = 0.01$$

For this example, suppose that the normalized node weights for m_1 along all other paths were smaller than the weight for P_1. It can then be concluded that the normalized node weight of m_1 is 0.01.

Now, suppose that similar calculations have been made for all the other nodes and paths. The normalized node weights could then be ordered as shown in Table 5.1.

From this analysis, it might be concluded that m_1 is of average importance in terms of its contribution to cost excess. It is likely that m_4 and br_{15} would be of particular concern. It may be important to look at which cost excessive paths those nodes fall on and try to determine how likely those paths are to occur. If those are thought to be likely, attention should be focused on reducing the cost excess of those nodes in order to reduce the overall probability of cost excessive transactions.

Indeed, the common path P_1 does contain m_4. Upon further inspection, it is found that m_4 invokes an outside data service whose per transaction cost is no longer market competitive. Ultimately, the decision could be made to either renegotiate a better rate for the data service or find an alternate provider.

5.6 Evaluation Methodology

Evaluation of the static analysis approach was accomplished by first implementing a research prototype of the WCET adaptation developed earlier, and then by verifying the prototype using a separate and independent utility.

Research Prototype

A prototype was built on the Volta suite of tools [33] for WCET calculations on Java programs. By default, Volta's target processor is JOP, hardware specifically designed for running Java for real time embedded applications [67].

Volta is easy to use and already supports the concept of pluggable strategies for various parts of the WCET calculations. Using a new cost based strategy along with other minor changes to the framework proved to be a very reasonable approach for implementing a prototype of the static analysis approach. Adapting another WCET system for Java, like the one described in [68], might also be useful in the future.

The WCET adaptation calculates the max monetary cost of a program starting at a root method. It also finds cost excessive paths and determines path independent normalized node weights for the nodes in the input program's DCG.

The implementation of the prototype relies heavily on the control flow analysis and execution time features of the WCET system. It builds a DCG during the WCET/cost analysis, and it calculates and saves base cost and max times values for the nodes in the DCG. It then uses the algorithms developed in the static analysis approach to determine path costs and node weights.

Verification Utility

A verification utility was implemented to test the prototype. Whereas the research prototype performs static analysis on an input program, the verification utility executes programs and measures costs dynamically.

The verification utility first generates random programs by producing a random DCG. Though the DCG is random, it is bounded by several constants in the verification utility such as maximum number of children nodes for a given node and maximum program call stack depth. These are designed to be within reason for actual programs while keeping the size of the random programs amenable to producing and analyzing many programs. For example, the average call stack depth was 5 and the average number of children nodes was 3. Calder studied object oriented programs (in this case written in C++) and found an average call stack depth of 13 (see [18]). An average number of children call graph nodes of 2 can also be derived from Calder's data. The stack depth is shorter in order to keep analysis times lower, but the average number of children is slightly higher in order to ensure reasonable diversity in paths.

The verification utility produces Java source code from the random DCG. In addition to the methods and conditions which make up the DCG, the Java source code includes loops (randomly selected DCG nodes are surrounded by loops). Furthermore, methods are chosen at random to be annotated with the various resource costs (bandwidth, service, and storage usage).

It is not uncommon to use generated programs to verify static analysis tools. In [25], refactoring engines are tested using generated programs. An approach for testing an optimizing compiler using randomly generated programs is described in [60].

Costs/rates for each of the different cloud resources (processing, storage, bandwidth) are fixed randomly for each program within appropriate bounds. The cost

threshold for each random program is also generated randomly and within a reasonable bound.

The Java source code produced by the verification utility introduces some limitations. The DCGs on which they are built contain no cycles and children nodes always have exactly one parent. The bodies of decision branches always consist of exactly one method, though method bodies contain a random (but bounded) number of method calls. Also, loop bodies always consist of exactly one DCG node (either decision or method call), and the loops themselves always execute all the way to their annotated bounds. If this were not the case, it would be difficult to use the random programs to verify the accuracy of the research prototype. On the other hand, it may better simulate actual variations in runtime conditions, depending on the particular application and domain.

The root method in the generated source accepts a random integer as input. This integer is used in the condition of each decision in the program. Each decision randomly selects a bit position to mask and test. When the bit is set, the *then* branch is taken otherwise the *else* branch is taken. This allows for a set of randomly generated inputs to produce an extensive variety of control flows.

The overall idea is to generate enough random programs and execute the random programs with enough random inputs so that a credible list of paths and calculated nodes weights can be built. Both the verification utility and the particular structure of the random programs themselves allow for capturing the cost of a particular execution of the program on a given input as well as the other essential information for each node (base cost and max times) required for the additional calculations developed here.

If the cost of a path (i.e. one execution of the program for a given random input) exceeds the predetermined (but random) threshold, then the additional

calculations developed here are performed to determine the path independent normalized node weight for each node that appears somewhere in a cost excessive path.

The verification utility also orchestrates the entire process of producing the random DCG, converting it to Java source code, compiling the Java source code, then executing it on random inputs. All of this was accomplished without much trouble using Java features including dynamic compilation as well as extensive use of the Reflection API.

The goal for the verification utility is to produce two main pieces of data that can be compared with corresponding data produced by the static analysis performed in the research prototype. First, the cost excessive paths found by the verification utility should always be a subset of those found by the research prototype. Second, the path independent normalized node weight values of the verification utility should generally be very close to those of the research prototype.

5.7 Evaluation Results

To ensure diversity in testing inputs for the research prototype, 1000 random programs were generated by the verification utility. As discussed earlier, the programs varied in size and complexity within certain bounds. For each random program, 10000 random inputs were generated and provided as input to the program. The paths and node weights determined by the verification utility were captured and saved. The program was then analyzed by the research prototype and the output data were then compared to the data saved from the random test runs of the program.

All cost excessive paths found by the verification utility were also found by the research prototype. This met the expectation that the cost excessive paths from the verification would be a subset of those found by the research prototype.

Of course, this does not fully guarantee that the research prototype is correct in this aspect, since there was no analysis of the cost excessive paths identified by the prototype but not by the verification utility. It is assumed that those paths (if any) were simply not found by any of the random test runs. However, future work could perform further analysis on these paths to verify (independent of the research prototype itself, since it is the component under test here) that they are indeed cost excessive.

A close correlation was found between the path independent normalized node weights of the verification utility and the research prototype. For those nodes with a nontrivial weight (i.e. at least 0.0001), the median weight difference between the verification utility and research prototype was just 4.2%. This demonstrates that the prototype is meeting its objectives in this aspect and making its calculations accurately.

Looking through various test data and the source code of randomly generated programs, some anecdotal observations regarding the verification utility were made. First, when all cost excessive paths are found by test runs in the verification utility, differences in node weights from the research prototype are most likely due to execution time measurements and are generally very small. More significant differences in node weights are probably introduced when some cost excessive paths are not found by the verification utility.

Also, highly weighted nodes can often be predicted simply by scanning the source code (though the absolute and relative magnitude is not so easily predicted). Those nodes that have annotated resource costs, those that are called within loops, and those with ancestor methods called within loops are almost always the nodes that are weighted more heavily.

It should be noted, however, that this does not trivialize the importance of the research prototype since real world programs can be very large and complex.

Statically traversing source code and describing cost excess can be a very nontrivial undertaking, one which is very likely to be prohibitive if attempted manually.

5.8 Assumptions and Limitations

The tools and algorithms in the static analysis approach leverage the significant body of research on WCET analysis which has been developed mostly over the past 10-15 years. The WCET problem is very important for real-time and safety critical systems, but it is also very difficult. The limitations of the work developed here are mostly carried over from the limitations of WCET analysis. Fortunately, ongoing work in WCET is addressing these limitations. Following is a discussion of the limitations, including how they are being addressed and why research into the limitations is outside the scope of the current research.

A basic WCET implementation provides a single upper bound for a function. A more advanced possibility (not yet available) might be to assign a range of values and perhaps a probability associated with each range. A parametric solution would include a variable upper bound using function inputs as parameters. Recent parametric timing research has produced some basic results [16]. However, these produce very complex parametric representations that need to be simplified in order to be useful. Addressing this and other problems is literally the work of another dissertation (namely, the dissertation of Stefan Bygde of Malardalen University in Sweden) and is outside the scope of this dissertation.

