Partitioning and Distribution of Web Applications to the Hybrid Cloud

by

Nima Kaviani

B.Sc., Bu-Ali Sina University, 2005
M.Sc., Simon Fraser University, 2007

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF DOCTOR OF PHILOSOPHY

in

The Faculty of Graduate and Postdoctoral Studies

(Computer Science)

THE UNIVERSITY OF BRITISH COLUMBIA

(Vancouver)
August 2014
© Nima Kaviani 2014
Abstract

Hybrid cloud deployment is an effective strategy in deploying software services across public cloud and private infrastructure. It allows deployed software systems to benefit from cost savings and scalability offerings of the cloud while keeping control over privacy- or security-sensitive code and data entities. However, the complexity of determining which code and data entities should reside on-premises, and which can be migrated to the cloud is daunting. Researchers have attempted to address this complexity by using partitioning algorithms to optimize distribution and deployment of code entities across public cloud and private infrastructure. However, we have identified the following shortfalls with the existing research work:

- Current research does not provide enough flexibility in placement of software function execution and data entities between public/private hosts. In particular it does not allow for replication or optimized separation of code and data entities in relation to one another.

- Current research on partitioning of software systems does not explicitly consider the dynamics of a hybrid cloud deployment when making decisions about public cloud and private infrastructure. Particularly, current research lacks support for making explicit tradeoffs between monetary cost and improved performance in hybrid cloud software systems.

- The dynamics of the cloud require partitioning algorithms to be tailored towards features inherent to a hybrid cloud deployment. This includes encoding data dependency models and component dependency models of a software system collectively into one unique mathematical optimization model. There is no existing algorithm that allows for combined code and data dependency requirements to be modelled under one optimization formula.

This thesis presents my work on implementing algorithms and tools that address the shortcomings of the previous research as discussed above. These algorithms and tools are put together under a partitioning and distribution
Abstract

framework named MANTICORE. MANTICORE has been used to drive partitioning and deployment decisions on several open source software systems. The experiment results show an estimate of up to 54% reduction in monetary costs compared to a premises only deployment and 56% improvement in performance compared to a naïve separation of code entities from data entities in a hybrid cloud deployment.
Preface

This dissertation is original, unpublished, independent work by the author, Nima Kaviani.

The contributions and evaluations presented in this dissertation are summarized and published in three conference papers, namely: IEEE Conference on Service Oriented Computing 2011 (ICSOC 2011) [66], IEEE Cloud Computing Conference 2012 (CloudCom 2012) [67] and Usenix Middleware Conference 2013 [68].
Table of Contents

Abstract .......................................................... ii
Preface .............................................................. iv
Table of Contents ................................................... v
List of Tables ....................................................... viii
List of Figures ....................................................... x
List of Programs ..................................................... xiii
Acknowledgements .................................................. xiv
Dedication ........................................................... xvi

1 Introduction ....................................................... 1
  1.1 Deploying Applications to Hybrid Cloud .................... 2
  1.2 Problem Statement ............................................ 4
  1.3 Methodology .................................................. 7
  1.4 Contributions ................................................ 11

2 Related Work .................................................... 14
  2.1 Code Mobility & Distributed Middleware .................. 15
  2.2 Application Partitioning ..................................... 19
  2.3 Software Migration and Resource Utilization ............. 30
  2.4 Summary ..................................................... 36

3 Code Dependency Modeling and Application-tier Partitioning ......................................................... 37
  3.1 Creating the Dependency Graph .............................. 38
  3.2 Modeling Code Components in the Dependency Graph .... 39
     3.2.1 Request-based Model (RBM) ............................ 40
### Table of Contents

3.2.2 Static Structure Model (SSM) ........................................ 42
3.2.3 Context-Sensitive Model (CSM) ................................. 43
3.3 Defining Constraints for Placement & Cost ......................... 46
  3.3.1 Policy Specification Scheme ..................................... 46
  3.3.2 Cost Scheme .................................................... 47
3.4 Augmenting Dynamic Dependency Graphs with Cost Metrics .... 47
3.5 Applying the Partitioning Algorithm ............................... 50
  3.5.1 The Partitioning Algorithm for Symmetric Data Exchange Costs .................................................. 50
  3.5.2 The Partitioning Algorithm for Asymmetric Data Exchange Costs .................................................. 52
3.6 Evaluation .............................................................. 54
  3.6.1 Micro-Benchmarks on Performance Improvements .......... 57
  3.6.2 Cost Models vs Measured Deployments ....................... 62
  3.6.3 Evaluation of Context-Sensitive Modeling .................. 64
  3.6.4 Evaluation of Flexible Cost Modeling ......................... 66
  3.6.5 Evaluation of Scalability with Code-only Partitioning ...... 68
  3.6.6 Evaluation of Deployment Costs ............................... 70
  3.6.7 Evaluating Symmetric vs. Asymmetric Partitioning Algorithms .................................................. 73
  3.6.8 Evaluating Data Entity Placement Constraints ............. 75
3.7 Summary ...................................................................... 76

4 Data Dependency Modeling and Data-tier Partitioning .......... 79
  4.1 Extending Application-tier Partitioning with Data-tier Partitioning .................................................. 83
  4.2 High-Level Overview of Data-tier Partitioning ................. 86
  4.3 Profiling the Data-tier with EXPLAIN PLAN .................... 90
  4.4 Analyzing Dependencies for the Data-tier ....................... 92
  4.5 Partitioning the Data-tier .............................................. 95
    4.5.1 Defining BIP Constraints for Data Dependencies ........ 96
    4.5.2 Defining the BIP Objective for Data Dependencies .......... 97
  4.6 Evaluation .............................................................. 102
    4.6.1 Evaluation of Performance .................................... 104
    4.6.2 Evaluation of Deployment Costs .............................. 107
    4.6.3 Evaluation of Scalability ...................................... 109
  4.7 Summary ...................................................................... 112
# Table of Contents

5 Manticore:  
The Partitioning Framework .................................. 114  
5.1 Manticore Overview ..................................... 115  
5.2 The Software Profiling Tool ............................. 118  
5.3 The Tool for Analysis and Generating the Dynamic Dependency Graph .................................. 120  
5.4 The Tool for Partitioning the Dynamic Dependency Graph .................................. 121  
5.5 Distribution and Deployment ............................ 123  
5.6 Discussion .............................................. 125  
5.7 Summary ................................................. 126  

6 Conclusion & Future Work .................................. 128  
6.1 Primary Research Contributions ......................... 129  
  6.1.1 Context-Sensitive Dependency Modeling ......................... 129  
  6.1.2 Flexible Cost Modeling ................................ 129  
  6.1.3 Cross-tier Partitioning ................................ 130  
  6.1.4 Asymmetric Data Exchange Costs ......................... 130  
6.2 Secondary Research Contributions ....................... 131  
  6.2.1 Simulation and Real-world Evaluation ...................... 131  
  6.2.2 Comprehensive Tooling ................................ 131  
6.3 Thesis Review and Observations .......................... 131  
6.4 Challenges and Future Work ............................... 132  
  6.4.1 Semi-automatic Partitioning ......................... 132  
  6.4.2 Heterogeneous vs. Homogeneous Machines .............. 133  
  6.4.3 Multi-way and Online Partitioning ...................... 133  
  6.4.4 Data Dependency Analysis ............................. 134  
  6.4.5 Security and Privacy Requirements ...................... 135  
6.5 Summary ................................................. 135  

Bibliography .................................................. 137  

Appendices  
A Appendix on Hybrid Cloud Solutions .................... 146  
B Appendix on Dynamic Dependency Graph ................ 149  
C Appendix on Cost and Policy Specifications for Partitioning .................................. 151  
  C.1 Policy Specification ..................................... 151  
  C.2 Cost Schema ............................................ 152
# List of Tables

2.1 Hilda’s Characteristics .............................................. 21
2.2 SWIFT’s Characteristics ............................................. 22
2.3 Wishbone’s Characteristics ......................................... 23
2.4 J-Orchestra’s Characteristics ....................................... 24
2.5 Characteristics of the approach by Diaconescu et al. ........... 24
2.6 Characteristics of the approach by Gu et al. ..................... 25
2.7 Characteristics of the approach by Ou et al. ..................... 26
2.8 Coign’s Characteristics .............................................. 27
2.9 ABACUS’s Characteristics ............................................ 28
2.10 Characteristics of the approach by Nanda et al. ................ 29
2.11 Characteristics of the approach by Giurgiu et al. .......... ... 30
2.12 Dimensions of Investigation for application partitioning ap- ... 31
3.1 Symbols used in application trace of data dependency graph . 39
3.2 The number of nodes in the DDG of DayTrader as well as the ... 56
3.3 Microbenchmarks of Database and RPC roundtrips for JFor- ... 59
3.4 Microbenchmarks on the amount of data exchanged between ... 62
3.5 Code placement for different request types in Apache Day- ... 67
3.6 Deployment costs for DayTrader with changes in constrained ... 77
4.1 Constraint generation functions ................................. 97
4.2 Functions for generating objective helper constraints .......... 98
4.3 Functions for generating objective function ..................... 99
4.4 Naming conventions for deployment models used in evaluation 103
4.5 Response time evaluation of the Cross-Tier deployment compared to other models of deployment (a) for doLogin and (b) across all business logic functionality. Results show how a deployment based on cross-tier partitioning improves or degrades response time compared to the other models of deployment. ................................................................. 107

4.6 Cost improvements of Cross-Tier deployment compared to other models of deployment. Results show how a deployment based on cross-tier partitioning improves deployment costs compared to the other models of deployment. .................. 109
List of Figures

1.1 Hybrid deployment architecture with distributed code and data. 7
1.2 Highlevel representation of our methodology involving the steps of profiling, analysis, and partitioning. 8
1.3 Cross-tier partitioning of a monolithic web application. 11

3.1 Application Dependency Models: a) RBM, b) SSM, c) CSM. Circles are application nodes and cylinders are database tables. The edge weights represent number of round-trip data exchanges between nodes. 41
3.2 CSM visualization of partitioning results for doLogin. 45
3.3 DDG and data exchange for the sub-graph of doAccount in DayTrader. 54
3.4 A high-level model of how different partitioning algorithms would choose component placements between the public and the private cloud. 58
3.5 Inbound and Outbound data exchanges for Naïve Hybrid versus partitioned deployment models. 61
3.6 Simulated and measured execution times for hybrid and cloud code placements for each DayTrader request type. 68
3.7 Simulated and measured data transfer sizes for hybrid and cloud code placements for each DayTrader request type. 64
3.8 Comparison of latency adjustments for the SSM, RBM, and CSM as the premises cost changes. 65
3.9 Monetary cost of deploying requests of various type for DayTrader with respect to changes in Gamma ($\gamma$). 68
3.10 Scalability tests for full premises, full cloud, and hybrid deployments. 69
3.11 Comparison of monthly deployment costs for different deployment models of DayTrader. 71
3.12 Comparison of monthly deployment costs for different deployment models of JForum. 72
# List of Figures

3.13 Measured data exchanged between the cloud and premises using the Symmetric vs. Asymmetric partitioning algorithms.  
3.14 Cloud-to-premise ratio of module placements.  

4.1 A cross-tier partitioning suggested by our tool for the `doLogin` request from DayTrader.  
4.2 Possible query plans from one of the queries in DayTrader.  
4.3 The high-level steps in performing data-tier partitioning.  
4.4 The query plan tree representing alternatives in executing a given query joining three tables A, B, and C with one another.  
4.5 The first phase in partitioning the data-tier is to profile the data-tier with EXPLAIN PLAN.  
4.6 The second phase in partitioning the data-tier is to analyze dependencies in a query plan.  
4.7 The third phase in partitioning the data-tier is to encode the partitioning problem into a BIP formulation.  
4.8 Measured execution times for selected request types in the four deployments of DayTrader.  
4.9 Measured execution times for selected request types in the four deployments of RUBiS.  
4.10 Monthly cost comparison for different deployments of DayTrader and RUBiS.  
4.11 Monthly cost comparison for different deployments of RUBiS.  
4.12 Scalability tests for full premises, full cloud, and hybrid deployments for DayTrader.  
4.13 Scalability tests for full premises, full cloud, and hybrid deployments for RUBiS.  

5.1 Profiling, Analysis, and Partitioning in MANTICORE  
5.2 The screen shot for the eclipse launcher of the Java profiler in MANTICORE.  
5.3 The screen shot for selecting the profiling trace, the policy constraints, and the host configuration constraints in MANTICORE.  
5.4 The screen shot for selecting the execution cost model, the communication cost model, the partitioning algorithm, and applying partitioning in MANTICORE.  
5.5 The screen shot for the results of partitioning and the generated distribution plan in MANTICORE.
5.6 The screen shot for statistical information on partitioning results in MANTICORE.

125
List of Programs

4.1 Function to collect statistics for alternative query plan operators for the input query $Q$. $P_i$ is the powerset operator over sets of size $i$. 95
4.2 Constraint generation for cross-tier partitioning. 97
4.3 Objective generation 99

B.1 A sample profiling trace collected by jip-osgi. 150

C.1 A sample policy specification filtering on given $DDG$. 152
C.2 An example cost scheme provided to the profiler. 154
Acknowledgements

I am honoured and delighted to acknowledge the many people whose counsel, support, and encouragement have contributed immensely to the completion of my dissertation.

First and foremost I would like to thank my amazing supervisors Dr. Rodger Lea and Dr. Eric Wohlstadter for their constant help, support, and advice throughout my PhD. When I started my PhD, I had this assumption that it would be an easy ride from the start to the end. However, it did not take long for me to realize that PhD is not just a scientific adventure but more importantly it is a life adventure. It was in the first year of my PhD that I had to change the direction for my research. My passion was always in distributed systems, however if it wasn’t for the help and support of Eric and Rodger, I would have probably ended up in a completely different field of research. It was my great pleasure to work with the two of them and benefit from their insight, experience, and help not only as research supervisors but more importantly as role models in life.

I also need to thank the very many great friends from the MAGIC lab and the Software Practices lab who helped me with their constant advice, support, encouragement, and feedback:

I would like to thank Dr. Michael Blackstock for being an amazing friend and a great advisor throughout my PhD. He was the best counselor for me to talk to when making the early career decisions for my PhD.

My great level of appreciation also goes to Dr. Thomas Fritz for being an amazing friend. My friendship with Thomas very quickly grew beyond a simple work relation and we developed bonds to the point that I consider him one of my closest friends in life. Thomas has always been a wonderful source of energy, encouragement, and positive vibe. Particularly whenever I was low on steam, he constantly pushed me forward with his encouraging words. I consider myself very lucky to get to know him as a great individual.
Acknowledgements

and I hope I can return his favours at some point in life.

I also immensely appreciate the help and support I received from Dr. Sarah Rastkar, Dr. Aiman Erbad, Dr. Matthias Finke, Dr. Ebrahim Bagheri, Reza Babanezhad, Roberto Calderon, Vincent Tsao, Albert Thompson, and Julius Davis.

My deepest levels of gratitude also go to my amazing friend and brother Dr. Bardia Mohabbati who made a significant mark in my life with his presence, support, and encouragement. Being roommates for five years and still being best friends speaks a world about the level of trust and friendship we developed. I am very honoured to have him as my friend. Thanks very much Bardia for all the adventures we shared together, for all the all-nighters we pulled together, and for all the ideas we bounced off each other.

Last but not least, I would like to thank my beautiful and amazing friend Mona Erfani. Mona walked alongside me from the very starting point of my PhD career to the very last minute that I heard the word “pass” in the day of my PhD defence. Without her there would have been less meaning to life, to joy, to fun, and to friendship for me. Thanks Mona for all the help, for all the support, for your kind heart, and for your trustful manner. Stay amazing like this!
Dedication

To the three stars of my life:

“my dad, my mom, and my beautiful sister ...”
Chapter 1

Introduction

Cloud computing is a model for enabling convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction. The benefits of cloud computing are in providing i) on-demand elastic computing resources, ii) pay-per-use billing models, and iii) minimal up-front user commitments [38]. All these benefits have made cloud computing an attractive technology to be employed and used by businesses.

In early 2011, the Information Week Analytics survey [50] revealed growing interests in using cloud computing resources among companies, from 31% in 2009 to 46% in 2010. The trend continued to 56% in 2011 [29], 67% in 2012, and 75% in 2013 [30]. In a separate research survey, Gartner predicts a cloud computing service revenue growth from $131 billion in 2013 [52] to $180 billion by the end of 2015 [31].

Despite the benefits of public cloud in providing increased flexibility at lower costs, the idea of a complete migration of software systems to the cloud is not fully embraced by cloud customers. Issues such as operational challenges, data compliance requirements, data or architectural lock-in to a particular provider, and security and privacy concerns are among the major obstacles preventing a full application migration to the cloud [32, 55, 107]. This is to the extent that a recent study by RightScale[1] identifies privacy concerns and data compliance as the top most fears among IT managers when considering a full migration to the cloud [87]. To mitigate some of these challenges, companies have turned into a new architectural model, referred to as Hybrid Cloud.

In essence, Hybrid Cloud [38, 93, 97] is an architectural model in which computation and storage capacities from a public cloud are offered as supple-

---

[1]: http://www.rightscale.com/
1.1. Deploying Applications to Hybrid Cloud

The deployment of resources to a private infrastructure in order to obtain the benefits of both public cloud and private infrastructure. With a hybrid deployment, flexible business operations, enhanced performance, and optimized deployment costs are borrowed from the public cloud while stronger security and better control over resources are taken from the private infrastructure [32].

With the growing number of proponents for hybrid cloud architectures, public cloud providers have also started offering solutions that support hybrid deployments (i.e., combining their public cloud infrastructures with the private infrastructure from their customers). Existing hybrid solutions range from offerings as simple as establishing cheap software-enabled virtual private network connections (VPNs) between the cloud infrastructure and the private infrastructure, to having dedicated fiber-optic cables between the public cloud and the private infrastructure (see Appendix A for details on hybrid cloud solutions).

With all the hybrid solutions available, deploying applications to hybrid cloud is heavily influenced by the capabilities and offerings of the hybrid solution used for application deployment. These capabilities contribute differently to the overall performance, cost, and scalability of a deployed application. As a result, a system architect making decisions about a hybrid deployment has to deal with the challenging problem of deciding about the variabilities for all infrastructure elements and their effects on application behavior in order to achieve an optimal deployment.

1.1 Deploying Applications to Hybrid Cloud

There has been significant amount of academic and commercial interest in understanding the implications of deploying an application to the cloud [31]. This includes understanding factors such as the overall performance of the application, cost of deployment, bottlenecks in a deployment, points of vulnerability or failure in a deployment, etc. There has been both manual and automated efforts to gain this level of understanding. RightScale [28] and Microsoft [27] help their customers decide about their cloud deployments by having engineers manually look into the variabilities and requirements for public and private deployments of an application. On the contrary, frameworks such as PaaSLane [26] have tried to automatically achieve this by doing code-level inspection of software systems and verifying application deployments against cost and performance schemes offered by cloud providers.
1.1. Deploying Applications to Hybrid Cloud

A hybrid deployment should prevent placing privacy-critical code or data in the public cloud in order to adhere to confidentiality, integrity, and privacy requirements for critical business information. Constraining placement of these privacy-critical code and data entities to the premises infrastructure demands for careful planning in order to achieve performance and cost optimization in a hybrid cloud deployment. Suboptimal planning can lead to inaccurate provisioning of required resources (e.g., computation, communication, and storage resources) which may lead to expensive deployments with degraded performance.

Consequently, effective hybrid cloud deployment of a software system depends on identifying the appropriate location for deploying critical components or data entities in the system, identifying their inter-dependencies, assessing entity placement constraints or cost considerations, finding the right hybrid solution for the deployment, and finally distribution and deployment of the system. Identifying the inter-dependencies between code and data has been a major challenge from the early stages of research on code mobility [42, 51]. This challenge leads to difficulties in optimal separation and distribution of code and data entities of an application [60, 74]. Programmers’ intuitions on distribution and deployment of functionality is often inaccurate and can result in decreased performance and efficiency of the running software [63]. Also manual identification of business critical information and evaluation of the requirements for its secure deployment is error prone [108]. Zdancewic et al. [108] have shown that programmers often misjudge the sensitivity of information which may lead to undesirable placement of critical data when deploying software to the cloud. Since such decisions are usually made at the early stages of software design, making changes to the deployment can be costly if not done automatically. Automated techniques previously have been utilized to help with analysis and deployment of service implementations across distributed host machines. These techniques are commonly referred to as application partitioning [45, 58, 71]. Techniques for application partitioning can also be utilized in the context of hybrid cloud for efficient deployment of software systems.
1.2 Problem Statement

The problem of application partitioning has been addressed in previous research. Systems such as CloneCloud [45], Cloudward Bound [58], and Leymann et al.’s [71] partition only software but not data. Other work in the area provides for partitioning of data, e.g., partitioning of relational databases [69] or Map-Reduce job/data components [33, 70, 103]. Unfortunately, none of these approaches combines code and data partitioning in the context of a hybrid cloud. However, one cannot “cobble together” a hybrid solution by using independent results from such approaches. The problem of hybrid deployment requires detailed analysis of inter-dependencies between code and data within the context in which the code accesses the data. To address the requirements of a hybrid deployment, a new approach is needed that integrates application and data partitioning natively. This partitioning strategy also needs to be tailored to the type of application deployed in a hybrid cloud (see Chapter 4 for a detailed example).

Among all types of applications deployable to the cloud, web applications are of particular importance because of their prevalence and scalability, performance, and cost efficiency requirements. The focus of this thesis is particularly on partitioning of On-Line Transaction Processing (OLTP)-style web applications. Web applications follow the well known multi-tier architecture, generally consisting of tiers such as: client-tier, application-tier (serving dynamic web content), and back-end data-tier. Existing research on (semi-)automated partitioning of web applications has been only applied to one of the application- or data-tiers and does not address partitioning of software systems across both application- and data-tiers (which we refer to as cross-tier partitioning). Consequently, the main argument for this research work is that combined partitioning of code and data is a challenging task for software designers and system architects.

One major challenge in cross-tier partitioning is the tight coupling of data-flow between application- and data-tiers. The application-tier can make several queries during its execution, passing information to and from different queries. Even though developers follow best practices to ensure the source code for the business logic and the data access layer are loosely coupled, this loose coupling does not apply to the data-flow. The data-flow crosscuts application- and data-tiers requiring a new strategy that considers

\footnote{In the rest of the thesis we use the terms code and application-tier interchangeably.}
1.2. Problem Statement

the two simultaneously and explores data dependencies within the context of single requests in the web application. This is particularly important when making decisions about the secure and efficient deployment of software systems. Any efficient deployment must avoid, whenever possible, the latency and bandwidth requirements imposed by distributing such data-flow.

A second challenge for cross-tier partitioning is that it requires an analysis that simultaneously reasons about the execution of application-tier code and data-tier queries. On the one hand, previous work on partitioning of code is not applicable to database queries because it does not account for modeling execution of database queries particularly when several database tables are involved. On the other hand, existing work on data partitioning does not account for the data-flow or execution footprint of the application-tier [69].

Given all the above, a high-level formulation of the research problem can be defined as follows:

How could code and data dependency analysis, system resource usage, and cost models of a cloud platform be leveraged in a cross-tier partitioning framework in order to help developers make cost- or performance-effective decisions for hybrid deployment of OLTP-style web applications to the cloud?

To clarify the research problem, consider the following example. Assume that a web application company (e.g., a stock trading company, an online auction site, etc.) is willing to take its previously on-premises software system and deploy it to the cloud to take advantage of the cost savings and scalability capabilities. The migration plan suggests cost reductions on ownership and maintenance for individual applications, savings on energy consumption, and cost reductions on the required IT staff by leasing some infrastructure from public cloud provider companies. Savings on cost and energy are sufficiently high for the company to give up control and ownership on major parts of the system. However, the company requires private and confidential information (e.g., personal data for its users including their credit card information) to be stored on the company’s on-premises servers.

When migrating to the cloud, keeping control on where the data (particularly privacy-sensitive data) is physically stored is one of the major requirements of most medium and large enterprises in order to avoid data lock-in or reduce risks of exposing confidential data [87]. Revealing corre-
lations between application components and different types of data to be placed on servers within or outside company’s premises requires thorough investigation of the architecture of the application. The company thus requires tooling and solutions to determine and assess performance gains and cost savings of optimal partitioning plans for the target software system. In the absence of proper tooling and solutions, given the scale and number of applications to be re-architected and data to be partitioned, such deployment would require hours of planning and engineering by IT managers and system architects [1].

Figure 1.1 shows a high-level diagram for an ideal hybrid deployment for our target company, in which both code and data are partitioned across the public cloud and the private infrastructure. As shown in Figure 1.1 in an ideal deployment, the company will keep the minimal subset of code and data entities on-premises and push the rest to the cloud. This minimal set needs to be determined based on the defined placement constraints, the dependencies across code entities, interaction of code entities with data entities, computation resource needs, estimated data transfers to/from the cloud (inbound/outbound data), and all their associated monetary costs. Under such circumstances, the company needs to deal with the following questions:

- How to model inter-relations between code and data?
- How to model performance and cost implications of a deployment?
- How to optimize cost and performance towards an optimal hybrid deployment?
- Will the combination of optimization strategies and cost models lead to increased performance and cost savings?

The questions above are in fact derivatives of the high-level research question we formulated earlier in this section. The goal of this research work is to provide algorithms, techniques, and tools to help software architects and system engineers find answers to the above questions accurately and avoid uninformed hybrid cloud deployments. Consequently, we define the thesis of this research as follows:

We can develop context-sensitive dependency models and cross-tier partitioning algorithms to facilitate the migration of OLTP-style web applications to a hybrid cloud deployment. This is achieved through identification
1.3 Methodology

Figure 1.1: Hybrid deployment architecture with distributed code and data.

of constraints imposed on cost and data placement, modeling and analysis of code and data dependencies in a monolithic software system on a reference machine, formulating the dependency models and constraints into an optimization problem, transforming the monolithic dependency models and constraints to a target distributed deployment through solving the optimization problem.