The implication for this dissertation is that the kinds of applications that can benefit from it will be those that have fixed limits on data sizes (consumption of memory, storage, and bandwidth) and therefore have a fixed cost that can be determined and compared against a threshold instead of a variable threshold depending on inputs. Such applications are likely to charge users a flat fee for a single use (or a fixed number of uses). An example of a fixed rate SaaS application would be a stock transaction processing system (pay-per-trade).

WCET also has some limitations significant to cloud applications which are more likely to be built using modern programming languages. These limitations include dealing with recursive methods and exception handling (which significantly complicate control flow) as well as polymorphism (which requires advanced analysis to determine type information statically when such a determination is even possible). Historically, these have not been too limiting for WCET since real time systems have been implemented using low level languages or even assembly (which do not support polymorphism and exception handling), or have accepted limitations in order to guarantee safety (in the case of recursive methods). However, as higher level languages are increasingly used in real time systems, these limitations become more restrictive. Therefore, it is likely that there will be more research in these areas in the coming years.

One of the more prominent WCET tools for Java has some support for polymorphism using data flow analysis (specifically receiver type analysis) [68]. This provides some narrowing of possible virtual method invocations, but static analysis cannot always determine the type of an object and therefore cannot determine exactly which virtual method implementation will be invoked at run time.

Some research into WCET in the presence of exception handling has taken place [21]. It is difficult because it increases (and to some extent obfuscates) the control paths that must be considered. None of the main WCET tools for Java currently deal with exception handling. This is mitigated by the common use of exception handling as error handling which generally limits-not expands-the flow of control through the program and therefore would generally not increase the upper timing bound.

Recursive methods complicate possible control flow even more. The extensive work of Blieberger provides good evidence for this [10]. There are various special cases of recursive methods including indirect recursion, multiple entry and exit points

to a cycle of recursive methods in a control flow graph, overlapping recursive cycles, and multiple recursive calls within a single method. Complete analysis of these and other issues related to recursion would be enough material for a separate dissertation and if included here would likely be a distraction from the main hypothesis of this dissertation.

Another assumption involves lower level hardware concerns, which are sometimes abstracted away from the developer in cloud computing. In WCET analysis (and in the static analysis approach by extension), a tight bound on processing time can be derived when developers have intimate knowledge of the target processor. For example, the Volta tool requires developers to specify the number of cycles used for basic Java bytecode instructions. Default values are provided but they can be customized based on the actual target processor. For this dissertation, the default values were left in place, and a processor speed of 4GHz was assumed. However, this processor speed can easily be customized. Some cloud providers do publish CPU speeds and they differentiate between the speeds in their pricing schemes [56].

5.9 Observations

While the research prototype for the static analysis approach has delivered accurate results, these results were derived from input programs that were constructed so as to avoid the limitations discussed earlier. These limitations are quite severe. A sample program (one of the shorter programs) that was analyzed is shown in Appendix A. While this program and the others that were analyzed represent various possible call graphs of varying sizes, they do not represent real world Java programs well. An approach that could handle more standard Java programs would be important to consider.

Chapter 6

Hybrid Approach

6.1 Overview

The static analysis approach discussed previously delivered concrete results and calculations for determining cost excessive paths in a cloud application. However, the limitations associated with that approach are severe. The sample programs constructed as part of the verification of the static analysis approach were tailored to the limitations, but a real world example (much less a real world production system) would not be handled by the tools and research in its current state.

An approach based more on the dynamic approaches discussed earlier could handle a real world example. This chapter describes such an approach. It is based on the instrumentation approach to dynamic measurement but it involves calculations that are similar to those developed for the static analysis approach. Hence, it is called a hybrid approach.

The hybrid approach will also take operational profiles into account. Operational profiles are one way of representing expected or observed usage information into cost calculations. At a fundamental level, an operational profile is simply a set of inputs and a probability that the set will be executed for a given run of the program (for a more complete explanation see [57] and [28]). This data fits nicely into the hybrid approach. The probability informs weighting calculations so that when an input from a certain operational profile is chosen, the corresponding costs are weighted with the probability.

Note that because this approach gathers and analyzes aggregate usage data, it answers questions related to overall averages. Whereas the static analysis approach looked at individual control flow paths, the hybrid approach uses probabilities to present an overall picture of cost excess.

The hybrid approach allows helps to answer questions such as those listed below:

- On average, how much profit (or loss) is expected per transaction?
- Which operational profiles are most likely to exceed a cost threshold?
- Which operational profiles contribute most to cost excess?
- Which methods contribute most to cost excess?

Following is a more detailed description of the hybrid approach including the calculations that can be made with it. Also provided is a real world example application that demonstrates the use of the hybrid approach.

6.2 Dynamic Measurements

The hybrid approach is a basic form of the dynamic instrumentation approach discussed earlier. For each operational profile, the following measurements are taken on a set of test runs:

- When resources are requested, the cost is added to the running total for that method.
- At the end of each run, a cost for each method for the given transaction has been calculated.
- Adding these costs together, an overall cost for the given transaction is determined (each transaction is associated with the operational profile that contains its inputs).

Measurements for running time (CPU) are handled in a similar way both on a method and transaction level. Running the software on the CPU is not exactly the

same as requesting a resource like bandwidth, but it is the same in that it is consuming a fundamental cloud resource. CPU usage is added to the cost of a method or transaction before summary calculations are made, so it is treated the same as other cloud resources.

If the measurements are taken locally (off the cloud) the CPU times would not be quite as accurate as if they were run on the cloud unless local hardware is nearly identical to the hardware on instances reserved in the cloud.

With these basic pieces of data, summary calculations can be performed similar to those used in the static analysis approach. These calculations are discussed in the following section.

6.3 Summary Calculations

From the dynamic measurements the following calculations can be made given a cost threshold (THR), for an operation profile (P) which involves some number of method calls (M) and is part of a complete set of operational profiles (S).

$$
\begin{aligned}
TotalCost(P) &= \text{total measured cost for all runs of P} \\
NumRuns(P) &= \text{the number of runs of P} \\
AverageCost(P) &= \frac{TotalCost(P)}{NumRuns(P)} \\
AverageCostExcess(P) &= \frac{AverageCost(P) - THR}{THR}
\end{aligned}
$$

(Average cost excess is expressed here as a percentage of cost excess.)

Since the operational profile includes a probability, it is used to "weight" the profile as well the methods invoked during transactions under that profile. Hence, a weighed cost for the operational profile is calculated as follows:

$$
Weight(P) = AverageCostExcess(P) \times Probability(P)
$$

The following calculations are specific to methods invoked during the test runs of an operational profile. First, calculate a method's contribution to the operational profile under which it was invoked, then use that operational profile's weight to produce a weighting for the method. This weighting is ultimately the contribution for this path to the overall weighting of the method.

$$\begin{aligned} TotalCost(M,P) &= \text{total measured cost for M for all runs of P} \\ AverageContribution(M,P) &= \frac{TotalCost(M,P)}{TotalCost(P)} \\ Weight(M,P) &= Weight(P) \times AverageContribution(M,P) \end{aligned}$$

The overall cost excess for all operational profiles can be derived based on their weighted cost excesses. Furthermore, the normalized weight for a given profile takes all other profile weights into account. It represents the impact or importance the operational profile plays in relation to cost excess.

$$\begin{aligned} OverallCostExcess &= \sum_{P \in S} Weight(P) \\ NormalizedWeight(P) &= \frac{Weight(P)}{OverallCostExcess} \end{aligned}$$

Likewise, by summing the weight of a particular method for each of the operational profiles in which it is invoked, the full weight can be calculated (already weighted appropriately from each operational profile weight). Comparing the weight against the overall cost excess allows for determining the normalized node weight. This is the ultimate determination of which methods contribute most to cost excess.

$$\begin{aligned} FullWeight(M) &= \sum_{P \in S} Weight(M,P) \\ NormalizedWeight(M) &= \frac{FullWeight(M)}{OverallCostExcess} \end{aligned}$$

6.4 Using the Hybrid Approach

To briefly summarize how one would use the hybrid approach to characterize cost excess, think of it in terms of measurements, calculations, and results. First, measure

the cost of resources consumed by a test run of the program (as described above), and associate each cost with the method it was called from.