Next, we describe our methodology to validate the thesis above.

1.3 Methodology

The approach taken in this thesis builds on top of existing research work on application partitioning, resource provisioning, and cost modeling. Formulating a cross-tier partitioning strategy requires combining profiling, cost modeling, and partitioning for the application-tier with those of the data-tier. As shown in Figure 1.2, creating partitioning formulations at each tier consists of the same high-level steps (although the specific details vary):

- Profiling
  - measuring execution of software functions on a reference machine.
  - measuring execution of SQL queries on a reference machine.
1.3. Methodology

Figure 1.2: Highlevel representation of our methodology involving the steps of profiling, analysis, and partitioning.

- measuring data exchange between software functions and data entities on a reference machine.

- **Dependency Analysis and Cost Modeling**
  - converting the collected profiling data into a dependency model representing the software system.
  - transforming the reference software dependency model into a distributed deployment model on target machines.
  - applying cost and performance models to the distributed deployment model.
  - identifying placement constraints for code and data in the distributed model.

- **Application Partitioning**
  - transforming the distributed model into an optimization problem.
  - solving the optimization problem using *binary integer linear programming (BIP)*.

*Profiling.* We rely on profiling of the application-tier and the data-tier to provide a model of the system under analysis and monitor load on different application components. This process involves injecting extra profiling code into the application and data tiers to collect traces of application execution
1.3. Methodology

and data access for each tier in the system. The data is used to create a *dynamic dependency graph (DDG)* \[39 49 78\] of the application as a directed acyclic graph where functions and data entities (e.g., database tables) are represented as graph nodes and their dependencies are represented as graph edges. Weights on vertices and edges of the graph represent cpu usage and data exchange respectively between entities in the application.

**Analysis.** Dependency analysis involves understanding which code or data entities are inter-related and what model of separating components would contribute to a cost effective yet performant deployment of the software system.

Existing research on software dependency modeling provides techniques to determine the optimal mapping of software functions and data entities to network hosts (e.g. client or server) \[43 63 78\]. Such research only supports a simple one-to-one mapping of functions and data to hosts. However, in our research we have found this simple mapping to be inadequate because the optimal placement of a function depends on the context in which the function is used or the data is accessed. In short, sometimes it is better to execute a particular function in the public cloud and sometimes it is better to execute it on premises. We have developed a technique called *context-sensitive dependency modeling* that allows for replication of software functions or data entities depending on their context of use. In Chapter 3 we discuss the details of this dependency model and report on the results of using this model compared to the existing dependency models.

Upon creating a dependency model, the model is augmented with data that can reflect on either of the two primary objectives for optimization:

1. performance, e.g. request processing latency
2. monetary cost of deployment

Previous work on performance and cost models \[45 58 71\] does not provide a means for developers to explicitly make trade-offs between these two objectives. We provide a *flexible cost modeling* technique which allows developers to make trade-offs between latency and monetary costs. The flexible cost model supports specification of code replication, data replication, and adjusting cost-to-performance trade-offs when declaring performance or cost characteristics of an application. All these constraints are formulated into the partitioning algorithm to help with achieving the optimal partitioning
1.3. Methodology

results. This also allows for different hybrid deployment solutions (see Appendix A) to be encoded into the cost model (see Chapter 3 for details).

For our cost model to be effective within the context of hybrid cloud deployments, financial and operational characteristics of the host cloud need to be defined into the cost model and utilized in the partitioning algorithms. Our analysis of cost schemes for various cloud providers revealed that public cloud providers impose asymmetric charges for data to/from their infrastructure [2, 3]. Purposefully enough, these asymmetric data transfer charges encourage pushing data to the cloud by making cost of data transfer to the cloud considerably cheaper than data retrieved from the cloud [3]. Exploiting this asymmetric cost model, we developed an optimization formulation for our partitioning algorithm that further reduces the cost of a hybrid deployment (see Section 3.5).

Partitioning. At its core, partitioning is a method for applying mathematical optimization to distributed software development. Binary Integer Programming [90] has been utilized previously for partitioning of applications (although not for cross-tier partitioning) [44, 67, 78, 105]. A binary integer program (BIP) consists of the following:

- **Binary variables:** A set of binary variables \( x_1, x_2, \ldots, x_n \in \{0, 1\} \).
- **Constraints:** A set of linear constraints between variables where each constraint has the form: \( c_0x_0 + c_1x_1 + \ldots + c_nx_n \{\leq, =, \geq\} c_m \) and \( c_i \) is a constant.
- **Objective:** A linear expression to minimize or maximize: \( cost_1x_1 + cost_2x_2 + \ldots + cost_nx_n \), with \( cost_i \) being the cost charged to the model when \( x_i = 1 \). The job of a BIP optimizer is to choose the set of values for the binary variables which minimize/maximize this expression.

We combine our code and data partitioning algorithms into a cross-tier partitioning algorithm in order to guarantee an optimal hybrid cloud deployment (see Chapter 4 for details).

The overall process of applying cross-tier partitioning is shown in Figure 1.3. In the top left we see a monolithic web application before partitioning. Notice that the profiling logs are split in two branches with the upper.

\footnote{Amazon and RackSpace have no charges for data transfer into their cloud infrastructure while they charge between $0.12 to $0.18 for every GB of data leaving the cloud.}
1.4 Contributions

The work presented here makes four primary contributions and two secondary contributions. The four primary contributions correspond to the four research questions we listed in Section 1.2 and are discussed below:

贡献 #1（基于上下文的依赖模型）：我们设计和开发了应用分层工具，这些工具能够生成动态依赖模型，并捕捉到目标应用的资源使用情况。

贡献 #2（资源分配策略）：我们提出了一种新的资源分配策略，该策略在考虑依赖关系的同时，能够最大化应用性能。

贡献 #3（跨级并行处理）：我们设计了一种新的跨级并行处理算法，该算法能够在不增加计算复杂性的前提下，提高应用性能。

贡献 #4（数据并行处理）：我们提出了一种新的数据并行处理策略，该策略能够在保持数据完整性的同时，提高应用性能。

贡献 #1 (Context-Sensitive Dependency Modeling): We designed and developed application profiling tools which generate dynamic dependency models of the target application and capture resource usage within
1.4. Contributions

the context of application execution.

**Contribution #2 (Flexible Cost Modeling):** We developed a flexible cost model that captures specification of code and data replication as well as cost-to-performance trade-offs to capture the details of the deployment environments (both public and private).

**Contribution #3 (Cross-tier Partitioning):** We demonstrate that, in the applications we studied, combined partitioning of code and data entities to a hybrid cloud can provide a cost improvement of more than 55% compared to a naïve hybrid deployment, and 40% compared to when only code (and not data) is partitioned. The cross-tier partitioning work presented here is the first of its kind in tackling the problem of combined code and data partitioning for hybrid cloud.

**Contribution #4 (Asymmetric Data Exchange Costs):** We show that by exploiting the asymmetry in data exchange costs in our evaluated applications, we are able to reduce the monthly cost of hybrid deployments by 11% compared to when this asymmetry in communication costs is ignored. Employing the asymmetric cost models offers an interesting opportunity to further refine partitioning algorithms for hybrid cloud deployment.

Besides the primary contributions above, this research work also makes the following secondary contributions:

**Contribution #5 (Detailed Evaluations):** To validate the effectiveness of the developed solutions, we applied the developed techniques to several open source software systems (i.e., RUBiS [4], Apache DayTrader [5], and JForum [6]) and provided real world deployments to Amazon EC2 [2] to measure cost and performance of deployed systems.

**Contribution #6 (Tooling):** All the algorithms and tools described in this thesis are implemented under a framework for application partitioning and distribution to the cloud, named MANTICORE [67][4]. MANTICORE helps software architects make informed decisions with cost- or performance-effective deployments of their applications.

The contributions were summarized and published in three conference

1.4. Contributions


In the remainder of this dissertation, we describe the details of our methodology and highlight the contributions. This thesis is organized as follows: In Chapter 2 we discuss the related work on application partitioning, resource provisioning, and cost modeling. In Chapter 3 we discuss code partitioning and address the contributions on context sensitive dependency modeling and flexible cost modeling. In Chapter 4 we explain how code partitioning can be augmented with data partitioning to provide cross-tier partitioning. In that chapter we also present the contribution on asymmetric cost models for data exchange. In Chapter 5 we discuss the implementation of our MANTICORE framework. Finally in Chapter 6 we summarize the work presented in this research, explain the challenges, highlight the path for the future work, and conclude.
Chapter 2

Related Work

The problem of hybrid cloud deployment sits at the intersection of existing research work on automated application partitioning, resource provisioning, and software distribution. These research works correspond to the three steps of Profiling, Modeling, and Partitioning that we described in Chapter 1.

Application partitioning allows for identifying the size and the level of granularity for application components to be distributed for hybrid cloud deployment. That includes identifying whether and which components of the application (e.g., software functions, modules, or servers) need to be placed on premises, and which ones need to be placed in the cloud for the overall cost or performance of the application to be optimized. In case of hybrid cloud deployment, we are particularly interested in identifying existing research work that allows us to understand the right level of granularity for partitioning and distribution of web applications and the type of partitioning algorithms that allow us to better capture system characteristics when it comes to application partitioning. We are also interested to know the set of metrics that are collected when doing application partitioning in order to determine which ones are potentially applicable to hybrid cloud deployment. In the area of application partitioning, we narrow our focus on approaches that discuss two-way partitioning. Even though we are fully aware of the work on multi-way partitioning, we do not report on that body of research here mostly because at this point we are not aware of a practical approach to combine software modelling and multi-way application partitioning into a combined strategy for hybrid cloud deployment.

Resource provisioning helps with predicting the amount of required resources for each set of migrated components. Existing research work in this area presents models and mappings to analyze the behaviour of a software system on a target platform based on its execution profile on a source platform. This is particularly useful where hosts of deployment are not completely identical in their capabilities or where the overall behaviour of the system could be affected by external factors such as number of
requests to be processed. In case of hybrid cloud deployment, resource provisioning techniques are helpful when modelling the non-determinism in network capabilities, incoming requests, multi-threading issues, etc. While focusing on research provisioning, we omit any work that focuses on maximizing resource usage since that is not part of our focus in this research. From the resource provisioning literature we only borrow the models suggested to capture and map the behaviour of a software system when deployed from one infrastructure to another.

The final step in having effective hybrid deployments of an application is to facilitate migration of code components into target hosts and linking them together, for the distributed application to behave similar to monolithic software system. By looking into code mobility techniques and distribution platforms, our aim is to identify how this overall process can be facilitated. From the area of code mobility, we ignore previous literature on agent-based code migration where the are assumptions on pre-configurations of the source or the target hosts. Also out of all the existing distribution stacks (e.g., CORBA, Java RMI, etc.), we only focus on the ones supported in the default runtime environment of web applications (e.g., Java) and ignore non-standard efforts.

In this section, we briefly discuss the most related research work in each of the areas mentioned above and discuss how each work has contributed to my research.

2.1 Code Mobility & Distributed Middleware

Code mobility enables dynamic changing of the bindings between fragments of code, where they are resided in a distributed system [41]. Fugetta et al. [51] provide a comprehensive study and analysis of code mobility from three angles of technologies, design paradigms, and application domains. Code technologies refer to the languages and systems that provide the required mechanisms for supporting and enabling code mobility. The authors consider a five layer architecture for a system supporting code mobility. At the lowest layer reside the hardware and physical resources, followed by the core operating system providing basic system functionalities at the second layer. At the third layer, network operating system provides low-level functionalities for network access. Individual components, at the top most layer (i.e. fifth layer), are placed on top of a computational environment
2.1. Code Mobility & Distributed Middleware

(CE) layer (i.e., the fourth layer) that handles bridging between low-level functionalities for network and core operating system layers and the high-level requirements of components. Furthermore, CE provides components with the capability to dynamically relocate their components on different hosts. Individual components at the fifth layer can be execution units (EUs) representing flows of computation, or they can be resources capable of being shared by multiple EUs. The authors have classified mobility mechanisms to code and execution state management and data space management.

The above classification resembles the cloud support for application execution where the first three layers could be combined to the Infrastructure-as-a-Service (IaaS) layer in cloud computing with the second and the third layer combined into the abstract concept of hypervisor in cloud computing. The fourth layer could be considered as the Platform-as-a-Service (PaaS) and the fifth layer could be considered equivalent to the Software-as-a-Service (SaaS) levels of abstraction in the cloud classification. In this sense, the concept of code mobility with the considerations on mobility and elasticity is well applicable to cloud computing. Code mobility is related to the problem of partitioning and placing services in a cloud network. Distribution and relocation of code units (e.g., software functions) across machines in the cloud may effectively reduce the overall monetary cost of deployment, improve performance, and minimize resource usage based on metrics such as network traffic, energy expenditure, quality of service (QoS), etc.

Here we review the recent systems providing the technology for distribution of software components and enabling location transparency for component placements in a distributed software system.

There is a strong body of previous research on code mobility and distribution at the level of software objects. Sadjadi [89] provides a comprehensive study of available object-oriented adaptive middleware systems. This includes systems like Common Object Request Broker Architecture (CORBA) [79], Java Remote Method Invocation (Java RMI) [76], and the Distributed Component Object Model (DCOM) [46]. What is common in all these distributed middleware systems is that they provide an interface for binding a remote object to be realized as a local object located in the local address space. In our research we particularly make use of the Java RMI middleware to allow for distributed execution of software systems after deployment to hybrid cloud. Benefiting from the distribution model offered
in Java RMI, we choose to have some of the software code executed in the public cloud and some of it in the private cloud. The boundaries on how much of the code should be executed in either of the two infrastructures is determined based on analysis we perform on software metrics such as execution time, data exchange, quality of service, etc (see Chapter 3 for details).

Aside from the major object-oriented middleware systems, there also exist service-oriented component-based frameworks. In these frameworks applications are realized at the level of modular units which form application components. These frameworks, once enabled for distributed deployment, allow service components for their applications to be deployed across different host infrastructures. OSGi is one of the prominent service-oriented component-based frameworks available. Remoting middleware built on top of OSGi have been widely used for distributed deployment of OSGi-based applications in a hybrid cloud setting. Next we review the remoting middleware, utilized in the context of hybrid cloud:

Remote OSGi (R-OSGi) is a distributed middleware platform capable of transparently distributing software modules of an application across accessible devices and platforms [85]. R-OSGi resembles a distributed service-oriented component model where cross-network distribution of services and their late bindings happen through dynamic generation of proxies that connect one peer to another over a network channel. Any failure in the network or in remote invocation of services in R-OSGi is mapped to local events, allowing dependent services to treat remote failures just like local failures. Performance analysis for R-OSGi shows that the overhead of remote method invocations is negligible (1.5%) and outperforms Java RMI (16% overhead). The prominent aspect about the R-OSGi distribution model is that its distribution of a software system happens at the boundaries of its components and with minimal intervention to the internal component code. While distribution at the level of software components allows for easier decoupling, not all software systems do have clear boundaries for their components. In our work we hold the position that analyzing a software system for distributed deployment should not be limited by logical boundaries built around units of code forming the software system. As such, in our distributed middleware we analyze dependencies among code entities at various levels of granularity and provide a flexible model of distribution (see Chapter 3 for details).

AlfredO [86] is a light-weight middleware architecture aiming to allow
mobile phones and resource constrained devices to be augmented by resources from the cloud for performing resource intensive computation [86]. Developed by the designers of R-OSGi, AlfredO fully relies on R-OSGi for the purpose of code mobility and exchange of required modules and components between the mobile phone and a cloud platform. AlfredO offers scalability and ease of administration, flexibility, device independence, security and efficiency. AlfredOs prominent enhancement over R-OSGi seems to be the possibility to integrate on-demand code mobility into its design which allows moving a thin client for modules and bundles to a resource constrained device while keeping this client as minimal as possible. The use of the thin-client strategy used in AlfredO helps to enhance efficiency. The use of a thin client strategy in AlfredO is also utilized in our approach for distribution of software components. However, in our approach it is achieved through replication or distribution of functions in a software system which fall into a finer level of granularity compared to modules used in AlfredO.

Further to the distribution support provided by OSGi-enabled middleware, PCOM [40] is another component-based service-oriented framework which inherently supports distribution. PCOMs major goal is to provide generic automatic adaptation support for service and component selection in a distributed environment. To do so, PCOM requires i) applications to be clearly specified in terms of their services and their non-functional properties, ii) services to be monitored in order to deal with missing or lost services and also changes in their nonfunctional properties, iii) users to define policies and strategies for choosing from a set of services with similar functional and quality-of-service properties, and iv) the resulting system to stay minimal in terms of resources consumption. Dynamism in PCOM is addressed through late binding and contract negotiation, nonetheless, due to not having a centralized source for registering and discovering services, a peer to peer process of contract exchange in search of a particular service might become extremely exhaustive and subject to failure. In our approach we stay away from a contract based model of distributing components due to the extra overhead of contract negotiation. We achieve distribution passively (i.e., at compile-time) as opposed to a runtime model of distribution through contract negotiation as done in case of PCOM.
2.2 Application Partitioning

As discussed in Section 1.3, application partitioning splits a monolithic application into standalone, yet dependent, modules that can run on multiple hosts while ensuring that the overall behavior of the partitioned application remains consistent with the original application. The process of partitioning is often performed in order to improve on the quality of service (QoS) (e.g., increasing throughput while decreasing latency and response time) within the context of business-to-business and business-to-customer applications. In order to classify the work done in the area of application partitioning for large and scalable distributed systems, we have extracted the following dimensions in the design of partitioning algorithms from the reviewed related work in this area:

- **Level of Granularity for Distribution**: Level of granularity refers to the boundaries at which the application partitioning algorithm separates the constructs of an application into standalone modules. These modules, while independent, should preserve their behavior within the context of the original application upon application’s distributed deployment. Partitioning is usually applied at the following levels of granularity: *Language Entities in the Source Code (e.g., software functions)*, *Application Binary*, *Application Modules and Components*, and *Application Execution Engine*.

- **Model of Profiling**: Model of profiling refers to the resource usage information collected from the application in order to make informed decisions about optimal partitioning of the application. We consider two main models for application profiling: i) *Static Profiling* and ii) *Dynamic Profiling*. In static profiling, analyzing the code (whether source or binary) happens statically and collecting required optimization information usually happens prior to or during compile time. In this case, analyzing the signatures for operations or functions allows for estimating the amount of data exchanged between different entities in the application and information about bandwidth and throughput can be collected without analyzing the running code. Dynamic profiling is done during runtime and when analyzing method signatures is not enough to collect all the information required to partition the application. Dynamic partitioning can be done in offline or online modes. During offline profiling, application entities are executed and monitored in a controlled environment. During online profiling however, light weight profiling is added to the modules and entities of the
application in a real setting. Dynamic profiling helps with collecting accurate information on CPU and memory usage, communication bandwidth, I/O operations, etc.

- **Model of Placement**: We have identified two major models of placement for a partitioned application: i) *Client/Server Placement* and, ii) *Multi-Node Placement*. In the client/server placement model, application entities are placed across two nodes, i.e., the client and the server. In the multi-node placement model, the entities of the application usually are placed on more than one node where nodes run separate independent and distinct entities and modules of an application or replications of modules. The distinction is important as some algorithms are only tractable for the client/server model of placement.

- **Partitioning Methods**: Graph based partitioning is a common and widely used partitioning strategy in which the relations between the application elements are modeled in the form of a directed acyclic graph (DAG). The nodes in the graph usually represent the application entities that are distributed across several nodes during the process of partitioning while the edges show the relations between these entities. The edges could also bear weights showing the size of data communicated between two consecutive vertices. Depending on what the entities are, the type of graph that is used to model the relations varies. The Program Dependency Graph (PDG) is used to show the data flow between the functions or operations in an application. PDG is used in situations where the bandwidth or the throughput are the critical parameters for the application partitioning. Object Relation Graph (ORG) is a different model in which the relations between the objects are modeled as a graph. The graph indicates what objects create the others, reference the others, or use the others. Finally, a Module Dependency Graph (MDG) is used to show the relations between modules or services composing an application. In this model, the edges in the graph represent coarse grained dependency relations between components in the applications and the amount of data exchanged between these components is usually encoded as the weight for the edges connecting these components.

- **Optimization Parameters**: When creating a graph of the application, whether it is a Program Dependency Graph (PDG), an Object Relation Graph (ORG), or a Module Dependency Graph (MDG), vertices and edges in the graph are weighted by information collected
2.2. Application Partitioning

during application execution in order to determine how the graph should be partitioned for the application distribution to be optimal. The weights are collected based on parameters like throughput ($TP$), response time ($RT$), CPU Usage ($CPU$), Bandwidth ($BW$), Memory Usage ($Mem$), and Battery Usage ($Battrey$). Defining the edge weights for the dependency graphs is influenced by what information from the application is collected prior to partitioning and how they are combined to properly reflect on the behavior of the application.

In the remainder of this section we go over the existing work on application partitioning. We investigate each work using the application partitioning dimensions we discussed earlier in this section.

Hilda [104, 106] is a high level declarative language for developing data driven Web applications. The latest version of Hilda supports automatic partitioning with performance optimization based on linear programming. Its programming model is based on SQL and is only suitable for data-driven applications. The entities in Hilda are called Application Units and are roughly comparable to classes. The programmer develops the code for the Application Units and hence partitioning criteria are enforced at the level of source code in Hilda. The performance optimization problem in Hilda is NP-Complete and is solved with a randomized rounding approximation algorithm [84]. The model of placement in Hilda is a client/server model due to the nature of the target Web applications it generates. As for optimization parameters, Hilda focuses on optimizing for response time and bandwidth. Table 2.2 summarizes the characteristics of Hilda.

<table>
<thead>
<tr>
<th>System</th>
<th>Level of Granularity</th>
<th>Profiling Model</th>
<th>Partitioning Model</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hilda</td>
<td>Source</td>
<td>Offline Dynamic</td>
<td>Integer Programming</td>
</tr>
<tr>
<td></td>
<td>Placement Model</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Params</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Client/Server</td>
<td>RT, BW</td>
<td></td>
</tr>
</tbody>
</table>

Table 2.1: Hilda’s Characteristics

SWIFT [43] is a system enabling development of 3-tier Web applications using Jif as a high level Java-like programming language. Jif enables developers to enforce security constraints on data flow between the client and the server. Jif is translated to WebIL as an intermediate language, gets augmented with module placement annotations, and then translated to
2.2. Application Partitioning

two Java applications one for the server and one for the client. The client code is further translated to JavaScript using Google Web Toolkit which enables placements of the client code on the client machine. Similar to Hilda, SWIFT also enforces partitioning at the level of source code through static and offline dynamic profiling. In order for the best partitioning to be decided, SWIFT employs linear integer programming technique which optimizes the flow of information in a client/server placement through optimization on response time and bandwidth. The authors provide an integral relaxation for their ILP formulation that is polynomial-time solvable following the optimizations performed on modeling data exchange between the elements in the SWIFT application. The graph that SWIFT uses in order to show the relations between the elements of the program is an acyclic control flow graph (CFG). The nodes in the graph are the statements in the program and the edges are given a weight of 1 if the consecutive vertices are split between client and server and 0 otherwise. Table 2.2 summarizes the characteristics of SWIFT.

<table>
<thead>
<tr>
<th>System</th>
<th>Level of Granularity</th>
<th>Profiling Model</th>
<th>Partitioning Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>SWIFT</td>
<td>Source</td>
<td>Static + Offline Dynamic</td>
<td>Integer Programming</td>
</tr>
<tr>
<td>Placement Model</td>
<td>Partitioning Params</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Client/Server</td>
<td>RT, BW</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 2.2: SWIFT’s Characteristics

Wishbone [78] performs distribution through defining language entities for partitioning. Wishbone focuses on partitioning the code between resource constrained sensors and servers. Wishbone addresses the problems of heterogeneity of resource constrained devices including motes, smart phones, embedded devices, etc. For Wishbone to be able to support proper partitioning, the developer develops code using WaveScript, as a stream processing language that has runtimes for both embedded nodes and servers. WaveScript has been extended for developers to logically define the data flow graph (DFG) and determine which parts of the dataflow graph should be replicated on all embedded nodes. When partitioning the application, operators are either considered as movable or pinned; with pinned operators having strict dependencies to the platform they are being executed on and movable operators with looser dependencies to their underlying platform. Having this information, Wishbone creates a directed acyclic graph (DAG) with its vertices as WaveScript streaming operators. The graph is then run through the partitioning algorithm to generate the
2.2. Application Partitioning

distributed entities. Wishbone optimizes the CPU usage and the bandwidth used by the streaming operators. It uses an integer programming model to define the optimization problem and uses the off-the-shelf lp_solve integer programming solver to solve the optimization problem. Table 2.2 summarizes the characteristics of Wishbone.

<table>
<thead>
<tr>
<th>System</th>
<th>Level of Granularity</th>
<th>Profiling Model</th>
<th>Partitioning Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>Wishbone</td>
<td>Source</td>
<td>Static + Offline Dynamic</td>
<td>Integer Programming</td>
</tr>
<tr>
<td></td>
<td>Placement Model</td>
<td>Partitioning Params</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Client/Server</td>
<td>CPU, BW, Mem</td>
<td></td>
</tr>
</tbody>
</table>

Table 2.3: Wishbone’s Characteristics

Our research work differs from Hilda, SWIFT, and Wishbone in that we  
i) have look into the broader range of 3-tier OLTP-style Web applications that potentially can be deployed to the cloud,  
ii) have focused on other partitioning parameters (e.g., monetary costs of deployment to the cloud, data placement constraints, CPU, I/O, and memory usage requirements), and  
iii) have dealt with partitioning the binary.

J-Orchestra [99] has been one of the early efforts towards enabling automatic partitioning of Java applications. The difference between J-Orchestra and systems mentioned to his point is that, unlike the previous systems, J-Orchestra does not require access to the source code of the system in order to enable distribution. It employs an offline dynamic partitioning approach at the level of application binary (bytecode) with potentials for multi-node distribution. J-Orchestra converts all application objects to remote-capable objects accessible from remote sites. It then enables the user to decide about the placement of these objects based on whether these objects are anchored (i.e., if they should stick to their location) or mobile (i.e., they can be moved to other locations). J-Orchestra classifies Java objects to modifiable and mobile objects and, using a classifier and a profiler, helps the user decide where these mobile objects should be placed. J-Orchestra focuses on minimizing network traffic and uses a heuristic strategy to decide about the proper partitioning with the user capable of override the results. Table 2.2 summarizes the characteristics of J-Orchestra.

Diaconescu et al. [18] introduce a dynamic runtime infrastructure and automatic partitioning approach that similar to J-Orchestra performs Java
2.2. Application Partitioning

<table>
<thead>
<tr>
<th>System</th>
<th>Level of Granularity</th>
<th>Profiling Model</th>
<th>Partitioning Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>J-Orchestra</td>
<td>Binary</td>
<td>Offline Dynamic</td>
<td>Heuristics</td>
</tr>
<tr>
<td>Placement Model</td>
<td>Partitioning Params</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Multi-Node</td>
<td>BW</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 2.4: J-Orchestra’s Characteristics

byte code analysis and rewriting to partition and monitor the entities of a Java application. In their approach an object dependence graph (ODG) is created with its edges indicating creation of one object by the other, using it, or referencing it. Similar to J-Orchestra, a classification for local and dependent objects is performed in which dependency relations between objects is resolved by placing inter-process communication code between them. For the system to support execution in heterogeneous environments, the code for the system is first converted to a highlevel intermediate representation to generate an abstract syntax tree (AST) and then to the code for the target platform using a bottom-up rewrite system. In order to match the accuracy of the sampling strategy against the real behavior of objects and elements in the application, automatic profiling through code instrumentation is performed. During profiling, information about method duration, method frequency, hot methods, hot paths, memory allocation, and dynamic call graph during is collected. The system performs multi-constraint graph partitioning following Hendrikson et al.’s [61] heuristic graph partitioning approach which allows for multi-node distribution. Table 2.2 summarizes the characteristics of the approach by Diaconescu et al.

<table>
<thead>
<tr>
<th>System</th>
<th>Level of Granularity</th>
<th>Profiling Model</th>
<th>Partitioning Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>Diaconescu et al.</td>
<td>Binary</td>
<td>Static + Offline Dynamic</td>
<td>Multi Constraint Partitioning</td>
</tr>
<tr>
<td>Placement Model</td>
<td>Partitioning Params</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Multi-Node</td>
<td>CPU, BW, Mem, Battery</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 2.5: Characteristics of the approach by Diaconescu et al.

Our work is close to the ones offered by J-Orchestra and Diaconescu et al. in that we also look into analyzing the binary for the applications during the profiling process. However, our profiling process then results in
2.2. Application Partitioning

creating a dependency graph that not only deals with code but also with how data entities are access in the application. As a result our partitioning relies on augmenting the code dependency graph of an application with data dependencies. Also, our partitioning strategy uses monetary costs of deployment to the private cloud as one of the primary motivations for application partitioning. Furthermore, the focus of our research is on the 3-tier OLTP-style web applications compared to the typical desktop Java applications used in the case of J-Orchestra and Diaconescu et al.’s work.

Gu et al. [57] propose a system called Spectra for adaptable offloading of Java applications from resource constraint devices to resource rich systems. Spectra offers the four phases of i) application execution monitoring, ii) resource monitoring, iii) application partitioning candidate generation, and iv) transparent remote procedure calls. The monitoring happens by providing an application execution graph (AEG) with each node in the graph indicating a class in the application and edges indicating method invocations and resource access between classes in the application. The weight metrics for each node in the graph formulate memory size, access frequency, location, and whether or not the class is native. The edges between classes represent interaction frequency and bandwidth requirements for the classes. As for partitioning, the system employs an efficient partitioning heuristic using Store and Wagner’s MinCut algorithm [96] to generate candidate partitioning plans. Their partitioning algorithm splits classes only between two nodes, a surrogate (a server) and a client. In order for the distribution to happen, they have modified the Hewlett-Packards Chai virtual machine by adding transparent migration of objects through using RPC. Their main contribution is to suggest an inference engine using the Fuzzy Control Model on when to do the partitioning of the applications based on the load on the system and the exhaustion of resources. Table 2.2 summarizes the characteristics of the approach by Gu et al.

<table>
<thead>
<tr>
<th>System</th>
<th>Level of Granularity</th>
<th>Profiling Model</th>
<th>Partitioning Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gu et al.</td>
<td>Binary</td>
<td>Online Dynamic</td>
<td>MinCut Algorithm</td>
</tr>
<tr>
<td>Placement Model</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Client/Server</td>
<td></td>
<td>CPU, BW, Mem</td>
<td></td>
</tr>
</tbody>
</table>

Table 2.6: Characteristics of the approach by Gu et al.
2.2. Application Partitioning

Ou et al. propose an adaptive heuristic (k+1) partitioning algorithm in which they partition an application into 1 unoffloadable partition and k offloadable partitions. Similar to Gu’s approach they create a dynamic multi-cost graph with classes in the application representing the vertices of the graph and edges representing associations between classes. The weights for the vertices indicate a combination of memory utilization, accumulated processing time, and bandwidth usage. The edge weights represent the number of invocations and data access between nodes or classes in the application. Their partitioning algorithm is performed by first identifying the unoffloadable nodes in the graph and combining them together as a single node. The algorithm then performs a (k+1) coarse partitioning based on a cost function to reduce the number of nodes to the k nodes desired for the partitioning. Finally, fine-tuning is done by shifting classes on the boundaries between nodes in order to reduce interactions between nodes. Their coarsening algorithm is based on their heavy-edge light-vertex matching (HELVM) algorithm which combines adjacent vertices with heavy edge weights and light vertex weights together.

Once the partitioning is done, a binary rewriting of the code is performed in order to perform offloading by placing proxies between classes on different nodes. Table 2.2 summarizes the characteristics of the approach by Ou et al.

<table>
<thead>
<tr>
<th>System</th>
<th>Level of Granularity</th>
<th>Profiling Model</th>
<th>Partitioning Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ou et al.</td>
<td>Binary</td>
<td>Online Dynamic</td>
<td>HELVM Heuristics</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Placement Model</th>
<th>Partitioning Params</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Multi-Node</td>
<td>CPU, BW, Mem</td>
<td></td>
</tr>
</tbody>
</table>

Table 2.7: Characteristics of the approach by Ou et al.

Our work employs the strategies from the HELVM algorithm and also how resource usage parameters are combined when creating the application graph. However, we have constraints on cost and data placement that are not considered in Gu et al.’s or Ou et al.’s work. Furthermore, the systems by Gu et al. and Ou et al. only care about the partitioning of the application and do not deal with how code in an application is dependent to the data. These works do not investigate the idea of cross-tier partitioning investigated in our work and furthermore, none of these works targets the context of cloud as the deployment infrastructure.
2.2. Application Partitioning

Coign [63] has been one of the early efforts supporting partitioning of MS COM components. In this sense, application partitioning in Coign has a coarse level of granularity compared to the approaches by J-Orchestra and Diaconescu and happens at the level of application modules. The problem that Coign solves is to facilitate repartitioning of the applications so that optimized partitioning can be performed in different execution scenarios. Coign enables application profiling by binary rewriting of the COM components. Based on the component communication profiles and component location constraints an abstract inter-component communication graph (ICC) is created. Location constrains are obtained from the programmer, or by analyzing the component communication record, or from application binaries. Once the ICC graph is created, the lift-to-front minimum-cut graph cutting algorithm is used to produce a two-machine client-server distribution of the original application. Once the application is distributed, a component factory is replicated on each host the application is running on. It is responsible for redirecting the calls for remote components to the hosts where those components are located. Coign supports both heavy-weight offline and light-weight online profiling mechanisms to monitor and analyze the COM components during runtime as well. Table 2.2 summarizes the characteristics of Coign.

<table>
<thead>
<tr>
<th>System</th>
<th>Level of Granularity</th>
<th>Profiling Model</th>
<th>Partitioning Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>Coign</td>
<td>Module</td>
<td>Offline + Online Dynamic</td>
<td>Lift-to-Front MinCut Alg.</td>
</tr>
<tr>
<td>Placement Model</td>
<td>Partitioning</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Client/Server</td>
<td>BW</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 2.8: Coign’s Characteristics

ABACUS [37] follows an application programming model in which a programmer composes an application from explicitly-migratable components or objects that have no dependency in their functionality. ABACUS deals with application programs written in C++ and for the system to perform properly, the programmer needs to embed functions associated with data streams into distinct components that can store or retrieve their states during migration. ABACUS sits at the border line of an application partitioning system and a load balancing system in that it provides support for fine-grained distribution and redistribution of components among different hosts. The components in ABACUS are mobile objects of low granularity performing
self-contained data intensive processing. ABACUS uses a heuristic approach for partitioning and distribution of mobile objects between different hosts and only deals with distribution of applications between a client and a server. The interesting point about ABACUS is that it provides support for dynamic monitoring of the behavior of the application which then enables the application modules to be redistributed across the client and the server based on the immediate changes in the profiling information. ABACUS is similar to J-Orchestra and Coign in that it only deals with the amount of data transferred between the source and the sink and uses dynamic online profiling to collect information about entities involved in partitioning and distribution. The important point about distribution in ABACUS is that, profiling is done based on a sample of potential scenarios. Table 2.2 summarizes the characteristics of ABACUS.

<table>
<thead>
<tr>
<th>System</th>
<th>Level of Granularity</th>
<th>Profiling Model</th>
<th>Partitioning Model</th>
<th>Partitioning Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABACUS</td>
<td>Module</td>
<td>Offline + Online Dynamic</td>
<td>Heuristics</td>
<td></td>
</tr>
<tr>
<td>Placement Model</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Client/Server</td>
<td>TP, RT, CPU, Mem</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 2.9: ABACUS’s Characteristics

Nanda et al. [77] deal with distribution of composite Web services. Their system takes in the centralized orchestration of Web services represented in a Business Process Execution Language (BPEL) format and tries to decentralize the execution in order to increase parallelism and reduce network traffic required for an application. In their case, execution entities are Web services running on different hosts. The decentralization helps in that sources of producing data directly send their data to less loaded sinks for consumption which helps with distribution of the load on receiving and processing data. Following a program dependency graph, services involved in decentralized orchestration are of finer level of granularity compared to the original services. As a result, a fixed node is considered a node in the program dependency graph that has execution resources, whereas a portable node could be a simple assignment of data to a variable, only indicating flow of data. The authors assume that for a centralized orchestration to be partitioned to a decentralized orchestration, at least one fixed service and zero or more portable services must exist. Their partitioning algorithm is also concerned with flow of data between services and throughput is considered as the primary performance metric. Finally, for the graph to be partitioned, the authors used a simple heuristic technique, called the merge-by-def-use.
2.2. Application Partitioning

Table 2.2 summarizes the characteristics of the approach by Nanda et al.

<table>
<thead>
<tr>
<th>System</th>
<th>Level of Granularity</th>
<th>Profiling Model</th>
<th>Partitioning Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nanda et al.</td>
<td>Module</td>
<td>Online Dynamic</td>
<td>Heuristic Merge-by-def-use</td>
</tr>
<tr>
<td>Placement Model</td>
<td>Partitioning Farams</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Multi-Node</td>
<td>TP, RT</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Giurgiu et al. [54] represent a distribution model at the level of application modules that works with components built on top of the OSGi framework [36]. The goal is to minimize communication between OSGi modules working together in creating an OSGi-based Java application. All the instrumentation and profiling of OSGi bundles is done manually and the collected information is statically analyzed to decide about the best partitioning for the OSGi bundles. For the sake of simplicity, the authors assume that the developer marks the movable bundles and each bundle only provides one single service shared with other bundles in the application. The model of distribution for the work by Giurgiu et al. is built on top of AlfredO [86]. For optimized partitioning of OSGi bundles, the parameters considered in partitioning include the memory, the code size, and also the amount of communicated data between the source and the client. Giurgiu et al. employ two heuristic models to find an optimized partitioning in the graph. In their first approach they use a full heuristic approach analyzing all possible distributions of OSGi bundles across the client and the server. In a K-step algorithm a minimized version of the full heuristic approach is chosen, where a local optimum is chosen as the possible distribution for the application. As mentioned earlier, their approach also distributes bundles only between two nodes, i.e., a resource constrained mobile node and a resource rich server node. Table 2.2 summarizes the characteristics of the approach by Giurgiu et al.

Modular applications are closer to the typical model of 3-tier or multi-component applications deployed on a PaaS in a cloud environment. As a result, applications evaluated in this thesis are similar to the ones investigated in Coign, ABACUS, or by Nanda et al. and Giurgiu et al. As discussed earlier, our research work has focused on profiling and partitioning at the level of application binary and through offline dynamic profiling. Consequently, our work is different from the one by Giurgiu et al.
2.3 Software Migration and Resource Utilization

Table 2.11: Characteristics of the approach by Giurgiu et al.

<table>
<thead>
<tr>
<th>System</th>
<th>Level of Granularity</th>
<th>Profiling Model</th>
<th>Partitioning Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>Giurgiu et al.</td>
<td>Module</td>
<td>Offline Dynamic</td>
<td>Heuristic K-Step Partitioning</td>
</tr>
</tbody>
</table>

As discussed earlier, the primary difference between the work presented in this dissertation and the previous body of research is our primary focus on providing cross-tier partitioning targeted to OLTP-style web applications. Also, the special constraints in the cloud, particularly the monetary costs and data privacy constraints, are the constraints driving profiling, partitioning, and distributed deployment of the applications.

Table 2.2 summarizes the contributions from the all the research works presented above, based on the dimensions we identified at the beginning of Section 2.2.

2.3 Software Migration and Resource Utilization

Another area of research related to our work is the research on predicting resource utilization by a software system and performing software migration to improve non-functional requirements in a system (e.g., quality of service and security). Software migration is similar to application partitioning in that both address nonfunctional requirements. The two are however different in that: the former assumes tighter coupling among software components where the loosening of components happens through partitioning; while the latter assumes that software components are loosely coupled with clear component boundaries that make components easily separable for relocation.

Within the context of both application partitioning and software migration it is critical to accurately predict resource utilization by software components. Both approaches rely on algorithms that require resource usage for software components to be analyzed prior to distribution and deployment on a target host. Accurate capturing of these metrics in the
### 2.3. Software Migration and Resource Utilization

<table>
<thead>
<tr>
<th>System</th>
<th>Level of Granularity</th>
<th>Profiling Model</th>
<th>Partitioning Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hilda</td>
<td>Source</td>
<td>Offline Dynamic</td>
<td>Integer Programming</td>
</tr>
<tr>
<td>SWIFT</td>
<td>Source</td>
<td>Offline Dynamic</td>
<td>Integer Programming</td>
</tr>
<tr>
<td>Wishbone</td>
<td>Source</td>
<td>Static + Offline Dynamic</td>
<td>Integer Programming</td>
</tr>
<tr>
<td>J-Orchestra</td>
<td>Binary</td>
<td>Offline Dynamic</td>
<td>Heuristics</td>
</tr>
<tr>
<td>Diaconescu et al.</td>
<td>Binary</td>
<td>Static + Offline Dynamic</td>
<td>Multi Constraint Partitioning</td>
</tr>
<tr>
<td>Gu et al.</td>
<td>Binary</td>
<td>Online Dynamic</td>
<td>MinCut Algorithm</td>
</tr>
<tr>
<td>Ou et al.</td>
<td>Binary</td>
<td>Online Dynamic</td>
<td>HELVM Heuristics</td>
</tr>
<tr>
<td>Coign</td>
<td>Module</td>
<td>Offline + Online Dynamic</td>
<td>Lift-to-Front MinCut Alg.</td>
</tr>
<tr>
<td>ABACUS</td>
<td>Module</td>
<td>Offline + Online Dynamic</td>
<td>Heuristics</td>
</tr>
<tr>
<td>Nanda et al.</td>
<td>Module</td>
<td>Online Dynamic</td>
<td>Heuristic Merge-by-def-use</td>
</tr>
<tr>
<td>Giurgiu et al.</td>
<td>Module</td>
<td>Offline Dynamic</td>
<td>Heuristic K-Step Partitioning</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Placement Model</th>
<th>Partitioning Params</th>
<th>Suggested Improvements</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hilda</td>
<td>RT, BW</td>
<td>Added Partitioning Constraints, (e.g., Cost and Placement)</td>
</tr>
<tr>
<td>SWIFT</td>
<td>RT, BW</td>
<td>Multi-node placement, broader range of Apps, Online</td>
</tr>
<tr>
<td>Wishbone</td>
<td>CPU, BW, Mem</td>
<td>Dynamic Profiling,</td>
</tr>
<tr>
<td>J-Orchestra</td>
<td>BW</td>
<td>Placement and Resource</td>
</tr>
<tr>
<td>Diaconescu et al.</td>
<td>CPU, BW, Mem, Battery</td>
<td>Expansion Models</td>
</tr>
<tr>
<td>Gu et al.</td>
<td>CPU, BW, Mem</td>
<td></td>
</tr>
<tr>
<td>Ou et al.</td>
<td>CPU, BW, Mem</td>
<td></td>
</tr>
<tr>
<td>Coign</td>
<td>BW</td>
<td></td>
</tr>
<tr>
<td>ABACUS</td>
<td>TP, RT, CPU, Mem</td>
<td></td>
</tr>
<tr>
<td>Nanda et al.</td>
<td>TP, RT</td>
<td></td>
</tr>
<tr>
<td>Giurgiu et al.</td>
<td>CPU, BW, Mem, Battery</td>
<td></td>
</tr>
</tbody>
</table>

Table 2.12: The Dimensions of Investigation for all of the reviewed approaches on application partitioning. In the “Partitioning Parameters” column, RT=response time, BW=bandwidth, Mem=memory usage, Battery=battery usage, and TP=throughput. The “Suggested Improvements” column, summarizes the improvements we suggest over the existing approaches as discussed within Section 2.2.
2.3. Software Migration and Resource Utilization

A software model would lead to partitioning solutions that are optimal in terms of using resources and reducing costs where constraints on cost and data placement are tight.

Ou et al. \[82, 83\] and Zhuang et al. \[109\] have shown that even for identical instance types for virtual machines leased from a given cloud provider (e.g., Amazon Web Services) different physical capabilities could be presented to the users. When modeling the behavior of an application for deployment to the hybrid cloud, it is important to understand how the application would behave in a target platform given the capabilities of the platform (e.g., CPU cycles, memory size, etc.). A conservative strategy to estimate the behavior of the application on a target machine is by doing real deployments. However, given the variety of physical machines assigned to a given instance type (e.g., on Amazon Web Services), physical deployments would introduce extra deployment costs, extra efforts during analysis, and potentially incomplete information on the behavior of the software system under study. An alternative strategy would be to try and model the estimated behavior of the target software system on heterogeneous hosts given the set of capabilities for those hosts.

Research on resource utilization has long looked into techniques on modeling the behavior of software systems, in particular to estimate how the behavior of the software system changes as the load on the software varies. In the rest of this section, we look into the previous work on modeling software resource utilization and software migration for deployment on heterogeneous machines. We combine the models of resource utilization from this section into the definition of our cost model when we combine cost schemes and performance constraints for partitioning (see Chapter 3 for details).

Urgaonkar et al. \[101, 102\] have looked into the problem of resource utilization in a shared platform where maximization of profit and optimized utilization of resources are desired. In their work, Urgaonkar et al. \[101, 102\] have employed kernel-based profiling as an empirical approach for collecting QoS measures on CPU and bandwidth usage. This usage based profiling is done for each component in the application running on a single machine. The prominent aspect of the method suggested by Urgaonkar et al. \[101, 102\] is that it performs online usage monitoring in order to provide predictions for how resource usage changes under varying load. The work of Urgaonkar et al., even though one of the major contributions in the area,
2.3. Software Migration and Resource Utilization

does not consider dependencies between code and data placement when making resource usage predictions. Furthermore, cost constraints for the underlying machines can heavily affect how resources will be used when deciding about migration and placement strategies.

In a more recent work in 2011, Tak, Urgaonkar, and Sivasubramaniam [98] have investigated the implications of migrating web applications in three settings of cloud-only, premises-only, and hybrid. For their cost models they have evaluated Net Present Value to measure the profitability of investing in each of the three deployment settings. For their cost model, Tak et al. have used the three dimensions of material costs (including software and hardware costs), maintenance costs, and expense costs (e.g., facility costs, etc.). In their modeling of costs, authors find workload, data transfer, storage capacities, and software licensing to be the important cost factors when combined. With all the analysis the authors discover that a full cloud deployment is only attractive to small businesses whereas with bigger organizations a hybrid deployment of applications is suitable particularly with horizontal partitioning and migration of application component being most beneficial to these organizations. The authors claim that in their analysis, vertical partitioning showed smaller benefits due to high costs of data transfer. In our work we have reached a similar conclusion in that partitioning would help with better utilization of cloud resources. We however show that a combination of vertical and horizontal partitioning could reach even further efficiency for hybrid deployment if patterns of data exchange are properly optimized. This is particularly useful when we utilize our asymmetric data exchange algorithm for optimizing ingress and egress data transfer to a public cloud platform (see Chapter 3 for details).

Stewart and Shen [94] have suggested a method for performance modeling for multi-component applications based on their component placements. Their goal is to enable system managers to have a cost effective approach for optimal capacity planning and component placement. The authors have divided their approach to two phases of application profiling and performance modeling. Application profiling is done by kernel profiling of application components based on CPU, disk, and communication overheads for a running application, and then generalizing the results into a linear function through linear fitting. The authors rely on the assumption that the overhead on resource usage can be modeled linearly and generalized to reflect on the overall behavior of the application. To derive the performance model for the application, the authors have employed the queueing the-
2.3. Software Migration and Resource Utilization

Shimizu et al. [92] suggest an approach for prediction and modeling of application resource usage similar to the one suggested by Stewart and Shen [94]. Shimizu et al. however try to provide a platform-independent model, so that the driven model can be used to predict the amount of required resources on any machine with arbitrary specifications. The authors use profiling techniques to collect computation (e.g., CPU clock speed, L2 cache size, etc.), communication (e.g., round-trip-time, interconnect bandwidth), and storage (e.g., main memory size, memory access bandwidth) parameters during application execution. In order to make the profiling information platform independent, the profiling data is contributed from several platforms with heterogeneous computation, communication, and storage capabilities. The evaluations are performed using real-world applications (e.g., the Postmark email server benchmark) on several platforms with different system specifications. The authors use numerical methods to ensure resource requirement predictions and application constraint satisfactions. Even though in our analysis and modeling of hybrid cloud deployments we assume cloud machines to be homogeneous, the possibilities to deal with varying resource capacities as offered by Shimizu et al. can particularly increase the effectiveness of our work when dealing with heterogeneous machines. We consider Shimizu et al.’s work to be integrated into our work as part of our future research plan.

Li et al. [72, 73] suggest a heuristic approach as a combination of linear programming and a nonlinear performance model to the problem of scalable component migration for service configuration with cost and QoS considerations. The authors deal with contention of software resources which is specifically effective in the multi-tenant setting of a cloud environment.
2.3. Software Migration and Resource Utilization

The approach starts with creating the Layered Queueing Network (LQN) model providing representations for resource usage, users, entries, tasks, processors, and delays. The model is accompanied by a linear network flow model representing resource usage (in particular, CPU demand) by services and based on user requests for each server task on each host. This network flow model, referred to as the processing network, produces the queueing delays for using resources. Their results show that for an application of 50 services it takes about 258 seconds for the algorithm to decide about the proper placement with error rates below 2%. The main issue with Li et al.’s technique for modeling the behavior of the system is that LQN is generated manually and hence its accuracy requires a good understanding of the original application. Furthermore, while the queueing model helps with mapping the communication behavior of the system, once combined with other resource usage metrics, the model becomes harder to analyze and use. On the contrary, our work focuses on a flexible method of resource modeling by doing it automatically and through a well-known dependency graph. This takes the extra work of modeling software systems off the shoulders of system architects by relying on analyzing the application binary for partitioning, distribution, and provisioning.

Hajjat et al. introduce Cloudward Bound [59] to provide a model for exploring the benefits of a hybrid migration approach for distributing multi-tier software systems between the public and the private cloud. For optimal migration of servers to the cloud, the authors solve optimization problems that assure data flow balance across the hybrid components of an application such that all transactions happening in a hybrid deployment are addressed either by the components in the public or the private cloud. The authors also try to minimize communication costs, minimize the increase in transaction delays, model and maximize cost benefits of migration, and assure the effectiveness of the security policies in place. The authors demonstrate that, even though the users will experience delays as a result of migration to the cloud, the increase in the delay stays within the accepted limits. The approach taken by Hajjat et al. is similar to our approach in taking cost and privacy constraints into account when planning a hybrid migration. However, their view into the problem is different from ours in that Cloudward Bound considers application components as coarse grained components in multi-tier software systems with each component consisting of several servers. On the contrary, our approach considers execution units or components of an application as fine-grained elements in the system. Migration of fine-grained components in the cloud allows
for better utilization of cloud resources whereas coarse grained migration may result in under utilization of resource and hence losing on cost savings. Furthermore, we put no assumptions on the existence of resource usage models for components in the application and try to dynamically measure resource usage for components in the system.