Next, provide these measurements as input to the summary calculations (also described above). Each of the results of these calculations provides an answer to one of the questions posed earlier, as follows:

- On average, how much profit (or loss) is expected per transaction?
  - The OverallCostExcess gives exactly the loss per transaction.
- Which operational profiles are most likely to exceed a cost threshold?
  - The AverageCostExcess(P) for each profile indicates whether that profile exceeds the threshold (on average) or not.
- Which operational profiles contribute most to cost excess?
  - The contribution of operational profiles to cost excess is measured in terms of NormalizedWeight(P) for each profile. A higher weight indicates a higher relative contribution.
- Which methods contribute most to cost excess?
  - The contribution of methods to cost excess is measured in terms of NormalizedWeight(M) for each method. A higher weight indicates a higher relative contribution.

6.5 Evaluation Methodology

The hybrid approach was evaluated by implementing a real world example program and taking measurements on test runs of that program. The summary calculations discussed previously were performed on the measurements taken during test runs, and the resulting data were analyzed for correctness.

Real World Example

An implementation of the real estate image service (discussed earlier in the dissertation) was implemented as a real world example. As discussed earlier, this example takes an address as input and produces a high quality image of the real estate at that address as output. To accomplish this work, the application first converts the address to a geolocation (latitude and longitude), then queries one or more image providers for a detailed image at that location. The images are analyzed by an algorithm that decides what image will best represent the real estate at that location.

The image providers offer satellite, aerial, or street level images via a SaaS interface. The real estate image service detects the type of real estate at a given geolocation (e.g. single family house, high rise condo, acreage, etc.) based on location and/or based on an initial high level image scan. It then requests the right kind of image for the real estate type from the appropriate image provider. For example, a one acre lot may look best on an aerial image, while a single family house may look better in a street level image.

The example application used to verify the hybrid approach is a prototype implementation of the real estate image service to serve as a proof of concept for the hybrid approach. Several pieces of the application only simulate what a real application would do. For example, latency is simulated with image providers via Thread.sleep calls, and the images themselves are simulated with default constructed images of various sizes.

A UML diagram of the real estate image service is provided in Figure 6.1.

The design makes use of two main types of components: directors and handlers. A director in this application is a class that encapsulates the procedural logic of the application along with associated algorithms.

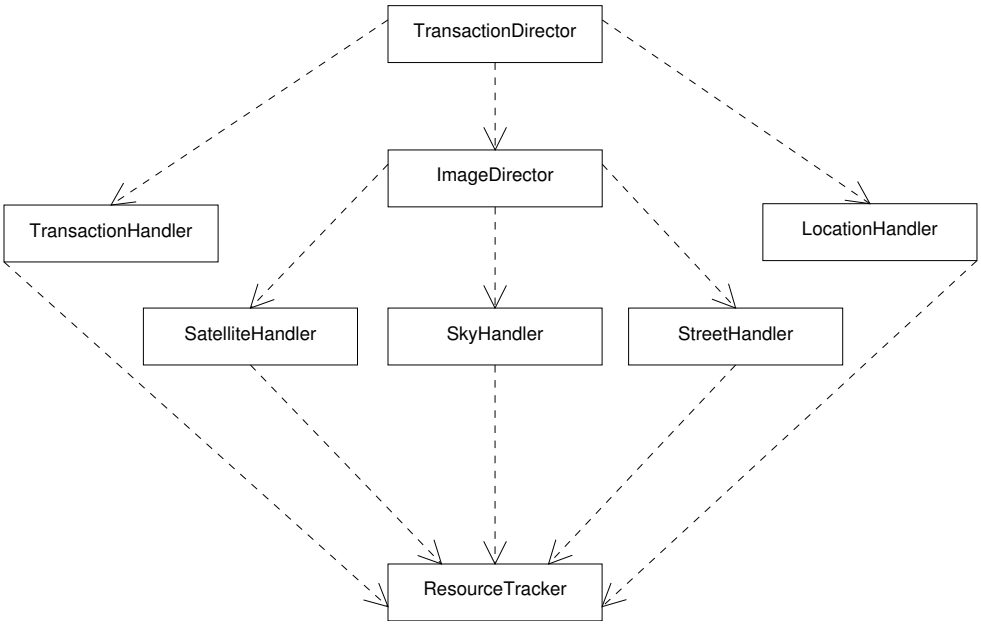

Figure 6.1: Real Estate Service Design

The TransactionDirector is the top level class that implements the overall logic of receiving the address and converting it to a geolocation, handing off work to the ImageDirector and getting a result, then sending this result as output. The ImageDirector encapsulates the core algorithm of the application. It works with handlers associated with each image provider to fetch and analyze images as needed.

A handler in this application is a class that handles all protocol responsibilities for sending and receiving data from outside services and clients. There are handlers for each image provider as well as for the service that converts from an address to a geolocation. There is also a handler for communication with clients of the real estate image service.

Each of the handlers uses bandwidth and outside services. Bandwidth is used for sending inputs and receiving outputs to/from the outside service. These resources are accounted for by the ResourceTracker. The ResourceTracker implements the instrumentation approach to dynamic measurements and is discussed in greater detail below.

The real estate image service is clearly a pay-per-use composite service. Customers are charged for each image (or perhaps group of images) that the service produces. Likewise, the service itself uses outside services to obtain raw images and for other parts of the application.

See Appendix B for some sample classes from the real estate image service. Note in particular how they are a much better representation of an actual Java program than are the test programs for the static analysis program, an example of which is provided in Appendix A.

Measurements and Calculations

Similar to the dynamic instrumentation approach discussed earlier, a single method call was inserted in each of the handlers to record usage of a cloud resource. For the real estate image service, this included tracking bandwidth and service costs. The method using the resource was detected via Java stack inspection utilities, and resource usage was recorded by simply adding the usage to the running total for the detected method. The recorded values were serialized to a file for all the runs of a particular operational profile (and later deserialized to perform summary calculations).

To determine the CPU cost of each method as well as the full transaction as a whole, java heap profiling was utilized. This is enabled by adding "-Xrunhprof:cpu=times" as an argument to the Java virtual machine. It produces an hprof file (heap profile) which is then parsed for information about each method executed during the transaction. (Each method's running time is recorded and stored in text format in the hprof file.)

To simulate latency associated with working with outside services, each handler inserted some amount of time to the execution via a Thread.sleep call. Hence, they were always ranked at the top of methods requiring the most CPU time.

Constant (but configurable) values for the bandwidth cost per byte, the CPU cost per millisecond, and the overall cost threshold were also specified in the ResourceTracker. Operational profile probabilities were similarly specified. The cost of invoking an outside service was accounted for at the point of usage since it was specific to the particular service requested. Total transaction costs were calculated based on these low level cloud resource costs as has been discussed previously.

The final step of the process was to deserialize operational profile specific data combined with hprof information and aggregate the data to determine overall results. As discussed in the section on calculations, the number of test runs was recorded along with the resource cost for each invocation of a method as well as the total for the entire transaction (test run). These were summed for each operational profile. Those data formed the basis for the remaining calculations. The results that were collected are discussed in the following section.

6.6 Evaluation Results

To demonstrate the effectiveness of the hybrid approach (implemented in the ResourceTracker) several scenarios were explored for using the real estate image service. Normal (expected) values and test runs produce normal/expected results. A "normal" test run and result is one in which each of the operational profiles produces different costs but all of them fall below the threshold and none of them has cost excess. All tests were run locally (off the cloud) since running on the cloud would not provide any significant benefit from an evaluation perspective.

After observing expected results in the normal case, some cost excesses were seeded in the application and it was demonstrated that the hybrid approach correctly identifies them and the resulting overall cost excess in its summary calculations. This is very similar to standard mutation testing (see [5] and [6]), though instead of using generic mutation operators, specific and targeted mutations to produce deterministic mutants were used.