2.4 Summary

In this section we described the related work in the areas of code Mobility & distributed middleware, application partitioning, and software migration and resource utilization. As discussed earlier in this chapter, our work benefits from each of the three areas by, understanding and modelling the behaviour of software application in a distributed environment, analysis and partitioning of the software application for optimized deployment in the hybrid cloud, and finally distribution and deployment of the application across the public cloud and the private infrastructure.

Our work, even though heavily inspired by the existing research work in all of the above areas, distinguishes itself from the previous work by: i) looking at a finer level of granularity for application partitioning (software functions), ii) doing cost and performance optimizations across different tiers of an application, taking both code and data dependencies into account, and iii) benefiting from cloud cost models that allow for better decision making on optimized application partitioning and distribution. As shown in the forthcoming chapters, the collection of these contributions positively affects the overall distribution of applications towards cheaper and better performing hybrid deployments.
Chapter 3

Code Dependency Modeling and Application-tier Partitioning

Our approach towards cheaper and faster deployment of a software system to a hybrid cloud relies on modeling the behavior of the target software system (e.g., dependencies between code components and data entities), assessing cost and performance measures for code components and data entities in the software system, and partitioning and distribution of the target software system. In this context, we rely on resource monitoring, cost specification, software modeling, and application partitioning of the software model in order to generate cost or performance effective hybrid deployments.

As discussed in Section 1.3 of Chapter 1, we perform cross-tier partitioning both of the application tier and the data tier of a multi-tier software system (particularly a web application) to achieve cost and performance efficiency in hybrid deployments. In this chapter of the thesis, we focus our attention on dependency modeling, cost/performance analysis, and partitioning of the application tier. In particular, in the remainder of this chapter we address three of the contributions discussed in Section 1.4, i.e., context-sensitive dependency modeling, flexible cost modelling, and asymmetric data exchange costs. We describe how combination of these contributions can lead to effective hybrid cloud deployment decisions. In the next chapter we discuss how this can be combined with data-tier partitioning to form an overall cross-tier partitioning strategy.

This chapter is summarized and published in IEEE Cloud Computing Conference 2012 (CloudCom 2012) [67]. The remainder of this chapter is organized as follows: in Section 3.1 we describe the process of creating the dynamic dependency graph for a given software system. We elaborate on the details of creating the dependency graphs for software functions in Sec-
3.1 Creating the Dependency Graph

A common approach to modeling the behavior of a software system is creating its Dynamic Dependency Graph \([39]\). A Dynamic Dependency Graph - \(DDG(V, E)\) - is a directed acyclic graph where the set of nodes \((V)\) in the graph represents software functions involved in the execution of a given software system, and the set of edges \((E)\) represents the dependencies between those entities. The \(DDG\) is created by profiling the software system, collecting profiling traces, and converting the traces into a DDG.

The process of profiling involves injecting extra profiling code into the software and collecting execution traces of the application representing code, data, and their respective resource usage. We have developed a software profiler called \(jip-osgi\) \([7, 66]\) to perform profiling and analysis of software systems and generate a profiling trace (details in Chapter 5). A profiling trace contains information on the name and reference to the code function used, its performance metrics, and its preceding code or data entities. Table 3.1 shows the set of data collected in our traces when instrumenting a software system. The DDG is the result of converting the profiling trace into its graph equivalent.

Upon creation of the DDG, it is then augmented with vertex weights and edge weights. Depending on the context of analysis for the software system, vertex weights and edge weights may take on different values. When software performance is the focus, a vertex weight \(w_v\) can be set to refer to the average execution time for node \(v\) in the \(DDG\), and an edge weight \(w_{e(u,v)}\) can refer to the average amount of time taken for data exchange between vertices \(u\) and \(v\) \([45, 78]\). On the other hand, for a cloud deployment, where monetary deployment costs are the focus, \(w_v\) is set to refer to the monetary cost of execution for \(v\) and \(w_{e(u,v)}\) to refer to the monetary cost of data exchange between \(u\) and \(v\) (cf. Section 3.4) \([58, 67]\).
3.2. Modeling Code Components in the Dependency Graph

<table>
<thead>
<tr>
<th><strong>Element Name</strong></th>
<th><strong>Description</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>bn</td>
<td>code module name</td>
</tr>
<tr>
<td>cn</td>
<td>code class name</td>
</tr>
<tr>
<td>mn</td>
<td>code method name</td>
</tr>
<tr>
<td>c</td>
<td>number of execution times for a given software function</td>
</tr>
<tr>
<td>t</td>
<td>time of execution for a given software function</td>
</tr>
<tr>
<td>dfp</td>
<td>data transfer from the caller function to the callee</td>
</tr>
<tr>
<td>dtp</td>
<td>data transfer from the callee function to the caller</td>
</tr>
<tr>
<td>cfp</td>
<td>number of calls from the caller function to the callee</td>
</tr>
<tr>
<td>ctp</td>
<td>number of callbacks from the callee function to the caller</td>
</tr>
</tbody>
</table>

Table 3.1: The set of symbols used in the application trace of a data dependency graph, generated by our profiler tool. An example of the generated trace can be found in Program B.1 of Appendix B.

We allow the dependency graph to be generated at various levels of granularity, e.g., a coarse level of granularity where nodes represent code components, software modules or jar files - or - a fine level of granularity, e.g., classes or methods (a detailed model of the generated dependency graph is presented in Program B.1 of Appendix B). The level of granularity used in modeling code or data in the dependency graph has a direct effect on how the partitioning algorithm is applied to the model. Next we describe various strategies in modeling code components for dependency graphs.

### 3.2 Modeling Code Components in the Dependency Graph

In this section we discuss two of the existing models for representing an application dependency graph and compare them with a third dependency model that we suggest for effective hybrid cloud partitioning. The models are discussed within the context of multi-tier web applications (cf. Section 1.2). In web applications, the entry point to the software system is an incoming web request to a front-end tier which gets passed along to the application-tier. The application-tier, in consultation with the database-tier, prepares a response to the incoming request. The DDG for the web application is captured through modeling the set of business logic functionality (i.e., incoming request types) that it implements.
3.2. Modeling Code Components in the Dependency Graph

Common to all three models is the fact that vertices represent execution components in the application with vertex weights corresponding to aggregated CPU usage during the profiling of the application. Edges represent data-links with edge weights capturing latency and data transfer. The models differ in how the notions of component and data link are realized. For example, nodes might represent low-level function definitions from source code - or - high-level service request types. While both of these capture the notion of an execution component, these different choices have an important effect on what kind of optimization can be achieved.

In the remainder of this section, we will refer to Figure 3.1, inspired by Apache DayTrader [5], to illustrate the details of our discussed models. Apache DayTrader is a benchmark multi-tier web application emulating the behavior of a stock trading system. DayTrader implements seven service request types allowing users to login (doLogin), view/update their account information (doAccount & doAccountUpdate), view their portfolio (doPortfolio), lookup stock quotes (doQuotes), and buy/sell (doBuy & doSell) stock shares. It also consists of five database tables, storing information for account, accountprofile, holding, order, and quote. It should be noted that Figure 3.1 is simply a high-level comparative mockup and does not correspond to the actual models generated or used by our framework. A real example of a dependency model can be seen in Figure 3.2.

3.2.1 Request-based Model (RBM)

In this model nodes represent either request types (e.g., implemented business logic functionality) or data objects (e.g. relational tables). Edges are created between each request-node and all of the data objects the request operates on [34].

An illustration of the RBM, inspired by DayTrader, is presented in Figure 3.1(a). Two request types doPortfolio and doBuy are shown, each dependent on some tables from the DayTrader database (the identity of the tables is not relevant for this example). The edge weights show the average number of round-trips between the code components and database tables in a single request.

When partitioning the application for a hybrid deployment, the partitioning algorithm needs to decide whether the code for each request type
3.2. Modeling Code Components in the Dependency Graph

Figure 3.1: Application Dependency Models: a) RBM, b) SSM, c) CSM. Circles are application nodes and cylinders are database tables. The edge weights represent number of round-trip data exchanges between nodes.
3.2. Modeling Code Components in the Dependency Graph

should be executed in the public cloud or the private premises. The decision is based on the CPU demands of processing each request type and the various data dependencies. Executing code in the cloud will benefit from higher CPU scalability at a decreased cost as compared to on-premises CPU resources. However, separating request types from the data they depend on could introduce additional latency or bandwidth requirements.

For example, in Figure 3.1(a), we see that \texttt{doPortfolio} accesses the middle database table an average of 15 times per request. This makes the separation of \texttt{doPortfolio} from the database table inefficient both in terms of cost and performance. If the database table is restricted to be placed on premises, the \texttt{doPortfolio} request also needs to be placed on-premises. Having both code and data on premises clearly would not help with any cloud specific cost or performance improvement.

3.2.2 Static Structure Model (SSM)

A drawback of the \textit{RBM} is that it does not separate request types into individual programming language functions used in the implementation. This means there is only a single node representing the entire software functions involved in executing each request type in the application. Consequently, the partitioner does not have the flexibility to split the execution of software functions for a given request type between the cloud and the premises, even if that would provide an advantage. Some other research on application partitioning ([43, 59, 63]) deals with an implementation-level model of a service, we refer to as \textit{SSM}.

In this model nodes in the graph represent the \textit{definition} of a function or the existence of data objects. By \textit{definition} we mean that a node corresponding to some function occurs only once in the graph, regardless of how many times the function is typically executed during the processing of a request. For data objects, this distinction is not necessary (because data does not execute). An edge between two nodes means that there is at least one call between those functions.

An illustration of the SSM, inspired by DayTrader, is provided in Figure 3.1(b). Here we see \texttt{doPortfolio} and \texttt{doBuy} broken down into the functions that implement them. Using this model, the job of the application partitioner is to decide whether each function should execute code in the
public cloud or the private premises. However, both request types make use of the function get\textbf{Quote}. This causes the edge weights induced by \texttt{doPortfolio} and \texttt{doBuy} over get\textbf{Quote} to become conflated, resulting in an overly abstract representation. We found that having only one node representing functions such as get\textbf{Quote} may over-constrain the partitioning optimizer causing it to produce suboptimal code placement suggestions. Referring back to Figure 3.1, the conflated edge from get\textbf{Quote} to the database table bears a weight of 16 for which the profiler is not able to distinguish the weight induced by \texttt{doPortfolio} and the weight induced by \texttt{doBuy}. This results for the partitioner to assume a tight bound between get\textbf{Quote} and the database table for both requests \texttt{doPortfolio} and \texttt{doBuy} in a situation where in fact only \texttt{doPortfolio} has a tight dependency with the database.

In a SSM, applying the partitioning strategy would place each and every software function either in the cloud or on-premises, leaving no further refinement to be done to the final function placement decisions. Furthermore, conflated dependencies across multiple request types may confuse the partitioner in finding an optimal partitioning and separation of functions in a SSM.

Conflated dependencies are less of a problem in a request-based model. This is because there will be a new instance of each function under every request type, allowing for higher replicability of functions. However, functions are at most replicated as many times as the number of request types in the model, and functions within a particular request will not be replicated. As a result, the dependency model internal to each request in a RBM would in fact follow a SSM model.

### 3.2.3 Context-Sensitive Model (CSM)

In order to address the problems of the previous two models, we have built our framework on a context-sensitive model of software behavior. In this model each node represents the execution of a function in a distinct calling context. A calling context captures the transitive set of callers to each function execution (aka. an execution stack). Using the execution of a function rather than a function’s static definition as the basis for our framework allows our runtime to execute the same function on premises or in the public cloud depending on the situation. Note,
however, if we simply tried to apply automated optimization to a graph which contained one node for each distinct function execution in a captured execution profile, the graph would grow too large. We built on the assumption that we can capture differences in each function execution by only considering each execution different if it occurs in a different calling context.

An illustration inspired by DayTrader is provided in Figure 3.1(c). Now we can see that two copies of \texttt{getQuote} are created, one where it is in the context of \texttt{getQuotes} and one where it is in the context of \texttt{doBuy}. Consider, as is the case for DayTrader, that \texttt{getQuotes} calls \texttt{getQuote} several times in a loop; an average of 15 times as indicated by the edge weight. If \texttt{getQuote} was executed only on the public cloud, this would cause a large number of round-trips over the network, so in an optimal deployment it should be placed on premises where it is close to the data. However, in the context of \texttt{doBuy}, \texttt{getQuote} is only called once, so for this case executing \texttt{getQuote} in the public cloud does not incur extra latency, and yet we can take advantage of the public cloud’s pricing benefits. Different from previous research, our approach provides this capability for such context-sensitive partitioning. A realistic example of a context sensitive model, collected by our framework for the DayTrader application, is shown in Figure 3.2. The figure captures the CSM model for the \texttt{doLogin} request type in Apache DayTrader.
3.2. Modeling Code Components in the Dependency Graph

Figure 3.2: CSM Visualization of partitioning results for the doLogin request type (root node of the tree) in the DayTrader example provided by our framework - MANTICORE. Yellow nodes (alternatively light gray nodes) represent database tables and all the other nodes represent individual function executions with dark gray nodes being chosen to be placed on premises and white nodes to be placed in the cloud. Note in the figure that nodes representing database tables are replicated in the figure for presentation purposes, where in fact, the actual graph is a DAG instead of a tree. The figure exemplifies a case where all database tables are restricted to be placed on premises. As the figure shows, with latency being the dominant performance bottleneck, functions are pulled to the private premises when there is frequent database access under the partitioned subgraph (this also depends on other factors such as CPU usage of functions and size of data transferred).
3.3 Defining Constraints for Placement & Cost

As it has been discussed so far in this thesis, minimizing deployment costs while keeping control over placement of particular code and data entities is the main motivation behind having hybrid deployments for the cloud. Consequently, enabling software architects and designers to be able to effectively specify their code placement constraints and their deployment costs has been one of our major areas of concentration throughout this research. To provide enough support for system architects to flexibly define their software placement constraints and the costs associated with deployment of their software to public or private clouds we have introduced two deployment schemes: i) a Policy Specification Scheme that allows for defining the placement constraints and ii) a Cost Scheme that allows for specifying deployment costs.

In the rest of this section, we describe how these two specifications can be utilized by software architects to define constraints for a hybrid cloud deployment.

3.3.1 Policy Specification Scheme

The policy specification scheme (policy spec) is used to declare business logic functionality to be included in the DDG; the level of granularity for different entities in the DDG (e.g., methods, classes, etc.); list of entities that can possibly be ignored; and constraints on co-location or replication of code or data entities. We describe some of the features supported in our policy specification model below:

*Changing the level of granularity* allows for system developers to go from a coarse-grained dependency model (e.g., a request-based model) to finer levels of granularity (e.g., static structure model or context-sensitive model) when creating the DDG.

*Possibility to include or exclude modules in the DDG* enables software architects to narrow down their focus when analyzing a software system, e.g., by isolating the analysis to particular request types in the application, excluding cross-cutting components of the system (e.g., security or logging components), etc.

*Co-location or replication* provides flexibility in deciding on component placements, e.g. by restricting the authentication or authorization functions to be in the private premises, or allowing for stateless components in the
3.4. Augmenting Dynamic Dependency Graphs with Cost Metrics

system to be replicated, etc.

Details of our declarative policy specification can be found in Appendix C.1.

3.3.2 Cost Scheme

The partitioner also accepts a cost scheme to be used for a hybrid cloud deployment. The cost scheme essentially encodes the hardware capabilities and associated monetary costs offered by a cloud provider, but in a format that can be processed by our framework.

The cost model captures hardware characteristics for the machines hosting the target application. In our cost model, we have included hardware capabilities for which there is a direct monetary cost defined in existing public cloud cost schemes. This includes CPU performance measures, network bandwidth, storage capacities, etc. Details of our cost scheme are presented in Appendix C.2.

The partitioner tool combines all the cost and policy specifications into the representation of the DDG prior to applying a partitioning algorithm to it. These constraints affect the DDG by changing the weights for vertices and edges to reflect on the cost metrics rather than the performance metrics. Also the placement constraints are modeled in the form of constraints fed to the partitioning algorithm before searching for an optimal solution. All the collected information is utilized to generate a model of the application for the analysis.

3.4 Augmenting Dynamic Dependency Graphs with Cost Metrics

Cost modeling involves associating quantifiable metrics to the execution footprint of a given application. Since the overall time of execution for an application directly contributes to the cost of deployment, a simplistic optimization strategy would be to only minimize the total time of execution. However, execution time does not account for all the costs billed by cloud providers. While solving the optimization problem only with execution times allows for tuning the system for an optimal performance, missing cost information prevents any optimizations on other deployment costs (e.g.,
3.4. Augmenting Dynamic Dependency Graphs with Cost Metrics

cost of data transfer or cost of data storage) to be performed. To reflect the costs of a hybrid deployment, we augment the dependency graphs from Section 3.2 with cost implications of a hybrid deployment. This is done in two phases:

**Phase I:** In the first phase, vertex weights and edge weights need to be updated to reflect the overall time of application execution when deployed across private premises and public cloud. For a given vertex \( v \) in the dependency graph, we generate two weights indicating the execution time of any corresponding function on-premises and in the cloud (based on the CPU capabilities of the host machines). Generating these edge weights can be done either by profiling the software system on machines both in the public cloud and the private premises, or by applying interpolation techniques (e.g., linear fitting [95]) to estimate its execution time on a target machine \( (t_{\text{exec}, v}) \) based on its execution time \( (t_{\text{measured}, v}) \) during the profiling phase:

\[
t_{\text{exec}, v} = t_{\text{measured}, v} \times \frac{\text{CPU}_{\text{target}}}{\text{CPU}_{\text{source}}}
\]

where \( \text{CPU}_{\text{target}} \) represents CPU capabilities of the target deployment host and \( \text{CPU}_{\text{source}} \) represents CPU capabilities of the machine used during the profiling phase.

The edge weights are set to reflect the amount of time it takes for data to be communicated between two components \( u, v \) in the application where those components are split between the cloud and the on-premises data center. We utilize the communication latency model introduced in [65] as follows:

\[
t_{\text{comm}, u,v} = \left[ \Pi + \frac{d_{u,v}}{\beta} + \lambda \right]
\]

where \( d_{u,v} \) indicates the amount of data communicated on the edge \( e \) and \( \beta \) is the communication bandwidth between the premises and the cloud provider. The first expression (\( \Pi \)) determines the queuing delay for data transfer on the network; \( \frac{d_{u,v}}{\beta} \) is the data transmission time between the premises and the cloud; and \( \lambda \) is the median measured latency between the private premises and a given public cloud.

**Phase II:** In the second phase, we include implications of cloud deployment. The model starts by accounting for the actual cost of deployment
by applying cost schemes offered by a cloud provider. For pricing, we use information from the cost scheme as shown in Program C.2 of Appendix C to update vertex weights for the dependency graph model as follows:

\[
\text{cost}_{\text{exec}} = \alpha \times \text{cost}_{\text{exec unit}} \times \left( \frac{t_{\text{measured}}}{T_{\text{unit}}} \right) \times \frac{\text{CPU}_{\text{target}}}{\text{CPU}_{\text{source}}} \tag{3.3}
\]

where \( T_{\text{unit}} \) represents the time unit for which cloud charges apply, \( \text{cost}_{\text{exec unit}} \) indicates cloud charges for each \( T_{\text{unit}} \), and \( \alpha \) is the cost ratio of running each function on a target premises machine versus running it on the cloud machine. \( \alpha \) is configurable by the system architect. In our evaluations we change it from 1 to 25 for varied premises deployment costs, evaluating a 0 to 25 times cost saving for public cloud deployment.

Next we model the monetary cost of communication between vertices \( u \) and \( v \) as follows:

\[
\text{cost}_{\text{comm}}_{u,v} = \gamma \times \text{cost}_{\text{exec unit}} \times \left( \frac{t_{\text{comm}}_{u,v}}{T_{\text{unit}}} \right) + \frac{d_{u,v}}{D_{\text{unit}}} \times \text{cost}_{\text{comm unit}} \tag{3.4}
\]

In the first part of the above equation, we account for charges incurred due to the latency of remote function calls introduced in the hybrid cloud deployment (e.g., extra data exchange, etc.). The second part of the equation accounts for charges directly related to transferring data between functions deployed in the cloud and the ones on premises. In the above formula, \( D_{\text{unit}} \) represents the data unit for which cloud data charges apply and \( \text{cost}_{\text{comm unit}} \) indicates cloud charges for each \( D_{\text{unit}} \).

Finally, we introduce \( \gamma \) as a configurable parameter reflecting the effect of latency on the cost of deployment. This allows developers to use our cost model to make flexible trade-offs between monetary cost and latency. The larger a developer chooses the value of \( \gamma \), the algorithm will work towards minimizing communication latency and improving the round trip time; whereas a smaller \( \gamma \) diminishes the effect of latency. We can formulate \( \gamma \) in the equation below:

\[
\gamma = \frac{T_{\text{unit}} \times \text{cost}_{\text{latency unit}}}{\text{cost}_{\text{exec unit}} \times \text{cost}_{\text{latency unit}} \times T_{\text{latency unit}}} \tag{3.5}
\]

with \( \text{cost}_{\text{latency unit}} \) being the monetary cost of perceiving the latency time \( T_{\text{latency unit}} \) incurred during an end-to-end execution of a request.
3.5 Applying the Partitioning Algorithm

The formulation of Equation 3.5 defines \( \gamma \) in relation to \( T_{\text{unit}} \) and \( \text{cost}_{\text{exec}_{\text{unit}}} \), allowing for latency costs to be tied into the cost measures given in a public cloud provider’s cost schema. For example, given a workload of 100 req/sec and a round-trip latency of 10msec/req, there will be 1 second of latency for every second of system execution. If a software developer defines a cost-to-latency policy indicating that every hour of time wasted on latency (\( T_{\text{latency}_{\text{unit}}} \)) is worth $0.32 (\( \text{cost}_{\text{latency}_{\text{unit}}} \)), and given \( \text{cost}_{\text{exec}_{\text{unit}}} \) is $0.32 per hour (\( T_{\text{unit}} \)), Equation 3.5 reveals the value of \( \gamma \) to be set equal to 1 for the cost formulation to account for the overhead.

In a real deployment setting, the value of \( \gamma \) is derived from the cost-to-latency value indicated by the system architect. The system architect can experiment with different cost-to-latency values to reach the appropriate partitioning cost. Upon introducing each new cost-to-latency value \( \gamma \), our framework performs partitioning of the dependency models of the software system and recalculates the estimated monetary cost and performance implications of the deployment.

3.5 Applying the Partitioning Algorithm

As explained in Chapter 2, application partitioning, at its core, is a method for applying mathematical optimization to a dependency model of a given software system with the objectives of minimizing latency and reducing cost. We discussed how vertices and edges in an application’s augmented \( \text{DDG} \) are assigned weights that represent performance measures [45, 78] or monetary costs [58, 67] of entities in the \( \text{DDG} \). For every \( v \in V \), the ultimate goal of the partitioning algorithm is to determine whether \( v \) should be placed on-premises or in the cloud, for the overall monetary cost of deployment to be minimized or performance to be maximized. In this section, we describe the mathematical formulation of the \( \text{DDG} \) into an application partitioning problem.

3.5.1 The Partitioning Algorithm for Symmetric Data Exchange Costs

For each \( v \in V \) we define \( \text{cost}_{\text{exec}_{v}} \) to represent cost of executing \( v \) on-premises and \( \text{cost}'_{\text{exec}_{v}} \) to represent cost of executing \( v \) in the cloud (see Equation 3.3). Also we simplify the definition of Equation 3.4 into the
3.5. Applying the Partitioning Algorithm

following formula:

\[
\text{cost}_{\text{comm},u,v} = \text{latency}_{\text{cost}}(u,v) + \frac{d_{u\leftrightarrow v}}{D_{\text{unit}}} \times \text{cost}_{\text{comm}_{\text{unit}}}
\]  

(3.6)

where \( \text{latency}_{\text{cost}}(u,v) \) is equivalent to \( \gamma \times \text{cost}_{\text{exec}_{\text{unit}}} \times \left( \frac{t_{\text{comm}_{u,v}}}{T_{\text{unit}}} \right) \) from Equation 3.4, \( D_{\text{unit}} \) would be the unit of data to which cloud data charges are applied and \( \text{cost}_{\text{comm}_{\text{unit}}} \) would be the cloud charges for \( D_{\text{unit}} \) of data transfer, and \( d_{u\leftrightarrow v} \) represents data exchange between vertices \( u \) and \( v \). It should be noted that \( \text{cost}_{\text{exec}} \) and \( \text{cost}_{\text{comm}_{u,v}} \) could hold either the performance costs or the monetary costs depending on whether we optimize for performance or monetary costs.

As briefly discussed in Chapter 1, we utilize Binary Integer Programming (BIP) to encode the DDG into an optimization problem. For every node \( u \) in the dependency graph we consider a variable \( x_u \) in the IP formulation, where the set \( s \) refers to entities placed on-premises and the set \( t \) refers to entities placed in the cloud.

\[
x_u \in \{0, 1\}
\forall x_u \in s, x_u = 0
\forall x_u \in t, x_u = 1
\]  

(3.7)

With all the above constraints and cost modeling, the following objective can then be defined:

\[
\min \left( \sum_{u \in V} (x_u \cdot \text{cost}'_{\text{exec}_u} + (1 - x_u) \cdot \text{cost}_{\text{exec}_u}) + \sum_{(u,v) \in E} (x_u - x_v)^2 \cdot \text{cost}_{\text{comm}_{u,v}} \right)
\]  

(3.8)

where placement of \( x_u \) in the cloud results in \( \text{cost}'_{\text{exec}_u} \) being charged to the objective function while its placement on premises results in \( \text{cost}_{\text{exec}_u} \) being charged to the objective function. Also if \( u \) and \( v \) are on different hosts (i.e., \( x_u \neq x_v \)), the equation above charges the communication cost \( \text{cost}_{\text{comm}_{u,v}} \) to the objective function.

The quadratic expression in the objective function of Equation 3.9 can
be relaxed by making the expansion suggested in [78].

\[\forall (u, v) \in E \quad e_{(u,v)} \geq 0, \quad e_{(u,v)} \leq 1\]

\[\forall (u, v) \in E \quad e'_{(u,v)} \geq 0, \quad e'_{(u,v)} \leq 1\] \quad (3.9)

\[\forall (u, v) \in E \quad x_u - x_v + e_{(u,v)} \geq 0\]

\[\forall (u, v) \in E \quad x_v - x_u + e'_{(u,v)} \geq 0\]

With the expansion of Equation 3.9, \(e_{(u,v)}\) and \(e'_{(u,v)}\) will be 0 if both \(x_u\) and \(x_v\) are assigned to the same host but \(e_{(u,v)}\) will be 0 and \(e'_{(u,v)}\) will be 1 if \(x_u\) is on \(t\) and \(x_v\) is on \(s\) or \(e_{(u,v)}\) will be 1 and \(e'_{(u,v)}\) will be 0 if \(x_u\) is on \(s\) and \(x_v\) is on \(t\). Given Equation 3.9, the objective function of Equation 3.8 is converted to the following non-quadratic formulation:

\[
\min \left( \sum_{u \in V} (x_u \cdot cost'_{exec_u} + (1 - x_u) \cdot cost_{exec_u}) + \right.

\left. \sum_{(u,v) \in E} (e_{uv} + e'_{uv}) \cdot cost_{comm_u,v} \right)
\]

We refer to the objective of Equation 3.10 as Symmetric IP in the rest of the thesis as it does not distinguish between inbound and outbound communication costs

### 3.5.2 The Partitioning Algorithm for Asymmetric Data Exchange Costs

With the expansion of Equation 3.9 comes an immediate benefit of being able to determine function placements in relation to one another. Knowing whether two immediate functions are co-located or distributed (i.e., separated between public cloud and private premises), and if data goes from the function in the public cloud to the function on premises or vice versa, we are able to formulate the asymmetric data exchange charges for a cloud deployment into the partitioning problem (cf. Contribution 4 in Section 1.4). As discussed earlier, public cloud providers have no monetary charges when data goes into their cloud data centers but they apply charges to the data that leaves their data centers. Being able to extract and encode inbound and outbound data charges into the partitioning algorithm allows for smarter monetary cost optimizations when applying partitioning to a DDG.
3.5. Applying the Partitioning Algorithm

Benefiting from the expansion in Equation 3.9, asymmetric billing charges can be added to the communication cost of Equation 3.4 as follows:

\[
\text{cost}'_{\text{comm} u,v} = (e_{uv} + e'_{uv}) \times \text{latency}_{cost(u,v)} + \frac{d_{u \rightarrow v}}{D_{\text{unit}}} \times \text{cost}_{\text{out comm} \text{unit}} \times e'_{(u,v)} + \frac{d_{v \rightarrow u}}{D_{\text{unit}}} \times \text{cost}_{\text{in comm} \text{unit}} \times e_{(u,v)}
\]  

(3.11)

where \(d_{u \rightarrow v}\) stands for data transfer from the public cloud to the premises with \(x_u\) being in the cloud and \(x_v\) being on-premises, and \(d_{v \rightarrow u}\) represents data transfer from cloud to the premises where \(x_u\) is on-premises and \(x_v\) is in the cloud. Furthermore, \(\text{cost}_{\text{out comm} \text{unit}}\) represents cloud data charges when data leaves the cloud and \(\text{cost}_{\text{in comm} \text{unit}}\) represents cloud data charges when data enters the cloud. This formulation, combined with separation of outgoing and ingoing data to an entity during application profiling leads to cost optimization of the target software service for a hybrid deployment. Following the changes above, the objective function of Equation 3.10 can be updated by replacing \((e_{uv} + e'_{uv})\text{cost}_{\text{comm} u,v}\) with \(\text{cost}'_{\text{comm} u,v}\). In our evaluations, we refer to this new IP formulation as the Asymmetric IP.

As a concrete example of how the asymmetric algorithm works within the context of DayTrader, we provide the code partitioning of Figure 3.3 as suggested by the execution of the asymmetric algorithm when applied to DayTrader’s DDG. Figure 3.3 shows a snippet of the DDG for the method doAccount. Let us assume that the accountprofile database table is constrained to stay on-premises, the dominant performance bottleneck is the round trip network latency, and each edge in the graph is exercised only once. Given that premises resources are more expensive than cloud resources and there are asymmetric data exchange costs for data transfer between the cloud and the premises, in the scenario of Figure 3.3, the optimal deployment needs to reduce data transfer over the network and minimize the number of functions deployed on premises. In a symmetric partitioning algorithm separating updateAccountData from doAccount (7 KB of data exchange) is preferred over cutting executeQuery (13 KB of data exchange). In an asymmetric partitioning however, the algorithm will assign no costs for inbound data to the cloud. As such, cutting executeQuery to be pushed to the premises has equivalent cost overhead (1 KB) compared to cutting updateAccountData. However, there is a
3.6 Evaluation

Figure 3.3: DDG and data exchange for the sub-graph of doAccount in DayTrader. Black nodes represent software functions in the graph.

gain in cutting executeQuery in that by pushing only this method to the premises, all other code entities in the DDG (two left branches of updateAccountData) can be placed in the cloud, benefiting from scalable and cheaper resources.

3.6 Evaluation

In this section we evaluate the contributions discussed earlier in this chapter, i.e. context-sensitive dependency modeling, flexible cost modeling, and asymmetric data exchange costs. In our evaluations, we need to analyze whether or not our models capture the behavior of the system effectively. Proper modeling of the system would allow for the results of the partitioning algorithms to match the real-world behavior of the software under analysis. We need to analyze if our context-sensitive model provides advantages over the other existing models and whether or not the asymmetric cost models and the cost-to-latency analysis provide any additional advantage when analyzing the system. To evaluate all these questions, we have formalized the evaluation points into the following questions:

1. How do the partitioning algorithms help with optimal hybrid deploy-
3.6. Evaluation

1. How does the partitioning of web applications? (cf. Section 3.6.1).

2. How are our dependency models comparable to real world deployment of the applications under the scenario of full cloud deployments or hybrid deployments? (cf. Section 3.6.2).

3. How does our context-sensitive dependency modeling compare to the request-based or static structure dependency models when used in application partitioning for hybrid cloud? (cf. Section 3.6.3).

4. How do our cost model and the cost-to-latency ratio ($\gamma$) contribute to an optimal partitioning of web applications for hybrid cloud deployment (cf. Section 3.6.4).

5. How does application partitioning affect the scalability of the system? (cf. Section 3.6.5).

6. How does application partitioning affect the overall deployment costs of the system? (cf. Section 3.6.6).

7. How is symmetric application partitioning compared to asymmetric application partitioning? (cf. Section 3.6.7).

8. How does the number of tables constrained to be on premises affect the partitioning performance and monetary costs? (cf. Section 3.6.8).

We evaluate our contributions discussed in this chapter using two different applications: Apache DayTrader [5] and JForum [6]. DayTrader (as discussed in Chapter 3) is a Java implementation benchmark of a stock trading system and has already been used for evaluation purposes in previous cloud computing research [64, 100]. JForum, on the other hand, is a widely used real world open source discussion board implemented in Java. It allows users with different levels of privilege to contribute to discussions in a forum by logging in (validateLogin), creating profiles (profile), browsing forums (show), listing posts (list), inserting/storing posts (insert, insertSave), quoting posts (quote), editing posts (edit, editSave), bookmarking posts (bookmark), etc. Compared to DayTrader which consists of 63 Java classes and almost 6200 LOC, JForum is a bigger application with 347 Java classes and 25.6K LOC. Additionally, JForum consists of 36 database tables and supports close to 100 different business logic functionality.
3.6. Evaluation

<table>
<thead>
<tr>
<th>Application</th>
<th>Business Logic Functionality</th>
<th>~ Node# in DDG</th>
</tr>
</thead>
<tbody>
<tr>
<td>DayTrader</td>
<td>doSell</td>
<td>299</td>
</tr>
<tr>
<td></td>
<td>doAccountUpdate</td>
<td>114</td>
</tr>
<tr>
<td></td>
<td>doQuotes</td>
<td>62</td>
</tr>
<tr>
<td></td>
<td>doAccount</td>
<td>79</td>
</tr>
<tr>
<td></td>
<td>doPortfolio</td>
<td>73</td>
</tr>
<tr>
<td></td>
<td>doBuy</td>
<td>293</td>
</tr>
<tr>
<td></td>
<td>doLogin</td>
<td>176</td>
</tr>
<tr>
<td></td>
<td>Total</td>
<td>1096</td>
</tr>
<tr>
<td>JForum</td>
<td>validateLogin</td>
<td>345</td>
</tr>
<tr>
<td></td>
<td>edit</td>
<td>450</td>
</tr>
<tr>
<td></td>
<td>quote</td>
<td>430</td>
</tr>
<tr>
<td></td>
<td>profile</td>
<td>310</td>
</tr>
<tr>
<td></td>
<td>list</td>
<td>570</td>
</tr>
<tr>
<td></td>
<td>insertSave</td>
<td>1200</td>
</tr>
<tr>
<td></td>
<td>editSave</td>
<td>500</td>
</tr>
<tr>
<td></td>
<td>insert</td>
<td>430</td>
</tr>
<tr>
<td></td>
<td>show</td>
<td>350</td>
</tr>
<tr>
<td></td>
<td>bookmark</td>
<td>430</td>
</tr>
<tr>
<td></td>
<td>Total</td>
<td>5015</td>
</tr>
</tbody>
</table>

Table 3.2: The number of nodes in the DDG of DayTrader as well as the top ten largest JForum business logic functionality.

Due to the large scale of JForum, we limit our evaluation of JForum to the top 10 business logic functionality with the most number of code entities in their dependency graphs. Table 3.2 shows the selected business logic functionality as well as the number of nodes in the DDG of each business logic functionality for both Apache DayTrader and JForum.

Given the two type of algorithms suggested for application partitioning, we evaluated DayTrader and JForum in the following four deployments: i) both code and data are deployed to the premises (Private premises); ii) data is on-premises and code is in the cloud (Naïve-Hybrid); iii) data is on-premises and code is partitioned using a symmetric partitioning algorithm (Symmetric-Split-Code); and iv) data is on-premises and code is partitioned using the asymmetric partitioning algorithm (Asymmetric-Split-Code). Also in order to compare our simulations with real world deployments of cloud platforms we have provided deployments where both code and data are in the cloud (Full-Cloud).
3.6. Evaluation

We used the following setup for the evaluation: for the premises machines, we used two 3.5 GHz dual core machines with 4.0 GB of memory, one as the application server and another as our database server. Both machines were located at our lab in Vancouver, and were connected through a 100 Mb/sec data link. For the cloud machines, we used a large EC2 instance with 2 EC2 Compute Units and 3.75 GB of memory as our application server and another large instance as our database server. Both machines were leased from Amazon’s US West region (Oregon) and were connected by a 1 Gb/sec data link. For the operating system, all the machines ran Ubuntu 13.04 with 64bit Java Virtual Machine v.7. Our Web applications ran on top of the Jetty Web server v.8 [8] and used Oracle Database 11g Express Edition [9] as the database servers. The only point of variability among all machines in our deployments was the underlying hardware as we kept the software stack the same across all the machines.

For all the experiments in this section, unless otherwise stated, we ran all the experiments in this section a minimum of five times. Also, we assumed the cost of a large EC2 instance to be $0.32/hour. With the assumption for the cloud machine cost to be 80% cheaper than the premises machine cost, we set the cost of leasing a premises machine to be $1.50/hour. We also assume the cost of data transfer to be $0.12 per gigabyte where data is going from the cloud to the premises, and $0 per Gigabyte where data is going from the premises to the cloud. Our cloud cost schemes match those of Amazon EC2 [2].

3.6.1 Micro-Benchmarks on Performance Improvements

We started evaluating our approach by performing micro-benchmarks on how applying partitioning affects the overall behavior of applications compared to a Naïve-Hybrid deployment. As discussed earlier, the main advantage of doing fine-grained profiling and context sensitive modeling is to track inter-code and code-to-data dependencies at a higher level of details and see how partitioning affects the total number of round-trips or amount of data going over the network for each of the transactions in the application. As a reference consider Figure 3.3 of Section 3.5 (also shown below). In the micro-benchmarks presented here, we evaluate how applying any of the partitioning algorithms could change the overall behavior of the application for each request type in terms of inbound-

---

5We assume fixed flat rate monthly charges for bandwidth.
3.6. Evaluation

Figure 3.4: A high-level model of how different partitioning algorithms would choose the placement of code and data components between the public and the private cloud. For each of the three lines, code elements and database tables falling below the line are considered to be placed on premises and all the other elements are placed in the public cloud.

/ou8tbound data exchange, and the number of network round-trips caused by separating software functions and database tables in a target application.

We omit any micro-benchmark evaluation for *Private premises* deployments since in a full premises deployment there is no data exchange or network round-trips. In order to collect micro-benchmark results for each transaction in the applications we developed a network sniffer software that would track number of connections and size of data exchanged between peers of a hybrid cloud deployment.

**Network Roundtrips**

In our first set of evaluations we measured changes in the number of network round-trips when moving from a *Naïve-Hybrid* deployment of JForum to a partitioned version of it (either *Symmetric Split-Code* or *Asymmetric Split-Code*). A network round-trip was measured when the source and the target of communication were located on two different machines, one on the private premises and the other in the public cloud. In a network
### 3.6. Evaluation

Round-trip the request would travel from the source machine to the target machine and the response would return back with the request being either a remote procedure call request or a database request. Network round-trips are particularly important as they cause performance degradation to the distributed deployment of a software system and it is desirable for them to be as minimal as possible.

<table>
<thead>
<tr>
<th>Deployment</th>
<th>Request Type</th>
<th>Database Roundtrips</th>
<th>Http Roundtrips</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Naïve Hybrid</strong></td>
<td>validateLogin</td>
<td>6</td>
<td>0</td>
<td>6</td>
</tr>
<tr>
<td></td>
<td>edit</td>
<td>7</td>
<td>0</td>
<td>7</td>
</tr>
<tr>
<td></td>
<td>quote</td>
<td>9</td>
<td>0</td>
<td>9</td>
</tr>
<tr>
<td></td>
<td>profile</td>
<td>5</td>
<td>0</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>list</td>
<td>9</td>
<td>0</td>
<td>9</td>
</tr>
<tr>
<td></td>
<td>inserSave</td>
<td>17</td>
<td>0</td>
<td>17</td>
</tr>
<tr>
<td></td>
<td>editSave</td>
<td>10</td>
<td>0</td>
<td>10</td>
</tr>
<tr>
<td></td>
<td>insert</td>
<td>3</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>show</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>bookmark</td>
<td>3</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td></td>
<td>70</td>
<td>0</td>
<td>70</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Deployment</th>
<th>Request Type</th>
<th>Database Roundtrips</th>
<th>Http Roundtrips</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Symmetric Split-Code</strong></td>
<td>validateLogin</td>
<td>3</td>
<td>2</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>edit</td>
<td>1</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>quote</td>
<td>0</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>profile</td>
<td>3</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>list</td>
<td>3</td>
<td>2</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>inserSave</td>
<td>6</td>
<td>3</td>
<td>9</td>
</tr>
<tr>
<td></td>
<td>editSave</td>
<td>4</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>insert</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>show</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>bookmark</td>
<td>3</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td></td>
<td>24</td>
<td>18</td>
<td>42</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Deployment</th>
<th>Request Type</th>
<th>Database Roundtrips</th>
<th>Http Roundtrips</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Asymmetric Split-Code</strong></td>
<td>validateLogin</td>
<td>6</td>
<td>0</td>
<td>6</td>
</tr>
<tr>
<td></td>
<td>edit</td>
<td>2</td>
<td>3</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>quote</td>
<td>2</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>profile</td>
<td>2</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>list</td>
<td>3</td>
<td>2</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>inserSave</td>
<td>5</td>
<td>3</td>
<td>8</td>
</tr>
<tr>
<td></td>
<td>editSave</td>
<td>3</td>
<td>2</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>insert</td>
<td>5</td>
<td>2</td>
<td>7</td>
</tr>
<tr>
<td></td>
<td>show</td>
<td>3</td>
<td>2</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>bookmark</td>
<td>0</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td></td>
<td>23</td>
<td>17</td>
<td>40</td>
</tr>
</tbody>
</table>

Table 3.3: Microbenchmarks on the number of round-trips forDataBase and RPC invocations between the public cloud and the private infrastructure for various application deployments to the cloud.

As shown in Table 3.3, moving from a Naïve Hybrid deployment to a Symmetric-Split-Code or Asymmetric-Split-Code deployment reduces the average number of overall round-trips between the private cloud and the
3.6. Evaluation

The table shows that even though having a hybrid deployment results in extra HTTP message exchanges between the public and the private cloud, it reduces the number of database round-trips by 65.7% through placing code and data closer to one another.

**Average Data Exchanges**

In another benchmark, we measured the amount of data exchange between the public cloud and the private infrastructure for every request type. Figure 3.5 shows this measured data. In the figure, *Http-In* implies average amount of application HTTP traffic data entering the public cloud and *Http-Out* shows average amount of application HTTP traffic data leaving the cloud. Similarly, *DB-In* and *DB-out* show the average amount of database traffic data entering and leaving the cloud, respectively. Given the fact that public cloud providers impose monetary charges based on the amount of data leaving the cloud, an effective optimization strategy would try to minimize the amount of outgoing data for the public cloud providers.

As Figure 3.5 shows, the *Symmetric Split-Code* algorithm reduces the overall amount of data exchange between the cloud and the private premises by 36% with a reduction of 33.2% in the total amount of outgoing data from the cloud. On the other hand, the *Assymmetric Split-Code* algorithm reduces the total amount of data exchange by 11.4%. However, there is a reduction of 49.8% in the amount of outgoing data from the cloud to the private premises. The *Assymmetric Split-Code* has 28.2% more data exchange compared to the *Symmetric Split-Code* algorithm, however when it comes to the amount of outgoing data, the *Assymmetric Split-Code* has nearly 25% less data transferred out of the cloud. Given that outgoing data exchanges impose extra monetary costs to a hybrid cloud deployment, the *Assymmetric Split-Code* algorithm would result in more cost savings for the deployment. It should be noted in the figure that, for the case of a Naïve Hybrid deployment (the bottom two pairs of bars in Figure 3.5), no HTTP inbound or outbound data is observed. This is due to the fact that the Naïve Hybrid deployment assumes a separation of code components from data entities and does not separate code components from one another and hence does not generate any HTTP traffic.

Table 3.4 shows more details on the amount of inbound and outbound http/database data exchange for each request type in JForum. The table shows that, for 9 out of 10 request types analyzed, partitioning of the code
3.6. Evaluation

Figure 3.5: Inbound and Outbound data exchanges for Naïve Hybrid versus partitioned deployment models

using the Split-Code algorithms results in observed HTTP network traffic. However, the overall amount of data exchange reduces when a partitioning algorithm is used for hybrid deployment. Also, as the table shows, an asymmetric data exchange algorithm, while increasing data transfer inbound to the public cloud, reduces the size of data leaving the cloud. This justifies the extra cost savings we observe when using the asymmetric partitioning algorithm compared to the symmetric algorithm (cf. Section 3.6.7).

To summarize, the analyses show that, compared to a naïve hybrid deployment, both symmetric and asymmetric split-code partitioning result in smaller number of network round-trips and also less data transfer between the public and the private clouds. Asymmetric split-code partitioning achieves more savings by decreasing the size of cloud out-bound data transfer.
Table 3.4: Microbenchmarks on the amount of data exchanged between the public cloud and the private cloud for various hybrid deployments.

3.6.2 Cost Models vs Measured Deployments

In order to verify the accuracy of the cost models, we compare the execution and data transfer measurements of the generated dependency models to those of real deployments for DayTrader. Since our profiling data was collected on the premises machine, the models and the real deployment for the machine on premises are identical. In a hybrid or a full cloud deployment, the models are generated by applying linear fitting techniques as described in Section 3.4. We compared the generated cost models with the real deployment of the DayTrader application for settings where the deployment is fully in the cloud (Full-Cloud) or when it is partitioned using the asymmetric algorithm (Asymmetric-Split-Code). Figure 3.6 compares
the results averaged over 1000 requests to each request type over 5 runs.

![Execution Time / Request (milliseconds)](image)

**Figure 3.6**: Simulated and measured execution times for hybrid and cloud code placements for each DayTrader request type

As can be seen from Figure 3.6, the execution time for the generated models in case of the *Asymmetric-Split-Code* are 81.3% similar in the average accuracy to the actual deployments ($\sigma=0.13$) while models for *Full-Cloud* deployments are 86.1% similar ($\sigma=0.052$) compared to a practical deployment. Similarly, we compared the actual data transfer to the modeled data transfer (cf. Figure 3.7). For data transfer the *Asymmetric-Split-Code* model provides 87.4% average estimation accuracy compared to the real deployment ($\sigma=0.061$) while for a *Full-Cloud* deployment we get 86.3% average estimation accuracy ($\sigma=0.076$).

In summary, the results show a variance of ±15% for our generated models compared to the real world deployments of the application. The results verify that partitioning of the real-world applications based on the simu-
lated models would lead to having real-world hybrid deployments whose cost and performance measures fall within boundaries of estimated cost and performance for the simulated models. In the following sections, we discuss real deployments of DayTrader and JForum, and as results follow, we verify that using cost models of this accuracy can provide benefits in actual deployments.

Figure 3.7: Simulated and measured data transfer sizes for hybrid and cloud code placements for each DayTrader request type

3.6.3 Evaluation of Context-Sensitive Modeling

Next, we evaluate the three dependency models from Section 3.2 to see which one contributes to a performant deployment under varied premises cost and fixed cloud cost. We used MANTICORE to decide about function placements for DayTrader as the cost of deployment to a premises machine linearly changes from $0.16 per hour to $2.5 under an expected load of 100 requests per second. For each cost value, we took the mapping of function executions suggested by MANTICORE for each of the models and physically
deployed it. We then measured the average perceived latency across all the requests to see how the partitioning affects the overall performance of the system. Results are shown in Figure 3.8. The line at the top of the graph shows the full deployment of the code to the cloud whereas the bottom line indicates a full premises deployment.

We observe that for premises costs greater than $0.16/hour, SSM yields to a full deployment to the cloud (cf. Figure 3.8). This is because this model does not account for function replication and thus, separation of code elements would imply introduce network round trips and increase the communication latency compared to a full cloud deployment. To prevent this, upon premises cost increase, the analysis framework chooses to have all the code in the cloud and only pay for the communication latency when

Figure 3.8: Comparison of latency adjustments for the SSM, RBM, and CSM as the premises cost changes.
3.6. Evaluation

it comes to retrieving data from database tables.

For RBM, the dependency model consists of subgraphs separated by request types and as a result there is freedom in the model to choose code placements based on request types. Compared to the SSM, this model tolerates changes to the cost of deployment on premises and compensates between cost and performance by pushing the code for different request types from premises to the cloud to the point where all the code is in the cloud. Figure 3.8) shows that for RBM, a full cloud deployment happens when premises charges exceed $0.50/hour.

In CSM, the fine level of granularity in the model allows for replication of code while enabling the partitioning algorithms to find edges with the lowest performance overhead to be cut. As a result, unlike SSM and RBM, cuts made in the CSM do not solely separate code from data but also separate code units with low communication overhead from one another thus improving costs of the deployment. In CSM, a full cloud deployment happens when premises charges exceed $2.15/hour which leads to higher performance in the deployment.

To summarize Figure 3.8 when cost of deployment to premises is cheap (left part of the graph), all three models choose to have the code deployed on premises. Similarly, when cost of deployment to the premises is expensive, all three models choose to push all the code to the cloud. However, between these two ends of the spectrum, CSM provides latency aware deployments by not only separating code from data but also by separating code units with lower performance overhead from one another. As a concrete example, for the premises resource cost of $0.48/hour, a CSM-based deployment has 46% less latency compared to SSM and 25.3% less latency compared to RBM.

3.6.4 Evaluation of Flexible Cost Modeling

Performance degradation is one of the biggest concerns when it comes to separation of code from data or code from code for a distributed application deployment. The degradation is concerned with the extra latency added to the overall execution process as a matter of having the data sent and received over the network. In this section we show how by changing $\gamma$ from Equation 3.5 we allow for latency and cost to be traded for one another.
3.6. Evaluation

With a measured median round-trip communication latency of 15ms between the cloud machine and our premises machine over 100 trials, we linearly increased $\gamma$ from 0.5 to 15 (i.e. $\times 30$) to see how it affects function placement decisions and the corresponding deployment costs. To measure deployment costs, we set $\alpha = 5$ representing 80% cost saving (cf. Equation 3.3). Table 3.5 shows code placement decisions made by the partitioning framework for different request types in Apache DayTrader.

As shown in Table 3.5, with $\gamma = 0.5$, the partitioning algorithm chooses to have all functions in the cloud (Naïve-Hybrid) where the deployment would be the cheapest (low $\gamma$ favors monetary cost over latency). As $\gamma$ increases, the algorithm pushes more functionality to the premises in order to accommodate a performant deployment, either by making a cut in the code (Asymmetric-Split-Code) or by pushing the entire code for a request type to the premises (Private-premises). Figure 3.9 shows the increase in cost of deployment as the functions are pushed to the premises. As we already discussed in Section 3.6.4, cost of deployment is increased as the partitioner moves more code entities to the premises.