Indeed, the seeding was done specifically at points in the program where resources were used by simply multiplying the existing resource usage/request by a relatively large factor to be sure cost excesses would result. Since the test program was not CPU intensive, cost excess seeding was necessarily biased toward spots where bandwidth and outside services were used. Increasing the resource usage in these spots ensured that cost excesses would result, and the overall goal of the seeding was to show that the measurement framework (which of course was not modified during seeding) would find the associated cost excesses.

Finally, the calculations in the hybrid approach were used to explore different scenarios to improve cost performance. The initial image processing algorithm was compared to a possible improvement, and the resulting summary calculations inform decisions about why the improvement would or would not produce a better cost profile.

An initial set of test runs of the real estate image service with resource tracking and summary calculations produced expected results. Given the following constants (as well as reasonable values for image sizes and service latencies), none of the operational profiles produced average costs above the threshold.

- Transaction cost threshold: $0.005
- Bandwidth cost: $0.12/GB
- CPU cost: $0.12/hour
- Satellite image service: $0.001 per use
- Street image service: $0.002 per use
- Aerial image service cost: $0.003 per use
- Large lot profile probability: 0.05

- Small lot profile probability: 0.05
- House with large lot profile probability: 0.10
- House with standard lot profile probability: 0.50
- Condo/Townhouse (within small clustered buildings) profile probability: 0.20
- Condo/Townhouse (within high rise building) profile probability: 0.10

The summary calculations for normal usage showed that the overall cost excess was negative (the overall average transaction cost did not exceed the threshold) and the individual cost excesses for each operational profile were also negative (none of the average costs for an individual profile exceeded the threshold).

Even though the threshold was not exceeded, method and operational profile weights were still calculated and showed that the retrieval of satellite and street level images contributed most to cost. This is likely due to the image director (image processing algorithm) which always requests an initial but low cost satellite image and requests a street level image for houses (the most common operational profile).

After demonstrating standard usage of the applications, cost excesses (through manual insertion) were then seeded in specific locations in order to produce deterministic cost excesses in the final calculations. This allowed us to verify that the resource tracker and summary calculations were accurately and consistently detecting cost excessive profiles and methods.

First, the cost of the aerial image service (which affects operational profiles for which those images are a best fit) was doubled. An additional fifty redundant transmissions of the final resulting image were also injected, which would affect all operational profiles. This latter cost excess seeding was to ensure that the hybrid approach would show the correct method containing elevated cost excess.

After these cost excesses were seeded, the expected results were observed. In the first case, when aerial image costs were doubled, the operational profiles that became cost excessive were for smaller lots of land, condos in small clustered buildings, and houses with relatively large lots, each of which uses an aerial image. In the second case with extra transmissions to the client, most operational profiles became cost excessive, but the particular method in the transaction handler to send the final resulting image was flagged as contributing most significantly to cost excess. Hence the resource tracker's calculations were able to successfully find the seeded cost excesses.

The calculations were also used to make decisions about the internals of the software. For example, the image director's algorithm always retrieves a satellite image as the first step, and decides from that image which kind of real estate is at the location. However, since street images are used for common operational profiles (e.g. standard house), it may be cheaper to instead request the street image and send it as the result, if it is available. (If it is not available, the algorithm would fall back to requesting the satellite image as usual.) The image director was modified to implement this new approach and ran the ResourceTracker to calculate results.

After executing the test runs for the original algorithm and for the modified algorithm, the results of the summary calculations for each algorithm were compared. Costs had indeed shifted and that the profiles that do not benefit from the optimization for street images became cost excessive. Furthermore, the overall average cost was still barely below the threshold, but still somewhat higher than the overall average cost for the original algorithm. With this information, the modified algorithm would not be chosen. This demonstration shows that the hybrid approach can also inform decisions about implementation concerns.

6.7 Assumptions and Limitations

The hybrid approach assumes that operational profile data are readily available. Without these data, the calculations at the core of this approach would not be possible. However, some scaled back information and calculations might be possible without using operational profile data (e.g. overall cost excess without weighting, cost excessive methods but also without weighting information).

As with any approach based on dynamic analysis, the data and calculations are only as accurate as the test runs are representative of real world conditions. If the verification phase involves tests that accurately simulate the types of transactions users will initiate, then the approach will be successful in indicating where cost excess will show up. However, if the inputs in the tests do not represent real world conditions well, then the cost data will not help.

Furthermore, the hybrid approach is particularly subject to the risk of not finding cost excessive methods or profiles that are executed so infrequently that they are not included in any test suite. These might be found by a robust implementation of a static analysis approach (if/when such an approach could overcome the many limitations to which static analysis approaches are subject!).

6.8 Observations

The hybrid approach is fairly easy to use and produces accurate results. With the implementation developed here, the developer must add some instrumentation to the source code (a more complicated but feature rich implementation could free the developer from manually adding instrumentation). However, the developer is not guessing at loop or bandwidth bounds as in the static analysis approach–those can be observed dynamically.

Operational profiles lend further credibility to the calculations and increases accuracy. Separating test runs by operational profiles and determining their overall

weight is informative. Ultimately, this information flows down to the methods which are more accurately weighted for determining sources of cost excess.

The hybrid approach could be limited by the test runs that are observed. As long as these test runs are representative of actual usage, the dynamic analysis will be useful. However, there is some risk that all types of usages of the software cannot be covered, and possibly some important ones. This may leave some dark corners of the application unexplored by the hybrid approach and only detectable by something with more extensive static analysis.

Chapter 7

Conclusions and Future Work

7.1 Conclusions

Finding cost excesses in cloud applications can be an important part of software verification. It is a type of nonfunctional testing that is particularly important for cloud applications precisely because of the benefits of the cloud: resource elasticity and a pay-as-you-go economic model. Without a clear understanding of how cloud applications incur costs, developers increase their risk of scaling up resource usage to meet high demand while possibly losing money on important and costly transactions.

This dissertation has presented several possibilities for verifying that cost objectives are met and for determining when costs may exceed a given threshold. As explained earlier, the hypothesis of this dissertation is as follows:

Cost excesses in cloud applications can be characterized before deployment.

This dissertation has shown the development of a static analysis approach for determining paths within a cloud application that can exceed a predetermined cost threshold. These paths, along with the most important/costly nodes within a program's interprocedural control flow graph can be brought to the attention of developers before an application is deployed. Using a research prototype and independent verification utility, it was determined that the static analysis approach found each path identified by the verification utility and only differed from the utility in summary calculations by a small amount. Hence, the prototype demonstrates the accuracy of the static analysis approach.

By building on the research and tools of worst case execution time analysis, the static analysis approach inherits all the benefits of a well established field. On the other hand, it also inherits the limitations of that field which still exist but which are

also being addressed over time. The limitations are nontrivial and include restrictions on commonly used programming language facilities. The approach also assumes advanced knowledge of bandwidth and possible storage usage, but only at the point of usage within a particular method of the cloud application.

This dissertation has also shown the development of several dynamic analysis methods that could be practical and potentially accurate. They may find a representative subset of the cost excessive methods in a cloud application, depending on how representative the tests are that are used to perform the analysis. They do not suffer from many of the limitations of the static analysis approach but also do not have some of the benefits of static analysis.

Static analysis does not depend so heavily on test suites finding all possible cost excesses. The advantage of static analysis is that it examines the entire program and therefore will find all cost excesses, even those in the "dark corners" of an application that are less commonly used.

A hybrid approach was developed and represents a pragmatic way to achieve reasonably accurate results in finding cost excesses while avoiding the limitations of the purely static analysis approach. This approach is mostly dynamic but its calculations are based on those developed for the static analysis approach. These calculations incorporate operational profile data to enhance their accuracy.

Overall, the results of the experiments conducted here point toward affirmation of the hypothesis. The work here represents an attempt to open the door into predeployment monetary cost analysis of cloud applications, and it is safe to say that such an analysis is possible and that the paths and/or methods in cloud applications that are cost excessive can indeed be identified and described. The major contributions of this dissertation can be summarized as follows:

1. Based on its significant benefits and modest limitations, a cost verification approach based mostly on dynamic analysis would be useful for determining and characterizing cost excesses. An approach based on instrumenting cloud applications can be easy to implement (with added features requiring additional effort) and can be useful for providing important insights to developers.

2. Based on its significant limitations, the static analysis approach developed and explored in this dissertation would require further research to be useful in a production environment. As related fields (WCET, energy aware computing) progress and their needs converge with those of cloud cost verification, the static analysis approach may become more immediately applicable as it will inherit the advances in those fields and overcome current limitations.

Several types of cloud application can benefit from this dissertation. Scientific workflows often involve data intensive transactions which may be costly. Business and consumer application developers are likely to be particularly sensitive to costs in order to maximize profits. Cloud services, particularly those that operate under a pay-per-use model, will likely benefit most from the cost verification presented here.

7.2 Future Work

This dissertation opens the door to many possible future paths of research. Though outside the scope of this dissertation and generally somewhat orthogonal to the research in this dissertation, the potential value of these future paths is at least highlighted by the work here.

It may be interesting to further explore a provider based approach to dynamic cost measurements. This would be the ideal approach from the cloud user's perspective, not only because it results in the least work for the user but also because the cloud provider has more immediate access to the cloud resources and can more easily account for their usage at a very fine level. Of course, the cloud provider must

already provide trustworthy resource accounting, but the incentive to provide the very low level accounting discussed in this dissertation has not previously presented itself.

It would be interesting to modify an existing cloud implementation like Eucalyptus [59] to record and make available fine grain resource usage measurements. Indeed, the technology to non-invasively and verifiably provide these metrics could be an entirely separate research direction altogether. This could encourage cloud providers to take on the value added functionality of fine grain resource accounting.

This dissertation assumes that a cost threshold has already been fixed. However, an extension of this dissertation may be to help cloud application developers decide on a cost based on the various resource usage scenarios along the various control flow paths in an application. Furthermore, building a resource usage profile could also provide a means for choosing a cloud provider. Since providers have different cost profiles for different resource types, matching a resource usage profile with a cost profile could allow for cost minimization.

It might be interesting to build simulation models of applications in order to derive cost usage models and/or determine resource usage bounds and probabilities. These could be associated with the previously mentioned work of minimizing costs by considering resource usage profiles along with the various cloud provider cost profiles. Simulation models working under different cost profiles could provide important data for making decisions about which cloud provider to use.

It may be even more interesting to determine a general approach for cost simulation modeling for cloud applications. Though clearly outside the scope of this dissertation, this could be a fruitful area of research and could be beneficial to cloud application cost verification.

The idea of resource usage profiles that are matched to a lowest cost cloud provider gives rise to another important question that could be explored in future

work. Are there parts of an application that should not be run on the cloud? For example, if a resource usage profile (or possibly one of the weighting schemes described in this dissertation) shows that a resource is in such high demand within an application, and if the application is run frequently enough, it may be more cost effective to provide dedicated, in house resources instead of using those available on the cloud. Indeed, it may be possible that a resource usage profile would lead to a decision not to run an application on the cloud at all.

Since economic concerns are a key component of the entire notion of cloud computing, deciding if and when to run applications or components on the cloud (from an economic standpoint) as well as verifying that running applications or components on the cloud will indeed be cost effective are both essential for developers to understand when planning the deployment of software applications.

[1] S. Agarwala, D. Jadav, and L. Bathen. Icostale: Adaptive cost optimization for storage clouds. In *Cloud Computing, IEEE International Conference on (CLOUD)*, July 2011.

[2] Amazon. Amazon web services, Dec. 2010.

[3] Amazon. Amazon cloudwatch, Feb. 2012.

[4] Amazon. Amazon ec2 pricing, June 2012.

[5] J. Andrews, L. Briand, and Y. Labiche. Is mutation an appropriate tool for testing experiments? [software testing]. In *Software Engineering, 2005. ICSE 2005. Proceedings. 27th International Conference on*, pages 402 – 411, May 2005.

[6] J. Andrews, L. Briand, Y. Labiche, and A. Namin. Using mutation analysis for assessing and comparing testing coverage criteria. *Software Engineering, IEEE Transactions on*, 32(8):608 –624, 2006.

[7] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, and M. Zaharia. A view of cloud computing. *Commun. ACM*, 53:50–58, April 2010.

[8] X. Bai, M. Li, B. Chen, W.-T. Tsai, and J. Gao. Cloud testing tools. In *Service Oriented System Engineering (SOSE), 2011 IEEE 6th International Symposium on*, pages 1 –12, dec. 2011.

[9] R. Bala and S. Carr. Usage-based pricing of software services under competition. *Journal of Revenue and Pricing Management*, 9(3):204–216, 2010.

[10] J. Blieberger. Real-time properties of indirect recursive procedures. *Inf. Comput.*, 171:156–182, January 2002.

[11] B. Boehm. Value-based software engineering: reinventing. *SIGSOFT Softw. Eng. Notes*, 28(2):3–, Mar. 2003.

[12] K. Buell and J. Collofello. Transaction level economics of cloud applications. *Services, IEEE Congress on*, 0:515–518, 2011.

[13] K. Buell and J. Collofello. Cost excessive paths in cloud based services. In *Information Reuse and Integration (IRI), 2012 IEEE International Conference on*, aug. 2012.

[14] K. Buell and J. Collofello. Dynamic cost verification for cloud applications. In *Proceedings of the 2012 Workshop on Dynamic Analysis*, WODA 2012, pages 18–23, New York, NY, USA, 2012. ACM.

[15] R. Buyya, C. S. Yeo, S. Venugopal, J. Broberg, and I. Brandic. Cloud computing and emerging it platforms: Vision, hype, and reality for delivering computing as the 5th utility. *Future Gener. Comput. Syst.*, 25:599–616, June 2009.

[16] S. Bygde and B. Lisper. Towards an automatic parametric wcet analysis. In R. Kirner, editor, *8th Intl. Workshop on Worst-Case Execution Time (WCET) Analysis*, Dagstuhl, Germany, 2008.

[17] H. Cai, K. Zhang, M. Wang, J. Li, L. Sun, and X. Mao. Customer centric cloud service model and a case study on commerce as a service. In *Cloud Computing, 2009. CLOUD '09. IEEE International Conference on*, pages 57 –64, sept. 2009.

[18] B. Calder, D. Grunwald, and B. Zorn. Quantifying behavioral differences between c and c++ programs. *JOURNAL OF PROGRAMMING LANGUAGES*, 2:313–351, 1994.

[19] R. N. Calheiros, R. Ranjan, A. Beloglazov, C. A. F. De Rose, and R. Buyya. Cloudsim: a toolkit for modeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms. *Softw. Pract. Exper.*, 41(1):23–50, Jan. 2011.

[20] W. Chan, L. Mei, and Z. Zhang. Modeling and testing of cloud applications. In *Services Computing Conference, 2009. APSCC 2009. IEEE Asia-Pacific*, pages 111 –118, 2009.

[21] R. Chapman, A. Burns, and A. Wellings. Worst-case timing analysis of exception handling in ada. In *Proceedings of the Ada UK Conference*, pages 148–164. IOS Press, 1993.

[22] J. Chen and Y. Yang. A taxonomy of grid workflow verification and validation. *Concurr. Comput. : Pract. Exper.*, 20(4):347–360, Mar. 2008.

[23] H. K. Cheng and G. J. Koehler. Optimal pricing policies of web-enabled application services. *Decis. Support Syst.*, 35(3):259–272, June 2003.

[24] M. A. Cusumano. The changing labyrinth of software pricing. *Commun. ACM*, 50(7):19–22, July 2007.

[25] B. Daniel, D. Dig, K. Garcia, and D. Marinov. Automated testing of refactoring engines. In *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, ESEC-FSE '07, pages 185–194, New York, NY, USA, 2007. ACM.

[26] E. Deelman, G. Singh, M. Livny, B. Berriman, and J. Good. The cost of doing science on the cloud: The montage example. In *High Performance Computing, Networking, Storage and Analysis, 2008. SC 2008. International Conference for*, pages 1 –12, 2008.

[27] N. Dun, K. Taura, and A. Yonezawa. Paratrac: a fine-grained profiler for data-intensive workflows. In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, HPDC '10, pages 37–48, New York, NY, USA, 2010. ACM.