Comparing each deployment with its predecessor and successor deployments illustrates that changing $\gamma$ for the partitioning algorithm can balance between cost and performance of the deployed application. For example, the deployment when $\gamma$ equals 5 is (on average) 50.9% more expensive than the successor deployment suggested when $\gamma$ equals 1.5, yet its average performance is only 21.7% slower. However, the deployment suggested for when $\gamma$ equals 5, compared to its predecessor deployment when $\gamma$ equals 15, is on average 24.6% slower but 2.6% more cost effective.

In nutshell, $\gamma$ allows for system architects to establish a relation between

<table>
<thead>
<tr>
<th>RequestType</th>
<th>Deployment choice for varied cost-to-latency ratio ($\gamma$)</th>
<th>0.5</th>
<th>1.5</th>
<th>5</th>
<th>15</th>
</tr>
</thead>
<tbody>
<tr>
<td>doLogin</td>
<td>Naïve-Hybrid</td>
<td>Asym-Split-Code</td>
<td>Asym-Split-Code</td>
<td>Private-premises</td>
<td></td>
</tr>
<tr>
<td>doBuy</td>
<td>Naïve-Hybrid</td>
<td>Asym-Split-Code</td>
<td>Asym-Split-Code</td>
<td>Private-premises</td>
<td></td>
</tr>
<tr>
<td>doPortfolio</td>
<td>Naïve-Hybrid</td>
<td>Private-premises</td>
<td>Private-premises</td>
<td>Private-premises</td>
<td></td>
</tr>
<tr>
<td>doAccount</td>
<td>Naïve-Hybrid</td>
<td>Naïve-Hybrid</td>
<td>Private-premises</td>
<td>Private-premises</td>
<td></td>
</tr>
<tr>
<td>doQuotes</td>
<td>Naïve-Hybrid</td>
<td>Naïve-Hybrid</td>
<td>Private-premises</td>
<td>Private-premises</td>
<td></td>
</tr>
<tr>
<td>doAccountUpdate</td>
<td>Naïve-Hybrid</td>
<td>Naïve-Hybrid</td>
<td>Private-premises</td>
<td>Private-premises</td>
<td></td>
</tr>
<tr>
<td>doSell</td>
<td>Naïve-Hybrid</td>
<td>Asym-Split-Code</td>
<td>Asym-Split-Code</td>
<td>Private-premises</td>
<td></td>
</tr>
</tbody>
</table>

Table 3.5: Code placement for different request types in Apache DayTrader as Gamma ($\gamma$) changes from 0.5 to 15.
3.6. Evaluation

Figure 3.9: Monetary cost of deploying requests of various type for DayTrader with respect to changes in Gamma ($\gamma$)

the cost of the deployment and the level of perceived latency. The higher the value of $\gamma$, the higher the cost of latency and the higher the chances for collocation of code and data. Considering $\gamma$ as a tuning knob, software developers are able to decide about the proper value of $\gamma$ for their distribution model based on the latency and cost policies of their system.

3.6.5 Evaluation of Scalability with Code-only Partitioning

We also performed a scalability analysis for DayTrader to see how different code placement choices affect application throughput. DayTrader comes with a client workload generator that models user behaviors for browsing and purchasing stocks. For the deployment tests, we used a range of 500 to 4000 simulated clients over a period of 5 minutes with 1 minute ramp-up time, and measured throughput. Figure 3.10 shows the throughput of the system under varied user load. As shown in the figure, the machine on the premises riches the highest throughput of 384 req/sec when the number of user threads sending requests is 2500. For 2500 user threads, Private-
3.6. Evaluation

*premises* has 18% better throughput compared to both *Naïve-Hybrid* and *Asymmetric-Split-Code* deployments. Once we passed this threshold, the premises machine got overloaded and was unable to properly handle incoming requests to the point that for 4500 user threads *Naïve-Hybrid* and *Asymmetric-Split-Code* deployments reached 32% better throughput compared to the *Private-premises* deployment.

![Graph showing scalability tests](image)

Figure 3.10: Scalability tests for full premises, full cloud, and hybrid deployments

Finally, we notice that the hybrid and cloud deployment had similar scalability. However, as shown in Section 3.6.3 a hybrid deployment using our *CSM* provides better response time when the CPU is not a bottleneck. So for the DayTrader case study there is extra advantage in using partitioning for hybrid cloud deployment over simple public cloud or private premises deployments.
3.6. Evaluation

3.6.6 Evaluation of Deployment Costs

Next, we evaluate the effects of combined code and data partitioning on the estimated monetary deployment costs (see [67]) of DayTrader and JForum (Figures 3.11 and 3.12) under various deployment models. For DayTrader, both symmetric and asymmetric partitioning algorithms generate the same partitioning suggestions and thus Symmetric-Split-Code and Asymmetric-Split-Code overlap.

As shown in both graphs, a full deployment of the Web applications on-premises (Private-premises) results in a deployment cost increase linear to the increase in premises machine costs, rendering such deployments expensive. As shown in Figure 3.11, in case of DayTrader with premises machines costing $0.5/hour, the premises deployment is 38% cheaper than Naïve-Hybrid, 34% cheaper than Symmetric-Split-Code, and 31% cheaper than Asymmetric-Split-Code. However, for the premises resource costs of $2.0/hour, premises deployment is 35% more expensive than any of the hybrid deployments.

In cases of Symmetric-Split-Code and Asymmetric-Split-Code, an increase in costs for machines on-premises results in the algorithm pushing more code to the cloud to the point where all the code is in the cloud and all the data is on-premises. In such a situation, both Symmetric-Split-Code and Asymmetric-Split-Code eventually yield to a deployment identical to the one suggested by Naïve-Hybrid; that is, pushing all the code to the cloud where all the data inevitably needs to be on-premises.

For JForum the trend in deployment costs (see Figure 3.12) is similar to DayTrader. With cheap premises resource costs, premises deployment is the cheapest of all deployments. However linear increase in cost of premises resources increases the cost of deployments linearly as well to the point that it becomes more expensive than hybrid deployments. For a premises cost of $1.5/hour, the Symmetric-Split-Code and Asymmetric-Split-Code deployments are 15% cheaper than Private-premises and Naïve-Hybrid. But eventually all three hybrid deployments converge by pushing all the code to the cloud while keeping all the data on premises. Also, there are differences between the deployment costs for JForm and DayTrader. Since JForum is a larger application (4.6 times more functions in its dependency graph), the overall cost of deploying JForum for each model is comparatively higher than for DayTrader. Furthermore in JForum, partitioning of code and data
3.6. Evaluation

results in fine-grained transition of code modules from the premises to the cloud and as a result changes in premises cost lead to changes in overall deployment linearly. However, in DayTrader, due to the smaller size of the application (4.6 times smaller), less code is moved to the cloud. Finally, as shown in Figure 3.12, when we only partition code, there is a difference of 11% between the costs of deployment for Symmetric-Split-Code and Asymmetric-Split-Code. This is because, due to the size of JForum, any optimization that results in relocating code and data deploys more code in the cloud and benefits from cloud resource costs.

When measuring the costs of deployment, we rely on the assumption that resource costs on premises grow at a higher rate compared to the cost

![Graph](image)

Figure 3.11: Comparison of monthly deployment costs for different deployment models of DayTrader. The cost is measured by detailed profiling of application resource usage for a 10 minute period over 5 runs, interpolating resource usage to a month time of deployment and then applying the cost metrics from the cost scheme to the measured resource usages.
3.6. Evaluation

Figure 3.12: Comparison of monthly deployment costs for different deployment models of JForum. The cost is measured by detailed profiling of application resource usage for a 10 minute period over 5 runs, interpolating resource usage to a month time of deployment and then applying the cost metrics from the cost scheme to the measured resource usages.

of resources in the cloud. In fact, in the years since the start of Amazon Web Services, the resource costs for the public cloud have only gone down. With growing costs of the premises cloud, deployments dependent to the premises cloud experience an increase in the overall deployment costs. However, the results show that using partitioning algorithms for hybrid deployments lessens the dependency of a deployment to premises resources, and as such their deployment costs are less affected by a growth on premises deployment costs.
3.6. Evaluation

3.6.7 Evaluating Symmetric vs. Asymmetric Partitioning Algorithms

In this section we evaluate how symmetric and asymmetric partitioning algorithms affect data exchange patterns and so lead to different partitioning choices. As discussed in Section 3.5.2, the asymmetric algorithm improves performance by loosening dependencies between code and data entities. The loosening of dependencies happens by reducing the monetary charges to data exchange costs (i.e., edge weights) in the dependency graph and thus allowing the partitioning algorithm to separate graph nodes from one another with less penalty. This allows for code entities tightly bound together, due to their data dependencies, to freely be partitioned. However, for the asymmetric algorithm to work effectively, the application under analysis needs to satisfy one of the following conditions: i) either it needs to be data intensive, implying that there needs to be heavy data dependencies between its software entities (code & data), or ii) its business logic functionality need to be computation intensive so that relocation of their code entities would result in performance changes.

DayTrader and JForum, as 3-tier Web applications do not fall within the category of data intensive applications. Moreover, as we already showed, DayTrader is a small application (see Table 3.3) and relocation of its code entities does not necessarily imply changes in its performance (cf. Figure 3.11). For JForum, however, relocation of code entities may result in cost changes. As shown in Figure 3.12 in JForum these cost changes can result in better performance and cheaper deployments in a Asymmetric-Split-Code compared to Symmetric-Split-Code.

Figure 3.13 shows how the partitioning decisions made using the asymmetric algorithm have different data exchange patterns compared to the symmetric algorithm. In particular, Figure 3.13 shows that even though partitioning using the asymmetric algorithm results in increased data transfer to the cloud by a factor of ×1.5, it reduces the amount of data leaving the cloud by a factor of ×0.25, and more importantly it allows for relocation of code entities from the premises to the cloud. In JForum this results in a cost saving of 11% when using Asymmetric-Split-Code compared to using Symmetric-Split-Code.

In a nutshell, the results show that the asymmetric partitioning algorithm achieves improvements in cost by reducing the size of data outbound
3.6. Evaluation

Figure 3.13: Measured data exchanged between the cloud and premises after using the Symmetric vs. Asymmetric partitioning algorithms of Section 3.5.2. In each bar, the left part of the bar shows data inbound to the cloud and the right part of the bar shows data outbound to the cloud. Requests partitioned using the Asymmetric Split-Code, have a smaller outbound data exchange (the right part of the bar) compared to when those request are partitioned using the Symmetric Split-Code.

to the cloud. Even though Asymmetric Split-Code partitioning may result in larger volumes of data going to the cloud, due to the lower cost of inbound data transfer, the overall deployment will be cheaper in cost.
3.6. Evaluation

3.6.8 Evaluating Data Entity Placement Constraints

So far, we have made specific assumptions about which database tables are constrained to be on-premises for DayTrader and JForum. Our criteria for constraining database tables have been security and privacy concerns that may lead to data protection issues. In this section we show how different choices for table placement constraints can change the deployment models and associated costs.

To do so, for DayTrader, we started by having all five tables of accountprofile, account, holding, order, quote placed in the cloud. Then following the order shown above, we constrained the tables to be pushed to the premises one-by-one to the point where all the tables were placed on-premises. Figure 3.14 shows how the placement for code entities change as more data entities are pulled to the premises.

As shown in Figure 3.14 when all the tables are in the cloud, all the code is also pushed to the cloud. However, as tables are pinned to the premises, code entities with tightly coupled data constraints are migrated to the premises. We noticed when we constrained accountprofile to be on-premises, its data dependencies with the account table pulled this table to the premises as well. For this reason, in Figure 3.14 the data shown, when only the accountprofile table is pinned to the premises, is identical to when both tables accountprofile and account are pinned to the premises.

Our cost analysis estimations for Asymmetric-Split-Code deployments with variable number of constrained tables is shown in Table 3.6. The results show that by only moving the quote table to the cloud, a cost saving of 20% can be achieved compared to where all the tables are on premises. On the other hand, pinning only accountprofile and account to the premises would result in 33% cost savings compared to a full pinning of all tables to the premises. However, placement choices of holding and order did not appear to affect the overall deployment costs.

The results from this evaluation show that minimizing data placement constraints contributes to the deployment costs. More importantly, our analysis shows that proper data placement decisions for particular data entities (e.g., quote in DayTrader) affects the overall cost of deployment. This further necessitates the need for in-depth analysis of code and data
dependencies when doing hybrid cloud deployments. We show in Chapter 4 how a thorough analysis of code and data dependencies combined with a cross-tier partitioning of code and data can lead to improvements in overall cost and performance of a hybrid deployment.

3.7 Summary

In this chapter, we proposed an extension to existing application partitioning techniques to provide for hybrid deployment of web applications. The evaluation on DayTrader showed that the new approach can contribute towards an optimized hybrid cloud deployment. In particular, it showed that: we are able to reduce network round-trips and data exchange outbound to the cloud by using partitioning algorithms (cf. Section 3.6.1); the costs of a
3.7. Summary

<table>
<thead>
<tr>
<th># Tables Pinned to premises</th>
<th>Deployment Cost ($/Month)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1032.86</td>
</tr>
<tr>
<td>2</td>
<td>1032.86</td>
</tr>
<tr>
<td>3</td>
<td>1109.12</td>
</tr>
<tr>
<td>4</td>
<td>1230.38</td>
</tr>
<tr>
<td>5</td>
<td>1539.16</td>
</tr>
</tbody>
</table>

Table 3.6: Deployment costs for DayTrader as the number of tables pinned to the premises change.

hybrid deployment extrapolated from monitoring a single-host test version of the service were at least 81.3% accurate (cf. Section 3.6.2); the context-sensitive modeling of service behavior provided a better representation to optimize placement of software function execution (cf. Section 3.6.3); our formulation of the objective function for optimization allows developers to tune the tradeoff between end-to-end round-trip time and deployment costs (cf. Section 3.6.4); and that a hybrid deployment while providing similar scalability as a full cloud deployment offers better round-trip latency (cf. Section 3.6.5), and is cheaper overall (cf. Section 3.6.6).

Throughout this chapter we made specific assumptions about the nature of web applications we analyzed that affect our process of profiling, modelling, and partitioning. Here we describe some of those assumptions, enumerate threats to validity, and describe solutions or potential directions for future research:

- We assumed that web applications are collections of stateless software functions. This implies that none of the software functions in the application should keep an internal state (i.e. a precedent event) to which the forthcoming sequence of interactions depend. While this assumption holds for the web applications we analyzed, there could be situations where the statelessness constraint is violated by some software functions in the application. Our current strategy to handle such cases is by manually inspecting and identifying functions in the application that keep internal state and avoiding their replication. As part of the future work this could be further automated by benefitting from techniques such as static analysis \[85\] or dynamic data dependency analysis in order to automate the process of identifying stateful software functions. The internal states then can be either converted
3.7. Summary

into a shared state to be accessed by different replicas of a software function, or such stateful software functions can be constrained to not get replicated.

- For the deployment machines, we assumed all the machines in the cloud to be homogeneous. As such we assumed that the overall behaviour of the application once replicated from one instance to another will stay the same as long as those instances are identical in their type. However, existing research shows that the assumption of homogeneity of replicated instances, while true in theory, does not hold in practice. As part of our future work we aim to look into techniques to encode heterogeneity of machines into the modelling and partitioning algorithms to allow for more accurate hybrid deployment of software applications in the cloud.

- The partitioning algorithms discussed in this chapter only allow for two-way partitioning of web applications. In our case it is useful because it allows for optimal partitioning between the public cloud and the private infrastructure. However, two-way partitioning relies on an assumption related to the previous point, i.e., that all machines in a given partition are homogeneous. It is due to this assumption on homogeneity that we can have estimations on the deployment costs for machines in each partition. As mentioned above, this assumption may not hold in the real world. As such, we intend to look into the existing algorithms on multi-way partitioning and investigate how their combination with modelling of heterogeneous machines could improve cost and performance in hybrid cloud deployments.

- Finally, for our cost model we focused on capturing the cost of using CPU resources and the cost of data transfer. While our current cost model is in line with the cost metrics provided by cloud providers, we acknowledge that the cost of having a hybrid deployment can also depend on other factors such as the cost of data storage, the cost of cooling, CO2 emission, etc. Even though our current cost model does not deal with these extra cost parameters, the flexibility encoded into our cost model allows these parameters to be declaratively included into the model. These factors then either can be related to performance metrics we collect from the application or can be added as a constant to the overall cost of deployment, where they will come into effect during the optimization process for hybrid deployment.
Chapter 4

Data Dependency Modeling and Data-tier Partitioning

In a real-world software system, partitioning for a hybrid cloud deployment requires reasoning on an optimal placement of software functions and database tables across all request types in the web application. In the previous chapter, we discussed how a combination of code dependency modeling and application-tier partitioning would allow for separation of software functions from one another. In this chapter we explain how application-tier partitioning is augmented with data-tier partitioning in order to improve on performance and cost for a hybrid-cloud deployment.

To motivate combined application-tier and data-tier partitioning, we again use the DayTrader doLogin example as a running scenario throughout this chapter. Figure 4.1 shows the output of our cross-tier partitioning for doLogin. The figure shows the call-tree of function executions in the application-tier and dependencies between the application-tier and the data-tier. For each database query, the data-tier dependencies are represented in the form of a dependency graph, known as a query plan. The query plan provides an ordered set of steps for how data from database tables is fetched in order to respond to the target query. In a query plan, database operands (i.e., tables) and operators (e.g., join or select) form the nodes of the graph and their relations within the context of the target query form the edges.
Figure 4.1: A cross-tier partitioning suggested by our tool for the doLogin request from DayTrader showing a partitioned application- and data-tier: nodes from the data-dependency graph on premises (black square nodes), nodes from the data-dependency graph in the cloud (white square nodes), functions from the code-dependency graph on premises (black circle nodes), and functions from the code-dependency graph in the cloud (white circle nodes). Here we only show the full dependency graph for doLogin.
Chapter 4. Data Dependency Modeling and Data-tier Partitioning

In Figure 4.1 (also at a higher level in Figure 1.1), we see four categories of components in 3 regions separated by dashed lines:

- Nodes from the data-dependency graph placed on premises, shown as black squares in Regions (A) and (C) of the figure.
- Nodes from the data-dependency graph placed in the cloud, shown as white squares in Region (B) of the figure.
- Functions from the code-dependency graph placed on premises, shown as black circles in Region (A) of the figure.
- Functions from the code-dependency graph placed in the cloud, shown as white circles in Region (B) of the figure.

We use each of these four categories to motivate cross-tier partitioning and achieve the following objectives:

First, some data is not suitable for deployment in the cloud due to privacy concerns or regulations. Thus, many enterprises need to avoid committing deployment of certain data in the public cloud, and instead host it on private premises infrastructure. For example, in DayTrader, given that user information is privacy sensitive, tables `account` and `accountprofile` in Figure 4.1 are constrained to stay on premises (black square nodes in Regions (A) and (C) of the figure).

Second, function execution requires CPU resources which are generally cheaper and easier to scale in the public cloud (some reports claim 80% savings using public cloud versus on-premises private systems). Thus placing function execution in the public cloud is useful to limit the amount of on-premises infrastructure. So without regard to other factors, we would want to execute application-tier functions in the cloud. For the example of Figure 4.1 this would include the majority of functions involved in the `doLogin` request type (i.e., white circle nodes in Region (B)).

Third, since we would like to deploy functions to the cloud, the associated non-sensitive data, tightly bound to those functions, should be deployed to the cloud, otherwise we will incur excessive latency and bandwidth usage. So there is motivation to move non-sensitive data to the cloud. However, such non-sensitive data may be bound to sensitive data through queries which operate over both. For this reason, moving
non-sensitive data to the public cloud is not always a winning proposition. We will need an analysis which, on one hand, could reason about the benefit of moving data closer to functions executing in the public cloud, and on the other hand, would consider the drawbacks of pulling non-sensitive database away from the sensitive data on premises. Figure 4.1 shows how the HOLDINGS table is marked as non-sensitive and thus is allowed to be placed in the cloud (i.e., white square nodes in Region (B) of the figure).

Finally, executing all functions in the public cloud is not always beneficial. Some functions are written as transactions over several data resources. Such functions may incur communication overhead if they execute in the public cloud but operate on private premises data. Furthermore, choices on function placements are going to have a propagating effect on deciding about the benefits of where to place non-sensitive data and vice-versa. So the benefit of executing functions in the cloud or on the premises needs to be balanced with the incurred communication overhead caused by placement decisions made for corresponding data. In Figure 4.1 black circle nodes of Region (A) show how constraining account and accountprofile tables to the premises infrastructure pulls the code closely dependent to those database tables to the premises infrastructure as well.

These four objectives help to illustrate the inter-dependencies between the application-tier and the data-tier. In the case of doLogin of Figure 4.1, a developer may manually arrive at a similar partitioning with only minor effort. The developer might be able to look into factors such as number of round-trips over communication links (i.e., edges in the graph), CPU usage for each function (i.e., nodes in the graph), and privacy-sensitivity of each node in the graph, and decide about about how to make a cheaper hybrid partitioning of the graph. However, in a real-world software application with hundreds to thousands of request types (each possibly containing hundreds to thousands of function and data nodes in their dependency graphs), partitioning an entire application requires developers to simultaneously reason about the effects of component placement across all request types. Without proper tooling and automation, the process of dependency analysis and partitioning will not be manageable.

This chapter is summarized and published in the Usenix Middleware Conference 2013 [68]. In the rest of this chapter we explain how we extend application-tier partitioning with data-tier partitioning to achieve a full cross-tier partitioning strategy for hybrid cloud deployment. In the ex-
4.1 Extending Application-tier Partitioning with Data-tier Partitioning

ample application of DayTrader, through cross-tier partitioning, we will be able to take the dependency graph of Figure 3.2, augment it with extra information on data dependencies, and apply our partitioning algorithms, in order to achieve a full dependency graph for application and data tiers, similar to the one in Figure 4.1. In Section 4.1 we provide an example scenario of augmenting application-tier partitioning of Section 3.2.3 for doLogin with data-tier partitioning to show how cross-tier partitioning improves cost and performance savings over a code-only partitioning strategy. In Section 4.3 the process of collecting profiling data for data entities are explained. Section 4.4 discusses our analysis to capture data interactions and dependencies across database tables in the software system. In Section 4.5 we elaborate on the mathematical formulation of the partitioning problem for the data-tier and how it can be combined with the application-tier to form cross-tier partitioning. Finally in Section 4.6 we evaluate the effectiveness of our approach through an analysis of benchmark software systems and their partitioning and deployment in hybrid cloud using our framework.

4.1 Extending Application-tier Partitioning with Data-tier Partitioning

We take the following steps when extending application-tier partitioning to integrate the data-tier: (i) weighing the benefits of distributing queries, (ii) comparing the trade-offs between join orders, (iii) taking into account intra-request data-dependencies, (iv) taking into account inter-request data-dependencies and (v) providing a query execution model comparable to application-tier function execution. We focus on a data-tier implemented with a traditional SQL database. While some web application workloads can benefit from the use of alternative NoSQL techniques, we chose to initially focus on SQL due to its generality and widespread adoption.

1. Weighing the Benefits of Distributing Queries. As described earlier, placing more of the unconstrained data in the cloud will allow for the corresponding code from the application-tier to also be placed in the cloud, thus increasing the overall efficiency of the deployment and reducing data transfer. However, this can result in splitting the set of tables used in a query across public and private locations. With the constrained tables kept on the premises and unconstrained tables moved to the cloud, any query operating on a combination of these two
4.1. Extending Application-tier Partitioning with Data-tier Partitioning

sets of tables would require to be turned into a distributed query. Particularly this would result in binary relational operations in a query (e.g., a join operation) to have to work on distributed tables. As such, partitioning of the join operations would lead to trade-offs between cost and performance similar to the case of code partitioning. The challenge for partitioning of data-dependency graphs is to find the right balance between having the minimum number of round-trips and data transfer between the cloud and the premises as well as the maximum use of cheap resources (e.g., CPU and memory) from the cloud. For our example of DayTrader, each user can have many stocks in her holdings which makes the HOLDING table quite large in size. As shown in Figure 4.1, in an optimal splitting of the join operation, the HOLDINGS table can be pushed to the cloud (white square nodes) to eliminate the traffic of moving its data to the cloud. This splitting also maintains our constraint to have the privacy sensitive ACCOUNT table on the private premises. An effective modeling of the data-tier needs to help the BIP optimizer reason about the trade-offs of distributing such queries across the hybrid architecture.

2. Comparing the Trade-offs Between Join Orders. The order in which tables are joined can have an effect not only on traditional processing time but also on round-trip latency. Throughout this section, we use the running example of the query shown in Figure 4.2 with two different join orders, left and right. If the query results are processed in the public cloud where the HOLDING table is in the cloud and ACCOUNT and ACCOUNTPROFILE are stored on the private premises, then the plan on the left will incur two round-trips between the public and the private clouds for distributed processing. On the other hand, the query on the right only requires one round-trip. While both queries are feasible to be processed by the database engine, it can be instructed to choose the plan incurring less round-trip and latency. Modeling the data-tier should help the BIP optimizer reason about the cost of execution plans for different placements of tables and accordingly instruct the database engine to execute the better performing query. The aim is to extend the capabilities of database engines to execute distributed queries with extra guidance on execution of database operations to achieve improved cost and performance.

3. Taking into Account Intra-request Data-dependencies. A web application is a collection of tens to hundreds of request types with each
4.1. Extending Application-tier Partitioning with Data-tier Partitioning

Figure 4.2: Two possible query plans from one of the queries in DayTrader: 

```sql
SELECT p.*, h.* FROM holding h, accountprofile p, account a WHERE h.accountid = a.accountid AND a.userid = p.userid AND h.quote_symbol = ? AND a.accountid = ?. The figure shows two different execution plans for the same query where in (a) first holding and account are joined and the result is then joined with accountprofile while in (b) account and accountprofile are joined prior to being joined with holding.
```

request type implementing a business logic functionality in the web application (e.g., doLogin or doAccount in Apache DayTrader). Due to the stateless nature of web applications, software functions involved in the execution of each request type and their data access patterns can be realized independently from other request types. Across all application request types, some can execute more than one query to access data. In these cases, it may be beneficial to partition functions internal to the request type into group executions with data at a single location. Such grouping helps to eliminate latency overhead otherwise needed to move data to the location where the application-tier code executes. An example of this is shown in Figure 4.1, where a sub-tree of function executions for TradeJdbc:login is labeled as “private” (black circle nodes). By pushing this sub-tree to the private premises, the computation needed for working over account and accountprofile data in the two queries under TradeJdbc:login can be completed at the premises without multiple round-trips between locations.

4. Taking into Account Inter-request Data-dependencies. While code dependencies can be analyzed internal to each request type in a web application (see the requirement above on intra-request data dependencies), data dependencies need to be analyzed across different request types. This is due to the fact that there is a single database shared across all request types that stores or supplies data from/to the code in each request type. As a result, while decisions on place-
4.2 High-Level Overview of Data-tier Partitioning

As shown in Section 1.3 of Chapter 1, the overall process of applying cross-tier partitioning involves the following three high level steps:

- Profiling
- Analysis
- Partitioning

In Chapter 3, we described how the three steps of profiling, analysis, and partitioning are applied to the application-tier of a web application. In this chapter, we describe how profiling, analysis, and partitioning are applied to the data-tier, with the next sections in this chapter elaborating on the details of each step. It should be noted that the main goal in our data partitioning approach is to decide about the proper placement of database tables and not to invent a new query optimization technique.

To do data-tier partitioning, we could have tried to implement a query optimizer for a hybrid cloud setting. However, that would have required...
4.2. High-Level Overview of Data-tier Partitioning

developing a database query optimization engine and re-inventing a lot of existing algorithms for query optimization. Instead of re-inventing the wheel we chose to implement a layer of abstraction on top of traditional query optimization engines that performs the following tasks: 

\textbf{i)} Collecting profiling data on implications of changing the orders of execution for join orders in a query plan 

\textbf{ii)} Analyzing the collected profiling data using the hybrid cloud optimizer which considers the implications of executing join orders across distributed tables, 

\textbf{iii)} Utilizing optimization techniques to chose the query plan whose distribution of join orders has the minimum overhead on performance and cost, and 

\textbf{iv)} Placing database tables following the selection of the optimal query plans for a hybrid cloud setting.

Figure 4.3 summarizes the steps involved in our data-tier partitioning process.

![Diagram](image)

Figure 4.3: The high-level steps in performing data-tier partitioning.

For profiling, unlike the case of code partitioning, where CPU usage and data exchange are evaluated at the level of software functions, for the data-tier these performance footprints are collected by monitoring the database engine and how it works on operations (e.g., join, selection, and projection) as well as operands (i.e., database tables) involved in a query.

Dependency analysis, as explained in Section 4.1, involves analyzing all applicable models of executing a query. Given a query, the order of applying operations involved in a query might be changeable. In this case, the analyzer would examine all different variations for the query plan, and collect statistics on each instance of the query plan. The collected data is then fed to the partitioner in order to create the mathematical formulation based on the identified query plans.
4.2. High-Level Overview of Data-tier Partitioning

The partitioning phase involves creating the constraints and the objective function for the optimization algorithm. The constraints consist of the business-level constraints as well as the constraints internal to the execution of join orders in a query plan. The business level constraints should assure the placement of nodes (i.e., operations and operands in the data dependency graph) of the chosen query plan either on premises or in the cloud. They also ensure satisfaction of constraints on cost of deployment or expected performance. The internal constraints for a query plan ensure exclusivity of query plans such that out of all possible variations for a given query, eventually only one is suggested as the optimal plan.

Finally, the generated constraints and objective function should be merged with the constraints and objective functions defined for the code dependency graph. The mathematical formulation of the partitioning algorithm for the code dependency graph makes assumptions about the value of variables representing node placements in the cloud or on the premises (see Section 3.5). Those same assumptions should hold for the data dependency graph and its mathematical formulation as well. The novelty of our approach is that instead of optimizing to a specific join order in isolation of the structure of application-tier execution, we encode the possible orders together with the BIP of the application-tier as a combined BIP.

As a high-level example for how the query plan execution works, let us consider a query plan joining three tables \( A, B, \) and \( C \) with two possible orders of executing joins: \( ((A \bowtie B) \bowtie C) \) and \( (A \bowtie (B \bowtie C)) \). We encode the possibilities for the execution of the query plans into a query tree similar to the one shown in Figure 4.4. The left subtree in the tree represents the first possible join and the right one represents the second possible join.

Our partitioning of the data-tier using the query plan requires this tree for every query plan to be created and encoded into a BIP formulation. The combined BIP formulation from all query plans defines the problem whose solution provides the optimized partitioning of the data-tier. Steps for the creation of the BIP formulation for this query tree consist of the following parts:

1. **Collecting Operator Status**: Identifying alternative execution of join orders in a given query plan. For the example of Figure 4.4, it is to identify that joining \( A, B, \) and \( C \) can be achieved through either \( ((A \bowtie B) \bowtie C) \) or \( (A \bowtie (B \bowtie C)) \)
4.2. High-Level Overview of Data-tier Partitioning

Figure 4.4: The query plan tree representing alternatives in executing a given query joining three tables A, B, and C with one another.

2. Generating Choices: Ensuring mutual exclusivity of alternative query plans. This implies that out of \(((A \bowtie B) \bowtie C)\) and \((A \bowtie (B \bowtie C))\) only one of them is supposed to be chosen by the analyzer.

3. Generating Input Constraints: Ensuring that selection of \((A \bowtie (B \bowtie C))\) requires its inputs, i.e., \((B \bowtie C)\) joined with A to also be chosen.

Further to this, for each join operation in the query plan the following constraints have to be met:

1. At Most One Location Placement: for every child in a join operation, there should be a single placement either in the cloud or on the premises. That is, for each table or join operation in Figure 4.4 it should happen on premises, in the cloud, or it should be ignored. That is, if \(((A \bowtie B) \bowtie C)\) is chosen then there should be placement decisions for A, B, and C as well as \((A \bowtie B)\); however \((B \bowtie C)\) should be canceled out of the BIP formulation.

2. Generate Execution Cost: For each child of a join operation, the cost of its executions should be considered.

3. Generate Communication Cost: For each child of a join operation, the cost of its data exchange should be considered.

In the following sections we go into the details of how each step of profiling, analysis, and partitioning for the data-tier is performed.
4.3 Profiling the Data-tier with EXPLAIN PLAN

The process of data-tier partitioning starts with Profiling the Data-tier with EXPLAIN PLAN as highlighted in Figure 4.5.

Figure 4.5: The first phase in partitioning the data-tier is to profile the data-tier with EXPLAIN PLAN.

Profiling information is available for query execution through the EXPLAIN PLAN SQL command. Given a particular query, this command provides a tree-structured result set detailing the execution of the query. We use a custom JDBC driver wrapper to collect information on the execution of queries. During application profiling (cf. Chapter 3) whenever a query is issued by the application-tier, our JDBC wrapper intercepts the query and collects the plan for its execution.

From the extracted statistics on the executed query, we are interested in the plan of execution for the query and its execution footprint. In particular we are interested to collect data on how much time it takes for the query to execute, what is the measured CPU usage and data exchange after query execution, and finally, the detailed plan on the execution of the query (i.e., the query plan). The query plan consists of a list of nodes, i.e., operations and operands, and also the order in which operations (i.e., joins, selections, or projections) are applied to the operands (i.e., database tables). The extracted plan shows only the pattern of execution for the recently issued query but it can also help extract alternative plans of execution for a query. In a query plan, we use the term join orders to refer to the order in which the join operators are composed.

From the plan returned by the database, we extract the following information:
4.3. Profiling the Data-tier with Explain Plan

1. **type**(op): Each node in the query plan is an operator such as a join, table access, selection (i.e. filter), sort, etc. We leverage the database’s own cost model directly by recording from the provided plan how much each operator costs. Hence, we don’t need to evaluate different operator implementations to evaluate their costs. On the other hand, we do need to handle joins specially because table placement is greatly affected by their ordering.

2. **cpu**(op): This statistic gives the expected time of execution for a specific operator. In general, we assume that the execution of a request in a hybrid web application will be dominated by the CPU processing of the application-tier and the network latency. So in many cases, this statistic is negligible. However, we include it to detect the odd case of expensive query operations which can benefit from executing on the public cloud.

3. **size**(op): This statistic captures the expected number of bytes output by an operator which is equal to the expected number of rows times the size of each retrieved row. From the perspective of the plan tree-structure, this is the data which flows from a child operator to its parent.

4. **predicates**(joinOp): Each join operator combines two inputs based on a set of predicates which relate those inputs. We use these predicates to determine if alternative join orders are possible for a query.

The process of collecting profiling data is conducted through the execution of the web application and collecting statistics. The data collected from execution of identical queries are aggregated and normalized in order to achieve an estimate average on cost and performance footprints. As a result the CPU usage and data size collected are in fact aggregates of all executions of a unique query plan.

As discussed earlier, when profiling the application, the profiler observes and collects execution statistics only for plans that get executed but not for its alternative join orders. However, the optimal plan executed by the database engine in a distributed hybrid deployment can be different from the one observed during a centralized non-distributed profiling of the application. When optimizing the deployment, it is important to be able to examine all the alternative plans and choose the one that results in better optimization. For example, in a centralized execution of the query in
Figure 4.4. Analyzing Dependencies for the Data-tier

Figure 4.2 both executions of the query may result in identical execution footprints. However, in a distributed execution, the query of Figure 4.2b is preferred over that of Figure 4.2a due to the reduced number of network round-trips. This makes the plan of Figure 4.2b the optimal plan in a distributed execution of the query.

In order to make the BIP partitioner aware of alternative orders, we have extended our JDBC wrapper to consult the database engine and examine the alternatives by utilizing a combination of EXPLAIN PLAN and join order hints. Our motivation is to leverage the already existing cost model from a production database for cost estimation of local operator processing, while still covering the space of all query plans. The profiler also captures which sets of tables are accessed together as part of an atomic transaction. This information is used to model additional costs of applying a two-phase commit protocol, should the tables get partitioned. The collected metrics for data exchange and CPU usage are then fed through the same cost models explained in Chapter 3 to provide equivalent monetary cost models to those of the application-tier for the data-tier.

4.4 Analyzing Dependencies for the Data-tier

Once the profiling data is collected using EXPLAIN PLAN, we start the second phase of Query Plan Dependency Analysis as highlighted in Figure 4.6.

Figure 4.6: The second phase in partitioning the data-tier is to analyze dependencies in a query plan.

As discussed in the previous section, we need to encode enough information in the BIP so it can reason over all possible plans. Otherwise, the BIP optimizer would mistakenly assume that the plan executed during our
4.4. Analyzing Dependencies for the Data-tier

initial profiling is the only one possible. For example, during initial profiling on a single host, we may only observe the plan from Figure 4.2a. However, in the example scenario, we saw that the plan in Figure 4.2b introduces fewer round-trips across a hybrid architecture. We need to make sure the right plan is accounted for when deciding about table placement. Our strategy to collect the necessary information for all plans consists of two steps: (i) gather statistics for all operators in all plans irrespective of how they are joined, and (ii) encode BIP constraints about how the operators from step (i) can be joined. Here we describe step 1 and then describe step 2 in the next subsection.

As is commonly the case in production databases, we assume a query plan to be left-deep [88]. In a left-deep query plan, a join takes two inputs: one from a single base relation (i.e. table) providing immediate input (referred to as the “inner relation”); and another one potentially derived as an intermediate result from a different set of relations (the “outer relation”). The identity of the inner relation and the set of tables comprising the outer relation uniquely determine the estimated best cost for an individual join operator. This is true regardless of the order in which the outer relation was derived [91]. For convenience in our presentation, we call this information the operator’s id, because we use it to represent an operator in the BIP.

As an example, the root operator in Figure 4.2a takes accountProfile as an inner input and \{holding, account\} as an outer input. The operator’s id is then \{(holding, account), accountProfile\}. We will refer to the union of these two inputs as a join set (the set of tables joined by that operator). For example, the join set of the aforementioned operator is \{holding, account, accountProfile\}. Notably, while the join sets for the roots of Figures 4.2a & 4.2b are the same, Figure 4.2b’s root node has the operator id \{(accountProfile, account), holding\} allowing us to differentiate the operators in our BIP formulation. Our task is to collect statistics on the execution of possible join operators with unique ids. As discussed in Section 4.3, this can be done by instructing the database engine to execute other variations of join orders for a given query.

Most databases provide the capability for developers to provide hints to the query optimizer in order to force certain join orders. For example in Oracle, a developer can use the hint LEADING(X, Y, Z, ...). This tells the optimizer to create a plan where X and Y are joined first, then their intermediate result is joined with Z, etc. We use this capability to extract
4.4. Analyzing Dependencies for the Data-tier

statistics for all join orders. For tables in a query that are not referenced in a LEADING hint, the optimizer is still free to choose any order. As mentioned earlier, when collecting statistics on the execution of a query plan, we can also extract information on the type and order of joins executed for the given query plan. This information can be utilized in order to understand other feasible query plans based on the set of observed tables and joins in the original query plan. We have developed an algorithm that allows for extraction and execution of all possible query plans by sending hints to the database engine.

Program 4.1 takes as input a query observed during profiling. In line 2, we extract the set of all tables referenced in the query. Next, we start collecting operator statistics for joins over two tables and progressively expand the size through each iteration of the loop on line 3. The table \(t\), selected for each iteration of line 4 can be considered as the inner input of a join. Then, on line 5 we loop through all sets of tables of size \(i\) which don’t contain \(t\). On line 6, we verify if \(t\) is joinable with the set \(S\) by making sure that at least one table in the set \(S\) shares a join (access) predicate with \(t\). This set forms the outer input to a join. Finally, on line 7, we collect statistics for this join operator by forcing the database to explain a plan in which the join order is prefixed by the outer input set, followed by the inner input relation. We record the information for each operator by associating it with its id.

For example, consider Figure 4.2 as the input \(Q\) to Program 4.1. In a particular iteration of line 5, \(i\) might be chosen as 2 and \(t\) as ACCOUNTPROFILE. Since ACCOUNTPROFILE has a predicate shared with ACCOUNT, \(S\) could be chosen as the set of size 2: \{ACCOUNT, HOLDINGS\}. Now on line 6, explainPlanWithLeadingTables({ACCOUNT, HOLDINGS}, ACCOUNTPROFILE) will get called and the statistics for the join operator with the corresponding id will get recorded.

The bottom-up structure of the program is similar to the classic dynamic programming algorithm for query optimization [91]. However, in our case we make calls into the database to extract costs by leveraging EXPLAIN PLAN and the LEADING hint. The complexity of Program 4.1 is \(O(2^n)\) (where \(n\) is the number of tables); which is the same as the classic algorithm for query optimization [91], so our approach scales in a similar fashion. Even though Program 4.1’s complexity is exponential, queries typically operate on an order of tens of tables making the performance of the program reasonable.
4.5 Partitioning the Data-tier

Program 4.1 Function to collect statistics for alternative query plan operators for the input query $Q$. $\mathcal{P}_i$ is the powerset operator over sets of size $i$.

```plaintext
Function collectOperatorStats(Q)
    tables ← getTables(Q) for $i ← 1$ to $|\text{tables}|$ do
        foreach $t \in \text{tables}$ do
            foreach $S \in \mathcal{P}_i(\text{tables} - \{t\})$ do
                if isJoinable($S$, $t$) then
                    explainPlanWithLeadingRelations($S$, $t$)
                end
            end
        end
    end
end
```

4.5 Partitioning the Data-tier

The last step in data-tier partitioning involves encoding the collected profiling data and dependency information for query plans into a BIP formulation. This is done through Defining the Constraints and the Objective Function as highlighted in Figure 4.7. Having the constraints for the data-tier defined and the objective function described, integration of the data-tier partitioning with the code-tier partitioning is as simple as defining a union set of the constraints and appending the objective functions. We describe defining the constraints in the first subsection and defining the objective function in the second subsection of this section.

![Diagram](image)

Figure 4.7: The third phase in partitioning the data-tier is to encode the partitioning problem into a BIP formulation.
4.5. Partitioning the Data-tier

4.5.1 Defining BIP Constraints for Data Dependencies

Given the statistics for all operators with a unique id, we need to instruct the BIP how they can be composed. The mathematical formulation for the composition is important as it needs to produce the target query and ensure the exclusive selection of the optimized query plan from the set of feasible ones for a given query. Our general strategy is to model each query plan operator, \( op \), as a binary variable in a BIP. The variable will take on the value 1 if the operator is part of the query plan which minimizes the objective of the BIP and 0 otherwise. Each possible join set is also modeled as a variable. Constraints are used to create a connection between operators that produce a join set and operators that consume a join set (cf. Table 4.1).

The optimizer will choose a plan having the least cost given both the optimizer’s choice of table placement and function execution placement (for the application-tier). Each operator also has associated variables \( op_{cloud} \) and \( op_{premises} \) which indicate the placement of the operator. Table placement is controlled by each table’s associated table access operators. The values of these variables for operators in the same query plan will allow us to model the communication costs associated with distributed queries.

Our program to formulate these composition constraints makes use of two helper functions as shown in Table 4.1, namely genChoice and genInputConstraint. When these functions are called by our programs, they append the generated constraint to the BIP that was already built for the application-tier. The first function, genChoice, encodes that a particular join set may be derived by multiple possible join operators (e.g., \{holding, account, accountprofile\} could be derived by either of the root nodes in Figure 4.2). The second function, genInputConstraint, encodes that a particular join operator takes as inputs the join sets of its two children. It ensures that if \( op \) is selected, both its children’s join sets (\( in_{left} \) and \( in_{right} \)) are selected as well, constraining which subtrees of the execution plan can appear under this operator. The “\( \geq \)” inequality in Table 4.1 helps to encode the boolean logic \( op \rightarrow in_{left} \land in_{right} \).

Starting with the final output join set of a query, Program 4.2 recursively generates these constraints encoding choices between join operators and how parent operators are connected to their children. It starts on line 2 by calling a function to retrieve all operator ids which could produce that join set (these operators were all collected during the execution of Program 4.1).
4.5. Partitioning the Data-tier

Function  | genChoice(joinSet, \{op_1 \ldots op_n\})
--- | ---
Generated constraint  | \(op_1 + \ldots + op_n = joinSet\)
Description  | a joinSet is produced by one and only one of the operators \(op_1 \ldots op_n\)

Function  | genInputConstraint(op, \{in_{left}, in_{right}\})
--- | ---
Generated constraint  | \(-2 \times op + in_{left} + in_{right} \geq 0\)
Description  | If \(op\) is 1, then variables representing its left and right inputs (\(in_{left}\) and \(in_{right}\)) must both be 1

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>genChoice</td>
<td>a joinSet is produced by one and only one of the operators (op_1 \ldots op_n)</td>
</tr>
<tr>
<td>genInputConstraint</td>
<td>(-2 \times op + in_{left} + in_{right} \geq 0)</td>
</tr>
</tbody>
</table>

Table 4.1: Constraint generation functions

It passes this information to \texttt{genChoice} on line 3. On line 4, we loop over all these operator ids, decomposing each into its two inputs on line 5. This information is then passed to \texttt{genInputConstraint}. Finally on line 7, we test for the base case of a table access operator. If we have not hit the base case, then the left input becomes the join set for recursion on line 8.

Once the constraints are defined, the next step would be to define the objective function for the BIP formulation.

4.5.2 Defining the BIP Objective for Data Dependencies

Creating the optimization objective function consists of two parts: (i) determining the cost of executing individual operators, and (ii) determining the costs of distributing operators in a hybrid setting. Once all the costs are

Program 4.2 Constraint generation, using functions from Table 4.1. The details for the functions \texttt{getOperatorsForJoinSet, getInputs, sizeof, and left} are not shown but their uses are described in the text.

Function \texttt{createConstraints(joinSet)}
- \texttt{ops} ← \texttt{getOperatorsForJoinSet(joinSet)}
- \texttt{genChoice(joinSet, ops)}
- foreach \texttt{op \in ops} do
  - \texttt{inputs} ← \texttt{getInputs(op)}
  - \texttt{genInputConstraint(op, inputs)}
  - if \texttt{sizeof(left(inputs))} > 0 then
    - \texttt{createConstraints(left(inputs))}
- end
- end

97
measured we need to create a mathematical formulation of those costs and encode it into the objective function.

**Determining the Cost of Executing Operators**

The magnitude of the execution cost for each operator and the communication cost between operators that are split across the network are computed using the cost model introduced in Chapter 3. This accounts for the variation between local execution and distributed execution in that the latter will make use of a semi-join optimization to reduce costs (i.e. input data to a distributed join operator will transmit only the columns needed to collect matching rows). We need to encode information on CPU and data transmission costs into the objective function. Table 4.2 shows functions to generate these constraints. The first constraint specifies that if an operator is included as part of a chosen query plan (its associated id variable is set to 1), then either the auxiliary variable \( op_{cloud} \) or \( op_{premises} \) will have to be 1 but not both. This enforces a single placement location for \( op \). The second builds on the first and toggles the auxiliary variable \( cut_{op_1,op_2} \) when \( op_{cloud} \) and \( op_{premises} \) are 1, or when \( op_{premises} \) and \( op_{cloud} \) are 1.

The objective function itself is generated using two functions in Table 4.3. For each operator \( op \), function \texttt{genExecutionCost} charges to the objective function one of the following three values: (i) the execution cost of the operator on the cloud infrastructure if \( op_{cloud} \) is set to 1, (ii) the execution cost of the operator on the premises infrastructure if \( op_{premises} \) is set to 1, or (iii) zero if neither of the two variables is set to 1. Note that it will never charge both due to the constraints of Table 4.2. The second function charges the communication cost between two operators.

<table>
<thead>
<tr>
<th>Function</th>
<th>\texttt{genAtMostOneLocation}(( op ))</th>
</tr>
</thead>
<tbody>
<tr>
<td>Generated constraint</td>
<td>( op_{cloud} + op_{premises} = op )</td>
</tr>
<tr>
<td>Description</td>
<td>If the variable representing ( op ) is 1, then either the variable representing it being placed in the cloud is 1 or the variable representing it being place in the premises is 1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Function</th>
<th>\texttt{genSeparated}(( op_1 ), ( op_2 ))</th>
</tr>
</thead>
<tbody>
<tr>
<td>Generated constraint</td>
<td>( op_{cloud} + op_{premises} - cut_{op_1,op_2} \leq 1 ) ( \newline ) ( op_{premises} + op_{cloud} - cut_{op_1,op_2} \leq 1 )</td>
</tr>
<tr>
<td>Description</td>
<td>If the variables representing the locations of two operators are different, then the variable ( cut_{op_1,op_2} ) is 1</td>
</tr>
</tbody>
</table>

Table 4.2: Functions for generating objective helper constraints
4.5. Partitioning the Data-tier

<table>
<thead>
<tr>
<th>Function</th>
<th>genExecutionCost(op)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Generated objective component</td>
<td>(op_{\text{cloud}} \times \text{execCost}<em>{\text{cloud}}(op) + op</em>{\text{premises}} \times \text{execCost}_{\text{premises}}(op))</td>
</tr>
<tr>
<td>Description</td>
<td>If the variable representing (op) deployed in the cloud/premises is 1, then charge the associated cost of executing it in the cloud/premises respectively</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Function</th>
<th>genCommCost(op_1, op_2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Generated objective component</td>
<td>(\text{cut}_{op_1,op_2} \times \text{commCost}(op_1, op_2))</td>
</tr>
<tr>
<td>Description</td>
<td>If (\text{cut}_{op_1,op_2}) for two operators (op_1) and (op_2) was set to 1, then charge their cost of communication</td>
</tr>
</tbody>
</table>

Table 4.3: Functions for generating objective function

if the associated \(\text{cut}\) variable was set to 1. In the case that there is no communication between two operators this cost is simply 0.

Program 4.3 takes a join set as input and follows a similar structure to Program 4.2. The outer loop on line 3, iterates over each operator that could produce the particular join set. It generates the location constraints on line 4 and the execution cost component to the objective function on

Program 4.3 Objective generation

```plaintext
Function createObjFunction(joinSet)
    ops ← getOperatorsForJoinSet(joinSet)
    foreach op ∈ ops do
        genAtMostOneLocation(op)
        genExecutionCost(op)
        inputs ← getInputs(op)
        foreach input ∈ inputs do
            foreach childOp ∈ getOperatorsForJoinSet(input) do
                genSeparated(op, childOp)
                genCommCost(op, childOp)
            end
        end
        if sizeof(left(inputs)) > 0 then
            createObjFunction(left(inputs))
        end
    end
end
```
4.5. Partitioning the Data-tier

line 5. Next, on line 7, it iterates over the two inputs to the operator. For each, it extracts the operators that could produce that input (line 8) and generates the communication constraint and objective function component. Finally, if the left input is not a base relation (line 11), it recurses using the left input now as the next join set.

Determining the Cost of Distributing Operators

We extend the previous cost model to account for possible transaction delays. We assume that if the tables involved in an atomic transaction are split across the cloud and the private premises, by default the transaction will be resolved using the two-phase commit protocol.