[28] M. Gittens, H. Lutfiyya, and M. Bauer. An extended operational profile model. In *Software Reliability Engineering, 2004. ISSRE 2004. 15th International Symposium on*, pages 314 – 325, 2004.

[29] D. Gmach, J. Rolia, and L. Cherkasova. Resource and virtualization costs up in the cloud: Models and design choices. In *Dependable Systems Networks (DSN), 2011 IEEE/IFIP 41st International Conference on*, pages 395 –402, june 2011.

[30] Google. Google app engine, Dec. 2010.

[31] Google. Google app engine - pricing and features, June 2012.

[32] I. Gorton, J. Chase, A. Wynne, J. Almquist, and A. Chappell. Services + components = data intensive scientific workflow applications with medici. In *Proceedings of the 12th International Symposium on Component-Based Software Engineering*, CBSE '09, pages 227–241, Berlin, Heidelberg, 2009. Springer-Verlag.

[33] T. W. Harmon. *Interactive worst-case execution time analysis of hard real-time systems*. PhD thesis, University of California, Irvine, 2009.

[34] T. A. Henzinger, A. V. Singh, V. Singh, T. Wies, and D. Zufferey. A marketplace for cloud resources. In *Proceedings of the tenth ACM international conference on Embedded software*, EMSOFT '10, pages 1–8, New York, NY, USA, 2010. ACM.

[35] Z. Hou, X. Zhou, J. Gu, Y. Wang, and T. Zhao. Asaas: Application software as a service for high performance cloud computing. In *High Performance Computing and Communications (HPCC), 2010 12th IEEE International Conference on*, pages 156 –163, sept. 2010.

[36] A. Ishii and T. Suzumura. Elastic stream computing with clouds. In *Cloud Computing, IEEE International Conference on (CLOUD)*, July 2011.

[37] R. Jayaseelan, T. Mitra, and X. Li. Estimating the worst-case energy consumption of embedded software. In *Proceedings of the 12th IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 81–90, Washington, DC, USA, 2006. IEEE Computer Society.

[38] A. Khajeh-Hosseini, D. Greenwood, and I. Sommerville. Cloud migration: A case study of migrating an enterprise it system to iaas. In *Cloud Computing (CLOUD), 2010 IEEE 3rd International Conference on*, pages 450 –457, 2010.

[39] T. M. King, A. S. Ganti, and D. Froslie. Enabling automated integration testing of cloud application services in virtualized environments. In *Proceedings of the 2011 Conference of the Center for Advanced Studies on Collaborative Research*, CASCON '11, pages 120–132, Riverton, NJ, USA, 2011. IBM Corp.

[40] M. Klems, J. Nimis, and S. Tai. Do clouds compute? a framework for estimating the value of cloud computing. In W. Aalst, J. Mylopoulos, N. M. Sadeh, M. J. Shaw, C. Szyperski, C. Weinhardt, S. Luckner, and J. Ster, editors, *Designing E-Business Systems. Markets, Services, and Networks*, volume 22 of *Lecture Notes in Business Information Processing*, pages 110–123. Springer Berlin Heidelberg, 2009.

[41] H. Kllapi, E. Sitaridi, M. M. Tsangaris, and Y. Ioannidis. Schedule optimization for data processing flows on the cloud. In *Proceedings of the 2011 international conference on Management of data*, SIGMOD '11, pages 289–300, New York, NY, USA, 2011. ACM.

[42] P. Koehler, A. Anandasivam, and D. Ma. Cloud Services from a Consumer Perspective. In *Proceedings of the 16th Americas Conference on Information Systems (AMCIS)*, Lima, Peru, 2010.

[43] D. Kossmann, T. Kraska, and S. Loesing. An evaluation of alternative architectures for transaction processing in the cloud. In *Proceedings of the 2010 international conference on Management of data*, SIGMOD '10, pages 579–590, New York, NY, USA, 2010. ACM.

[44] P. Kudtarkar, T. F. DeLuca, V. A. Fusaro, P. J. Tonellato, and D. P. Wall. Cost-effective cloud computing: A case study using the comparative genomics tool, roundup. *Evolutionary bioinformatics online*, 6:197–203, 2010.

[45] U. Lampe, T. Mayer, J. Hiemer, D. Schuller, and R. Steinmetz. Enabling cost-efficient software service distribution in infrastructure clouds at run time. In *Service-Oriented Computing and Applications (SOCA), 2011 IEEE International Conference on*, pages 1 –8, dec. 2011.

[46] P. Y. Lau, S. Park, J. Yoon, and J. Lee. Pay-as-you-use on-demand cloud service: An iptv case. In *Electronics and Information Engineering (ICEIE), 2010 International Conference On*, volume 1, pages V1–272 –V1–276, aug. 2010.

[47] J. Y. Lee, J. W. Lee, D. W. Cheun, and S. D. Kim. A quality model for evaluating software-as-a-service in cloud computing. In *Software Engineering Research, Management and Applications, 2009. SERA '09. 7th ACIS International Conference on*, pages 261 –266, dec. 2009.

[48] Y. C. Lee, C. Wang, A. Y. Zomaya, and B. B. Zhou. Profit-driven service request scheduling in clouds. In *Cluster, Cloud and Grid Computing (CCGrid), 2010 10th IEEE/ACM International Conference on*, pages 15 –24, May 2010.

[49] S. Lehmann and P. Buxmann. Pricing strategies of software vendors. *Business Information Systems Engineering*, 1(6):452–462, 2009.

[50] X. Li, Y. Li, T. Liu, J. Qiu, and F. Wang. The method and tool of cost analysis for cloud computing. In *Proceedings of the 2009 IEEE International Conference on Cloud Computing*, CLOUD '09, pages 93–100, Washington, DC, USA, 2009. IEEE Computer Society.

[51] K. Liu, H. Jin, J. Chen, X. Liu, D. Yuan, and Y. Yang. A compromised-time-cost scheduling algorithm in swindew-c for instance-intensive cost-constrained workflows on a cloud computing platform. *International Journal of High Performance Computing Applications*, 24(4):445–456, 2010.

[52] W. Lu, J. Jackson, J. Ekanayake, R. S. Barga, and N. Araujo. Performing large science experiments on azure: Pitfalls and solutions. In *Proceedings of the 2010 IEEE Second International Conference on Cloud Computing Technology and Science*, CLOUDCOM '10, pages 209–217, Washington, DC, USA, 2010. IEEE Computer Society.

[53] B. Martens, M. Walterbusch, and F. Teuteberg. Costing of cloud computing services: A total cost of ownership approach. In *System Science (HICSS), 2012 45th Hawaii International Conference on*, pages 1563 –1572, jan. 2012.

[54] L. Mei, Z. Zhang, and W. Chan. More tales of clouds: Software engineering research issues from the cloud application perspective. In *Computer Software and Applications Conference, 2009. COMPSAC '09. 33rd Annual IEEE International*, volume 1, pages 525 –530, july 2009.

[55] Microsoft. Windows azure platform, Dec. 2010.

[56] Microsoft. Windowsazure pay-as-you-go, June 2012.

[57] J. Musa. Operational profiles in software-reliability engineering. *Software, IEEE*, 10(2):14 –32, Mar. 1993.

[58] V. Nallur and R. Bahsoon. Design of a market-based mechanism for quality attribute tradeoff of services in the cloud. In *Proceedings of the 2010 ACM Symposium on Applied Computing*, SAC '10, pages 367–371, New York, NY, USA, 2010. ACM.

[59] D. Nurmi, R. Wolski, C. Grzegorczyk, G. Obertelli, S. Soman, L. Youseff, and D. Zagorodnov. The eucalyptus open-source cloud-computing system. In *Proceedings of the 2009 9th IEEE/ACM International Symposium on Cluster Computing and the Grid*, CCGRID '09, pages 124–131, Washington, DC, USA, 2009. IEEE Computer Society.

[60] M. H. Palka, K. Claessen, A. Russo, and J. Hughes. Testing an optimising compiler by generating random lambda terms. In *Proceeding of the 6th international workshop on Automation of software test*, AST '11, pages 91–97, New York, NY, USA, 2011. ACM.

[61] S. Pandey, A. Barker, K. Gupta, and R. Buyya. Minimizing execution costs when using globally distributed cloud services. In *Advanced Information Networking and Applications (AINA), 2010 24th IEEE International Conference on*, pages 222 –229, Apr. 2010.

[62] K.-W. Park, S. K. Park, J. Han, and K. H. Park. Themis: Towards mutually verifiable billing transactions in the cloud computing environment. In *Cloud Computing (CLOUD), 2010 IEEE 3rd International Conference on*, pages 139 –147, 2010.

[63] J. S. Rellermeyer, M. Duller, and G. Alonso. Engineering the cloud from software modules. In *Proceedings of the 2009 ICSE Workshop on Software Engineering Challenges of Cloud Computing*, CLOUD '09, pages 32–37, Washington, DC, USA, 2009. IEEE Computer Society.

[64] L. Riungu, O. Taipale, and K. Smolander. Research issues for software testing in the cloud. In *Cloud Computing Technology and Science (CloudCom), 2010 IEEE Second International Conference on*, pages 557 –564, 30 2010-dec. 3 2010.

[65] P. Robinson and C. Ragusa. Taxonomy and requirements rationalization for infrastructure in cloud-based software testing. In *Cloud Computing Technology and Science (CloudCom), 2011 IEEE Third International Conference on*, pages 454 –461, December 2011.

[66] T. Rostrom and C.-C. Teng. Secure communications for pacs in a cloud environment. In *Engineering in Medicine and Biology Society,EMBC, 2011 Annual International Conference of the IEEE*, pages 8219 –8222, 30 2011-sept. 3 2011.

[67] M. Schoeberl. *JOP: A Java Optimised Processor for Embedded Real-Time Systems*. PhD thesis, Vienna University of Technology, 2005.

[68] M. Schoeberl, W. Puffitsch, R. U. Pedersen, and B. Huber. Worst-case execution time analysis for a java processor. *Softw. Pract. Exper.*, 40:507–542, May 2010.

[69] V. Sekar and P. Maniatis. Verifiable resource accounting for cloud computing services. In *Proceedings of the 3rd ACM workshop on Cloud computing security workshop*, CCSW '11, pages 21–26, New York, NY, USA, 2011. ACM.

[70] T. Shibata, S. Choi, and K. Taura. File-access patterns of data-intensive workflow applications and their implications to distributed filesystems. In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, HPDC '10, pages 746–755, New York, NY, USA, 2010. ACM.

[71] G. Singh, G. Garg, P. Jain, and H. Singh. Article: The structure of cloud engineering. *International Journal of Computer Applications*, 33(8):44–49, November 2011. Published by Foundation of Computer Science, New York, USA.

[72] S. Tai, J. Nimis, A. Lenk, and M. Klems. Cloud service engineering. In *Software Engineering, 2010 ACM/IEEE 32nd International Conference on*, volume 2, pages 475 –476, may 2010.

[73] V. Tosic, H. Wada, A. Guabtni, K. Lee, and A. Liu. Management towards reducing cloud usage costs. In *Network Operations and Management Symposium (LANOMS), 2011 7th Latin American*, page 1, oct. 2011.

[74] H.-L. Truong and S. Dustdar. Composable cost estimation and monitoring for computational applications in cloud computing environments. *Procedia Computer Science*, 1(1):2175 – 2184, 2010. ICCS 2010.

[75] H.-L. Truong and S. Dustdar. Cloud computing for small research groups in computational science and engineering: current status and outlook. *Computing*, 91(1):75–91, Jan. 2011.

[76] T. Truong Huu, G. Koslovski, F. Anhalt, J. Montagnat, and P. Vicat-Blanc Primet. Joint elastic cloud and virtual network framework for application performance-cost optimization. *J. Grid Comput.*, 9(1):27–47, Mar. 2011.

[77] K. Tsakalozos, H. Kllapi, E. Sitaridi, M. Roussopoulos, D. Paparas, and A. Delis. Flexible Use of Cloud Resources through Profit Maximization and Price Discrimination. In *Proc. of the 27th IEEE Int. Conf. on Data Engineering (ICDE'11)*, Hannover, Germany, Apr. 2011.

[78] M. Vouk. Cloud computing–issues, research and implementations. In *Information Technology Interfaces, 2008. ITI 2008. 30th International Conference on*, pages 31 –40, 2008.

[79] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, and P. Stenström. The worst-case execution-time problem–overview of methods and survey of tools. *ACM Trans. Embed. Comput. Syst.*, 7:36:1–36:53, May 2008.

[80] J. Wu, C. Wang, Y. Liu, and L. Zhang. Agaric–a hybrid cloud based testing platform. In *Cloud and Service Computing (CSC), 2011 International Conference on*, pages 87 –94, dec. 2011.

[81] S. Yau and H. An. Software engineering meets services and cloud computing. *Computer*, 44(10):47 –53, oct. 2011.

[82] C. S. Yeo, S. Venugopal, X. Chu, and R. Buyya. Autonomic metered pricing for a utility computing service. *Future Gener. Comput. Syst.*, 26(8):1368–1380, Oct. 2010.

[83] L. Youseff, M. Butrico, and D. Da Silva. Toward a unified ontology of cloud computing. In *Grid Computing Environments Workshop, 2008. GCE '08*, pages 1 –10, 2008.

[84] J. Yu and R. Buyya. A taxonomy of scientific workflow systems for grid computing. *SIGMOD Rec.*, 34:44–49, September 2005.

[85] L. Zhang, X. Ma, J. Lu, T. Xie, N. Tillmann, and P. de Halleux. Environmental modeling for automated cloud application testing. *Software, IEEE*, 29(2):30 –35, march-april 2012.

APPENDIX A

SAMPLE GENERATED PROGRAM FOR STATIC ANALYSIS

```
import test.*;
import edu.uci.eecs.doc.clepsydra.cost.*;
import edu.uci.eecs.doc.clepsydra.loop.*;
public class RandomCode{
public static final double processingCost_ = 0.23;
public static final double outgoingBandwidthCost_ = 0.06;
public static final double incomingBandwidthCost_ = 0.05;
public static final double storageAccessCost_ = 0.0096;
public static final double storageMonthlyCost_ = 0.01;
public static final double costThreshold_ = 0.08;
public DCGNodeInfo[] M(Integer i)
{for (int c = 0; c < nodeInfos.length; ++c)
{nodeInfos[c]=new DCGNodeInfo();nodeInfos[c].maxTimes=0;}
M0(i, 1);return nodeInfos;}

public void M0(int i, int maxTimes){
nodeInfos[0].id = 0;
nodeInfos[0].baseCost = 0.0;
nodeInfos[0].maxTimes = maxTimes;
M1(i, maxTimes * 1);
M20(i, maxTimes * 1);
if((i & (1 << 19)) != 0){
M22(i, maxTimes * 1);
}
else{
M23(i, maxTimes * 1);}
}

public void M1(int i, int maxTimes){
nodeInfos[1].id = 1;
nodeInfos[1].baseCost = 0.0;
nodeInfos[1].maxTimes = maxTimes;
@LoopBound(max=8)
for (int j = 0; j < 8; ++j){
if((i & (1 << 1)) != 0){
M2(i, maxTimes * 8);
}
else{
M17(i, maxTimes * 8);}
}
}

public void M2(int i, int maxTimes){
nodeInfos[2].id = 2;
nodeInfos[2].baseCost = 0.0;
nodeInfos[2].maxTimes = maxTimes;
M3(i, maxTimes * 1);
M4(i, maxTimes * 1);
```

```
M11(i, maxTimes * 1);
M16(i, maxTimes * 1);
}

@IncomingBandwidthBound(maxBytes=19147486, maxDelay=1001608507)
public void M3(int i, int maxTimes){
nodeInfos[3].id = 3;
nodeInfos[3].baseCost = 9.556159525605805E-4;
nodeInfos[3].maxTimes = maxTimes;
}

@OutgoingBandwidthBound(maxBytes=30533291, maxDelay=563657498)
public void M4(int i, int maxTimes){
nodeInfos[4].id = 4;
nodeInfos[4].baseCost = 0.001742192042397328;
nodeInfos[4].maxTimes = maxTimes;
M5(i, maxTimes * 1);
M6(i, maxTimes * 1);
M7(i, maxTimes * 1);
M8(i, maxTimes * 1);
M9(i, maxTimes * 1);
M10(i, maxTimes * 1);
}

public void M5(int i, int maxTimes){
nodeInfos[5].id = 5;
nodeInfos[5].baseCost = 0.0;
nodeInfos[5].maxTimes = maxTimes;
}

public void M6(int i, int maxTimes){
nodeInfos[6].id = 6;
nodeInfos[6].baseCost = 0.0;
nodeInfos[6].maxTimes = maxTimes;
}

public void M7(int i, int maxTimes){
nodeInfos[7].id = 7;
nodeInfos[7].baseCost = 0.0;
nodeInfos[7].maxTimes = maxTimes;
}

@ServiceBound(maxCost=0.0076849871343693, maxDelay=537097828)
public void M8(int i, int maxTimes){
nodeInfos[8].id = 8;
nodeInfos[8].baseCost = 0.007719301717824856;
nodeInfos[8].maxTimes = maxTimes;
}
```

```
public void M9(int i, int maxTimes){
nodeInfos[9].id = 9;
nodeInfos[9].baseCost = 0.0;
nodeInfos[9].maxTimes = maxTimes;
}

public void M10(int i, int maxTimes){
nodeInfos[10].id = 10;
nodeInfos[10].baseCost = 0.0;
nodeInfos[10].maxTimes = maxTimes;
}

public void M11(int i, int maxTimes){
nodeInfos[11].id = 11;
nodeInfos[11].baseCost = 0.0;
nodeInfos[11].maxTimes = maxTimes;
M12(i, maxTimes * 1);
M13(i, maxTimes * 1);
M14(i, maxTimes * 1);
M15(i, maxTimes * 1);
}

public void M12(int i, int maxTimes){
nodeInfos[12].id = 12;
nodeInfos[12].baseCost = 0.0;
nodeInfos[12].maxTimes = maxTimes;
}

public void M13(int i, int maxTimes){
nodeInfos[13].id = 13;
nodeInfos[13].baseCost = 0.0;
nodeInfos[13].maxTimes = maxTimes;
}

public void M14(int i, int maxTimes){
nodeInfos[14].id = 14;
nodeInfos[14].baseCost = 0.0;
nodeInfos[14].maxTimes = maxTimes;
}

public void M15(int i, int maxTimes){
nodeInfos[15].id = 15;
nodeInfos[15].baseCost = 0.0;
nodeInfos[15].maxTimes = maxTimes;
}

public void M16(int i, int maxTimes){
```

```
nodeInfos[16].id = 16;
nodeInfos[16].baseCost = 0.0;
nodeInfos[16].maxTimes = maxTimes;
}

public void M17(int i, int maxTimes){
nodeInfos[17].id = 17;
nodeInfos[17].baseCost = 0.0;
nodeInfos[17].maxTimes = maxTimes;
M18(i, maxTimes * 1);
@LoopBound(max=41)
for (int j = 0; j < 41; ++j){
M19(i, maxTimes * 41);
}
}

public void M18(int i, int maxTimes){
nodeInfos[18].id = 18;
nodeInfos[18].baseCost = 0.0;
nodeInfos[18].maxTimes = maxTimes;
}

public void M19(int i, int maxTimes){
nodeInfos[19].id = 19;
nodeInfos[19].baseCost = 0.0;
nodeInfos[19].maxTimes = maxTimes;
}

public void M20(int i, int maxTimes){
nodeInfos[20].id = 20;
nodeInfos[20].baseCost = 0.0;
nodeInfos[20].maxTimes = maxTimes;
@LoopBound(max=8)
for (int j = 0; j < 8; ++j){
M21(i, maxTimes * 8);
}
}

public void M21(int i, int maxTimes){
nodeInfos[21].id = 21;
nodeInfos[21].baseCost = 0.0;
nodeInfos[21].maxTimes = maxTimes;
}

public void M22(int i, int maxTimes){
nodeInfos[22].id = 22;
nodeInfos[22].baseCost = 0.0;
nodeInfos[22].maxTimes = maxTimes;
```

```
}

public void M23(int i, int maxTimes){
nodeInfos[23].id = 23;
nodeInfos[23].baseCost = 0.0;
nodeInfos[23].maxTimes = maxTimes;
}

public DCGNodeInfo[] nodeInfos = new DCGNodeInfo[24];
}
```

APPENDIX B

SAMPLE CLASSES FOR DYNAMIC ANALYSIS

```
package realimage.directors;

import java.awt.Image;

import cloud.resources.ResourceTracker;

import realimage.handlers.LocationHandler;
import realimage.handlers.TransactionHandler;


public class TransactionDirector
{
  public static void main(String[] args)
  {
    for (String address : args)
    {
      ResourceTracker.beginTransaction();

      TransactionHandler transactionHandler = new TransactionHandler();

      LocationHandler locationHandler = new LocationHandler();
      String geoLocation = locationHandler.getGeolocation(address);

      ImageDirector imageDirector = new ImageDirector();
      Image result = imageDirector.getImage(geoLocation);

      transactionHandler.sendResult(result);

      ResourceTracker.endTransaction();
    }

    ResourceTracker.saveStats();
  }
};
```

```
package realimage.handlers;

import java.awt.Image;
import java.awt.image.BufferedImage;

import cloud.resources.ResourceTracker;


public class SatelliteHandler
{
  public SatelliteHandler(){}

  public Image getImage(String geolocation)
  {
    ResourceTracker.recordServiceUsage(.001);
    ResourceTracker.recordBandwidthUsage(50000);

    try
    {
      Thread.sleep(100);
    }
    catch (InterruptedException e)
    {
      e.printStackTrace();
    }

    return new BufferedImage(500, 100, BufferedImage.TYPE_3BYTE_BGR);
  }
};
```

```
package realimage.directors;

import java.awt.Image;

import realimage.handlers.SatelliteHandler;
import realimage.handlers.SkyHandler;
import realimage.handlers.StreetHandler;


public class ImageDirector
{
  public static enum RealEstateType{
    LARGE_LOT(.05),
    SMALL_LOT(.05),
    LARGE_LOT_HOUSE(.10),
    STANDARD_HOUSE(.50),
    CLUSTERS(.20),
    HIGH_RISE(.10);

    final public double probability;
    RealEstateType(double probability)
    {
      this.probability = probability;
    }
  }

  public ImageDirector(){}

  // Encodes real estate image selection and processing algorithm
  public Image getImage(String geolocation)
  {
    // Look at sat first and determine real estate type
    SatelliteHandler sat = new SatelliteHandler();
    Image satImage = sat.getImage(geolocation);
    RealEstateType type = getRealEstateType(geolocation, satImage);

    // if larger lot, then keep sat imagery
    if (type == RealEstateType.LARGE_LOT)
    {
      return satImage;
    }

    // if smaller lot, large lot house, or clusters,
    // then try for sky (if none, then try street, then sat)
    if (type == RealEstateType.SMALL_LOT ||
        type == RealEstateType.LARGE_LOT_HOUSE ||
        type == RealEstateType.CLUSTERS)
    {
```

```
      SkyHandler sky = new SkyHandler();
      Image skyImage = sky.getImage(geolocation);
      return skyImage != null ? skyImage : satImage;
    }

    // if standard house or high rise,
    // then try for street level (if none then sat)
    if (type == RealEstateType.STANDARD_HOUSE ||
        type == RealEstateType.HIGH_RISE)
    {
      StreetHandler street = new StreetHandler();
      Image streetImage = street.getImage(geolocation);
      return streetImage != null ? streetImage : satImage;
    }

    // unknown image types return sat image
    return satImage;
  }

  private RealEstateType getRealEstateType(
    String geolocation, Image satelliteImage)
  {
    // Simulate real estate type detection (just lookup type)
    return RealEstateType.values()[Integer.parseInt(geolocation) / 2];
  }
};
```