Performance overhead from atomic two-phase distributed transactions comes primarily from two sources: protocol overhead and lock contention \[47\]. Protocol overhead is caused by the latency of prepare and commit messages in a database’s two-phase commit protocol. Lock contention is caused by queuing delay which increases as transactions over common table rows become blocked. We provide two different strategies to account for the cost of such overhead when formulating our objective function:

- For some transactions, lock contention is negligible. This is because the application semantics do not induce sharing of table rows between multiple user sessions. By this we mean that at any point in time, during the execution of the application, only a single request to the application is tampering with the data at a given row of any database table in the application. For example, in the DayTrader example of Figure 4.1, although ACCOUNT and HOLDINGS tables are involved in an atomic transaction, specific rows of these tables are only ever accessed by a single user concurrently. The reason is that the operation joining the two tables applies its filtering based on the userid of the user making the request. As such, each join operation exclusively accesses the data that belongs to the given user. In such cases, where no table rows are shared between multiple user sessions, we account for the overhead of the two phase commit by charging it to the objective function. This is done by adding the cost of two extra round-trips between the cloud and the private premises to the objective function, one to prepare the remote site for the transaction and another to commit it. This model assumes that no extra request would cause any extra
4.5. Partitioning the Data-tier

delay in applying a two-phase commit or degrading the performance of executing the request. Applying the extra charges to the objective function reduces chances for tables involved in a two phase commit to be distributed across a hybrid deployment.

- In situations where congestion in accessing table rows is expected, the cost of a two-phase commit could be worsened if it is applied to a distributed deployment of tables involved. For such cases where lock contention is expected to be considerable, developers can request that certain tables be co-located in any partitioning suggested by our tool. This prevents locking for transactions over those tables to be delayed by network latency. Since such decisions require knowledge of application semantics that are difficult to infer automatically, our tool provides an interactive visualization of partitioning results. This allows developers to work through different “what-if” scenarios of table co-location constraints and the resulting suggested partitioning.

It should be noted that we have chosen a conservative strategy of assessing data partitioning costs by assuming that tables involved in the same atomic transaction will be resolved using a two-phase commit strategy. In practice, the two-phase commit strategy is to ensure that the ACID properties associated with a transaction are preserved. In theory, unless strong integrity, consistency, and durability is required, some transactions are resolved using strategies other than a two-phase commit, e.g., retry or compensation [62]. In fact, due to the scalability issues with preserving consistency for transactions in large scale web applications, a lot of the new web applications are choosing to provide eventual consistency in favor of performance. With respect to our algorithms, it implies smaller penalties for separating database tables from one another and thus more loose-coupling between tables when it comes to their partitioning.

With the possibility to freely separate database tables form one another, there are better chances for collocation of tightly coupled code and data. Partitioning at the data tier and increased collocation of code and data implies less network round-trip overhead, through which we expect hybrid partitioning of web applications to improve.

Having appended the constraints and objective components associated with query execution to the application-tier BIP, we make a connection between the two. This is done by encoding the dependency between each
function that executes a query and the possible root operators for the
associated query plan. This allows for the optimization formulation from
the application-tier partitioning to be combined with the optimization
formulation from the data-tier partitioning. Solving this combined op-
timization problem results in a deployment that considers optimality of
placement for software functions and database tables at the same time. We
continue to claim that the resulting cross-tier partitioning is efficient in
terms of performance and cost savings compared to partitioning only at the
application-tier of an OLTP-style software system.

In the next Section we show how hybrid deployments of a web application
following the cross-tier partitioning algorithm performs better compared to
a situation where only code is partitioned.

4.6 Evaluation

In this section we present the results of evaluations we performed to assess
the contributions discussed in this chapter. In particular, we provide answers
to the following research questions:

1. How does combined data-tier partitioning and application-tier parti-
tioning affect the performance of the application compared to only
application-tier partitioning, or no partitioning? (cf. Section 4.6.1)

2. How is the overall cost of deployment affected when application-tier
partitioning is combined with data-tier partitioning compared to other
models of application deployment? (cf. Section 4.6.2)

3. How is the scalability of the system affected when cross-tier partition-
ing is used, compared to other models of application deployment? (cf.
Section 4.6.3)

We evaluate the cross-tier partitioning work using two different ap-
plications: Apache DayTrader [5] and RUBiS [4]. DayTrader is a Java
benchmark of a stock trading system. RUBiS implements the functionality
of an auctioning Web site. Both applications have already been used in
evaluating previous cloud computing research [64, 95]. We have carefully
chosen Apache DayTrader and RUBiS for our evaluations, due to their
unique characteristics in representing web applications with requirements
for privacy and scalability at the same time. Even though our deploy-
ments for Apache DayTrader and RUBiS are not totally comparable with
real-world software deployments (due to incomparable network traffic, data size, and transaction frequency), they still allow us to analyze code and data inter-dependencies in an OLTP-style web application with privacy requirements on data placements.

We can have 9 possible deployment variations with each of the data-tier and the application tier being (i) on the private premises, (ii) on the public cloud, or (iii) partitioned for hybrid deployment. Out of all the placements we eliminate the 3 that place all data in the cloud as it contradicts the constraints to have constrained information on-premises. Also, we consider deployments where only data is partitioned as a subset of deployments with both code and data partitioned. Thus our evaluation of code and data partitioning deployments includes deployments where only data is partitioned. The remaining four models deployed for evaluations were as follows: (i) both code and data are deployed to the premises (Private-premises); (ii) data is on-premises and code is in the cloud (Naïve-Hybrid); (iii) data is on-premises and code is partitioned (Split-Code); and (iv) both data and code are partitioned (Cross-Tier). Table 4.4 shows the list of deployments. The main reason for not considering a data-only partitioning deployment for the evaluation is that the benefit of using the public cloud is in providing cheap resources, i.e. CPU, memory, and storage. According to [109], with web applications, the dominating bottleneck is the CPU when horizontally scaling the business logic tier of the application. In a data-only partitioning model, the business logic part of the application would be placed on the private premises, defeating the purpose of hybrid deployments.

For both DayTrader and RUBiS, we consider privacy incentives to be the reason behind constraining placement for some database tables. As such, when partitioning data, we constrain tables storing user information (account and accountprofile for DayTrader and users for RUBiS) to be placed on-premises. The remaining tables are allowed to be flexibly placed

<table>
<thead>
<tr>
<th>Data</th>
<th>premises</th>
<th>Cloud</th>
<th>Partitioned</th>
</tr>
</thead>
<tbody>
<tr>
<td>premises</td>
<td><strong>Private-premises</strong></td>
<td><strong>Naïve-Hybrid</strong></td>
<td><strong>Split-Code</strong></td>
</tr>
<tr>
<td>Cloud</td>
<td>n/a</td>
<td>n/a</td>
<td>n/a</td>
</tr>
<tr>
<td>Partitioned</td>
<td>n/a</td>
<td>n/a</td>
<td>Cross-Tier</td>
</tr>
</tbody>
</table>

Table 4.4: The map of naming conventions used for different deployment models in the evaluation.
4.6. Evaluation on-premises or in the cloud.

We used the following setup for the evaluation: for the premises machines, we used two 2.8 GHz core i5 machines (quad cores) with 8.0 GB of memory, one as the application server and another as our database server. Both machines were located at our lab in Vancouver, and were connected through a 100 Mb/sec data link. For the cloud machines, we used an extra large EC2 instance with 8 EC2 Compute Units and 7.0 GB of memory as our application server and another extra large instance as our database server. Both machines were leased from Amazon’s US West region (Oregon) and were connected by a 1 Gb/sec data link. We use Jetty as the web server and Oracle 11g Express Edition as the database servers. We measured the median round-trip latency between the cloud and our lab to be 15 milliseconds. Our intentions for choosing these setups is to create an environment where the cloud offers the faster and more scalable environment. To generate load for the deployments, we launched simulated clients from a 3.0 GHz quad core machine with 8 GB of memory located in our lab in Vancouver. To ensure validity of the collected results, all the experiments mentioned in this section are executed for 10 minutes and are repeated a minimum of five times. In the rest of this section we provide the following evaluation results for the four deployments described above: execution times (Section 4.6.1), expected monetary deployment costs (Section 4.6.2), and scalability under varying load (Section 4.6.3).

4.6.1 Evaluation of Performance

In this section we analyze how the performance of a hybrid deployment partitioned using a cross-tier algorithm is compared to other methods of deployment discussed earlier in this section. The hypothesis is that applying cross-tier partitioning for hybrid cloud deployment would result in better overall performance in the web applications.

We measured the execution time across all request types in DayTrader and RUBiS under a load of 100 requests per second, for ten minutes. By execution time we mean the elapsed wall clock time from the beginning to the end of each request type. Figures 4.8 and 4.9 show those with largest average execution times. We model a situation where CPU resources are not exhausted. As shown in Figure 4.8 and Figure 4.9, execution time in cross-tier partitioning is better than any other model of hybrid deployment and is comparable to a non-distributed private premises deployment. This
4.6. Evaluation

is due to the fact that, a higher percentage of code and data is co-located in a cross-tier deployment compared to any other hybrid deployment; resulting in a reduced number of network round-trips and improved performance.

![Figure 4.8: Measured execution times for selected request types in the four deployments of DayTrader.]

As Figure 4.8 shows, response time for DayTrader’s doLogin under Cross-Tier deployment is 50% faster than Naïve-Hybrid while doLogin’s response time for Cross-Tier is only 5% slower compared to Private-premises (i.e., the lowest bar in the graph). It can also be seen that, for doLogin, Cross-Tier has 25% better response time compared to Split-Code, showing the effectiveness of cross-tier partitioning compared to partitioning only at the application-tier. Similarly for other business logic functionality, we note that cross-tier partitioning achieves performance improvements when compared to other distributed deployment models. It results in performance measures similar to a full premises deployment. For the case of DayTrader - across all business logic functionality of Figure 4.8 - Cross-Tier results
4.6. Evaluation

![Figure 4.9: Measured execution times for selected request types in the four deployments of RUBiS.](image)

in an overall performance improvement of 56% compared to \textit{Naïve-Hybrid} and a performance improvement of around 45% compared to \textit{Split-Code}. By making data-partitioning possible and allowing for code and data to be co-located, we see improvements in performance. Compared to a code-only partitioning of the web application, the performance is improved since data-partitioning allows for moving less privacy sensitive data to the cloud. This leads to further reducing the number of network round-trips otherwise happening between distributed code and data.

We observed similar performance improvements for RUBiS. \textit{Cross-Tier} RUBiS performs 28.3% better - across all business logic functionality of Figure 4.9 - compared to its \textit{Naïve-Hybrid}, and 15.2% better compared to \textit{Split-Code}. Based on the results, cross-tier partitioning provides flexibility for moving function execution to the cloud and can increase performance for a hybrid deployment of an application. Table 4.5 these performance
improvements for DayTrader and RUBiS.

<table>
<thead>
<tr>
<th>Deployments</th>
<th>Cross-Tier</th>
<th>Private-premises</th>
<th>Naïve-Hybrid</th>
<th>Split-Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>DayTrader - doLogin</td>
<td>5% slower</td>
<td>50% faster</td>
<td>25% faster</td>
<td></td>
</tr>
<tr>
<td>DayTrader - Overall</td>
<td>9% slower</td>
<td>56% faster</td>
<td>45% faster</td>
<td></td>
</tr>
<tr>
<td>RUBiS - Overall</td>
<td>27% slower</td>
<td>42% faster</td>
<td>24% faster</td>
<td></td>
</tr>
</tbody>
</table>

Table 4.5: Response time evaluation of the Cross-Tier deployment compared to other models of deployment (a) for doLogin and (b) across all business logic functionality. Results show how a deployment based on cross-tier partitioning improves or degrades response time compared to the other models of deployment.

### 4.6.2 Evaluation of Deployment Costs

Next, we evaluate the effects of cross-tier partitioning on the estimated monetary deployment costs of DayTrader and RUBiS (Figure 4.10) under various deployment models. The hypothesis is that we would see an improvement for the cost of deployment when the hybrid deployment is driven by a cross-tier partitioning algorithm.

For computing monetary costs of deployments, we use parameters taken from the advertised Amazon EC2 service where the cost of an extra large EC2 instance is $0.48/hour and the cost of data transfer is $0.12/GB for data transfer inbound to the cloud and $0.0/GB for data transfer outbound to the cloud. To evaluate deployment costs, we apply these machine and data transfer costs to the performance results from Section 4.6.1, scale the obtained deployment costs to one month cost of deployment, and experiment with different ratio of premises-to-cloud deployment costs to assess the effects of varying cost of private premises on the overall deployment costs.

As shown in both graphs, a Private-premises deployment of web applications results in a cost increase linear to the increased cost of machines on premises, rendering such deployments inefficient. In contrast, all partitioned deployments of the applications result in optimal deployments (a logarithmic increase in cost with the cost increase for machines on premises) with Cross-Tier being the cheapest deployment. For a cloud cost 80% cheaper than the private-premises cost (5 times ratio), DayTrader's
4.6. Evaluation

Figure 4.10: Monthly cost comparison for different deployments of DayTrader and RUBiS.

Cross-Tier is 20.4% cheaper than Private-premises and 11.8% cheaper than Naïve-Hybrid and Split-Code deployments. RUBiS achieves even better cost savings with Cross-Tier being 54% cheaper than Private-premises and 29% cheaper than Naïve-Hybrid and Split-Code. As shown in Figure 4.10 and Figure 4.11, in cases where only code is partitioned, an increase in costs for machines on-premises eventually results in the algorithm pushing more code to the cloud to the point where all code is in the cloud and all data is on-premises. In such a situation Split-Code eventually converges to Naïve-Hybrid, i.e., pushing all the code to the cloud. Similarly, Cross-Tier will finally stabilize. However since in Cross-Tier part of the data is also moved to the cloud, the overall cost is cheaper than Naïve-Hybrid and Split-Code. Table 4.6 summarizes the overall cost savings of cross-tier deployments compared to other deployment models for DayTrader and RUBiS.

Intuitively, this cost reduction is reasonable. In cross-tier partitioning, by enabling data partitioning, the unconstrained data has freedom to be
4.6. Evaluation

Figure 4.11: Monthly cost comparison for different deployments of RUBiS.

pushed to the cloud. This leads to pushing the dependent code to the cloud as well, which increases utilization of cloud resources.

<table>
<thead>
<tr>
<th>Deployments</th>
<th>Cross-Tier</th>
<th>Private-premises</th>
<th>Naïve-Hybrid</th>
<th>Split-Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>DayTrader - Overall</td>
<td>20.4% cheaper</td>
<td>11.8% cheaper</td>
<td>11.8% cheaper</td>
<td></td>
</tr>
<tr>
<td>RUBiS - Overall</td>
<td>58.2% cheaper</td>
<td>38.8% cheaper</td>
<td>19.1% cheaper</td>
<td></td>
</tr>
</tbody>
</table>

Table 4.6: Cost improvements of Cross-Tier deployment compared to other models of deployment. Results show how a deployment based on cross-tier partitioning improves deployment costs compared to the other models of deployment.

4.6.3 Evaluation of Scalability

We also performed scalability analyses for both DayTrader and RUBiS to see how different placement choices affect application throughput. The aim
4.6. Evaluation

is to show that cross-tier partitioning results in better application throughput due to improved performance. DayTrader comes with a random client workload generator that dispatches requests to all available functionality on DayTrader. On the other hand, RUBiS has a client simulator designed to operate either in the browse mode or the buy mode. In the browse mode of RUBiS, only requests for browsing categories and items in the auctioning web site are dispatched to the web application. In the buy mode however, requests for buying or selling items from each category are also issued.

For both DayTrader and RUBiS we used a range of 10 to 1000 client threads to send requests to the applications in 5 minute intervals with 1 minute ramp-up. For RUBiS, we used the client in buy mode. Results are shown in Figures 4.12 and 4.13. As the figure shows, for both applications, after the number of requests passes a certain threshold of 300 requests/sec, Private-premises becomes overloaded. For Naïve-Hybrid and

![Figure 4.12: Scalability tests for full premises, full cloud, and hybrid deployments for DayTrader.](image)
4.6. Evaluation

![Chart showing scalability tests for full premises, full cloud, and hybrid deployments for RUBiS.](chart)

Figure 4.13: Scalability tests for full premises, full cloud, and hybrid deployments for RUBiS.

*Split-Code* on the other hand, the applications progressively provide better throughput. However, due to the bottleneck when accessing the data on premises, both deployments maintain a consistent increase in throughput. Finally, *Cross-Tier* achieved the best scalability. For 500 requests/sec in RUBiS, the *Cross-Tier* deployment is 30% better than *Private-premises*, 52% better than *Naïve-Hybrid*, and 49% better than *Split-Code*. Similarly for DayTrader with 500 requests/sec, *Cross-Tier* deployment is 17% better than *Private-premises*, 31% better than *Naïve-Hybrid*, and 22% better than *Split-Code*. With a large portion of the data in the cloud, the underlying resources for both code and data can scale to reach better overall throughput for the applications. Despite having part of the data on the private premises, due to its small size the database machine on premises becomes congested at a slower rate and the deployment can keep a high throughput.
4.7 Summary

In this chapter we demonstrated that combining code and data dependency models can lead to cheaper and better performing hybrid deployment of Web applications. In particular, we showed that for our evaluated applications, combined code and data partitioning can achieve up to 56% performance improvement compared to a naïve partitioning of code and data between the cloud and the premises and a more than 40% performance improvement compared to when only code is partitioned. Similarly, for deployment costs, we showed that combining code and data can provide up to 54% expected cost savings compared to a fully premises deployment and almost 30% expected savings compared to a naïvely partitioned deployment of code and data or a deployment where only code is partitioned.

The results presented here demonstrate that a cross-tier partitioning of both code and data improves performance, cost efficiency, and scalability of deployed web applications. Even though the applications presented in our evaluation are benchmark applications, their overall architecture is similar to those of real world software systems to ensure that the results are widely applicable to real world software systems. In particular, the results of our analysis demonstrate that co-location of code and data helps reduce data transfer rate and network round-trips. The results also demonstrate that the entanglement of software functions and database tables directly affects how easily software functions and database tables can be separated from one another for distributed deployment. These characteristics are not specific to the type of applications we analyzed, and in fact are independent from the workload the application perceives or the size of the database used. Consequently, the results discussed in this chapter can be equally applicable to real world deployments of software systems and in particular web applications.

Similar to Chapter 3, next we describe some of those assumptions we made throughout this chapter, enumerate threats to validity, and describe solutions or potential directions for future research:

- We assumed the behaviour of the web application across its different tiers to be deterministic in responding to incoming requests. It was based on this assumption that we were able to model and project cost and performance metrics for the web application for different deployments and under different load. We acknowledge that web applications
could behave nondeterministically when running in multi-threaded, resource constrained environments. While such behaviour may affect our overall modelling, the partitioning and distribution of the web application is independent from how the application is modelled. So a model that allows for better capturing of the nondeterministic behaviour of a web application can still benefit from our partitioning and distribution strategy.

- We relied on the data received from the database engine as part of the metrics to augment and analyzed our generated models of the application. This reliance tied us to the technology used in the web applications when modelling and partitioning. A more realistic approach would be to benefit from a middleware layer that would capture interactions between the application tier and the data tier and provide more accurate metrics for cross-tier partitioning.

- Our focus throughout this dissertation has been on making partitioning decisions based on constraints on data placement. As such, we are not dealing with how the data flows in the application. While constraints on data storage addresses regulatory requirements for hybrid cloud deployments, it does not necessarily prevent privacy sensitive data from flowing to the public cloud. We recognize that understanding the flow of privacy sensitive data requires research on data flow analysis and taint tracking in order to capture which software functions in the application consume privacy sensitive data. These software functions can then be modified to either work on encrypted data or be constrained to stay on premise. Further research in this area is a subject of our future work.
Chapter 5

Manticore: The Partitioning Framework

In the previous chapters, we have described how the combination of code and data partitioning can lead to effective application deployments for hybrid cloud. In particular, in Chapter 3 we looked at combining code dependency analysis and application partitioning to achieve optimal web application deployments for hybrid cloud with the constraint of having all the data on the private premises. In Chapter 4 we extended our approach to include data-tier partitioning and achieve cross-tier partitioning and deployment of web applications for hybrid cloud. We showed that cross-tier partitioning could achieve better performance, cheaper deployment costs, and better scalability compared to other models of hybrid cloud deployment.

During our work on analysis and deployment of applications for hybrid cloud, we came to realize that the process of converting a monolithic software system to its hybrid equivalent is cumbersome without proper tooling. First, a fully automatic partitioning of a hybrid cloud deployment is not feasible. Developers have different preferences for the amount of money they would like to pay and the performance they expect to perceive. As an example, the Internet is full of articles and entries by people that either praise the payment models and service offerings provided by Amazon Web Services (AWS) or express unsatisfaction with their monetary expenses and system performance when using AWS. As such there is no fixed cost-to-performance ratio to be considered the silver bullet of hybrid cloud deployments.

[8] http://blog.carlmercier.com/2012/01/05/ec2-is-basically-one-big-ripoff/
5.1. Manticore Overview

In order to support developers and system architects with efficient hybrid deployment of their software systems, we developed MANTICORE. MANTICORE provides a framework that allows for hybrid-cloud partitioning of software applications to be included as part of the development process. When developing MANTICORE we pursued the following goals:

- **Flexibility.** To allow for flexible and effective encoding of cost and performance preferences for a hybrid cloud deployment into the design and development process of a software system.

- **Modeling.** To facilitate the modeling and simulation of software partitioning, analyze the implications of software partitioning, and correct design decisions prior to an actual deployment of a software system to hybrid cloud.

- **Automation.** To provide automation in the process of partitioning and deployment of the final software system to the hybrid cloud, and hence eliminating the extra engineering effort in software partitioning.

MANTICORE is a semi-automatic hybrid partitioning and deployment framework which integrates all the algorithms and techniques discussed in the previous chapters, and facilitates a more interactive partitioning process for its users. MANTICORE has been integrated closely with the development environment of a software system such that as the software evolves, the framework can perform processing and suggest hybrid partitioning and deployments. Hence, MANTICORE allows for continuous and iterative analysis of a system towards a hybrid deployment. We implemented MANTICORE as an extension into the Java integrated development environment, Eclipse™, due to its extensible and customizable plugin system. The MANTICORE plugin integrates three steps of profiling, dependency modeling, and partitioning to a target cloud environment. In the rest of this chapter, we discuss each of these steps.

5.1 Manticore Overview

MANTICORE provides implementations for each of the three steps of profiling, analysis, and partitioning for both the code-tier and the data-tier of a Java web application.

- **Profiling.** Runtime instrumentation of applications is an effective method for understanding application behavior [66]. The process of
profiling involves injecting extra profiling code into the software and collecting execution traces of the application representing code and data dependencies as well as application resource usage. We have developed a software profiler called \textit{jip-osgi} \cite{4,66} to perform instrumentation and analysis of Java applications. The software profiler allows for the profiling process to happen at the level of Jar files, Java classes, or Java methods, i.e., from coarse-grained to fine-grained. Profiling at different levels of granularity helps software architects balance the trade-off between the level of details they want to capture during the instrumented execution of a software system and the overhead incurred as a matter of having the extra profiling code added to the software. Finer level of granularity implies more instrumentation to be done to more methods in the code, and can lead to lower performance for the target software system. It is left to system architects to decide on the level of granularity of their instrumentation for Jar file and Java classes in the application.

When instrumenting a Java web application, we hook a Java agent to the binary of the target web application, right at the beginning of running the application. The Java agent mines the code for every Java class loaded into JVM’s runtime. Through the loading process, the agent injects profiling code into the beginning and the end of methods in loaded Java classes, collecting information on the overall execution time of each method. This profiling code collects traces on how methods call one another, how much time it takes for a method to be executed, and how much data is transferred from a caller method to a callee. This information allows for creation of the dependency graph during the analysis phase. (cf. Section 5.2).

- Analysis. The data collected from the profiling process is used to create a dynamic dependency graph (DDG) \cite{39,49,78} of the application. We have developed an algorithm that transforms traces collected during the profiling process into a tree representation with CPU usages stored as node weights and data exchange stored as edge weights. Further to the profiling process, during the analysis phase, the architects can decide about the level of granularity of their dependency graph prior to applying any partitioning algorithm to it. The initial dependency graph holds information on performance metrics for the analyzed application, i.e., the execution time of the methods, volumes of data exchange across methods, etc. Any partitioning applied to the dependency graph at this stage would thus contribute to an improved...
5.1. Manticore Overview

performance of the application. In order to do analysis for minimized costs, the analyzer can also augment the dependency graph with a cost model of the target cloud platform where vertex weights and edge weights are updated to reflect on the monetary implications of the execution time and data exchange. Partitioning of this new graph would thus result in deployments that minimize the overall cost of deployments. In MANTICORE, developers can choose to optimize for cost or performance.

Also for enhanced understandability, a visual representation of this dependency model is implemented using the graph visualization tool, Jung [25]. Details of how MANTICORE performs this analysis can be found in Section 5.3.

- **Partitioning.** The partitioning process involves receiving the data dependency graph generated in the analysis phase and providing a mathematical representation of it as a BIP formulation prior to solving it using a BIP solver. The transformation applies all the constraints and policies of table deployments, cost constraints, and placement constraints and generates BIP constraints and a BIP objective function. The sets of formulations are then passed to a BIP solver. The results of the BIP solver allow for the final placement of methods, classes, and jar files in the Java application to be determined. Details on our implementation of the partitioning algorithms in MANTICORE are presented in Section 5.4.

Figure 5.1 shows the series of actions MANTICORE performs in order to go from profiling to partitioning and deployment. From left to right, MANTICORE operates on a Java application connected to an ORACLE database. At the first phase, the jip-osgi agent is hooked to the Java application where it performs byte-code instrumentation on the code and profiles database access patterns using the modified database driver. Next, at the analysis phase, the profiling trace, constraints, and host configurations are processed to generate the dependency model. At the partitioning phase, constraints are enforced to the dependency model and the partitioning algorithm is applied to generate a distribution plan for the distributed deployment of the Java application. Finally, the plan is passed to the deployment engine which takes care of the distributed deployment.

---

117 Other databases can also be supported but in our deployments we particularly used ORACLE due to its capabilities in supporting distribution of database tables.
In the following sections we describe the implementation details of MANTICORE. Each section describes the implementation of the tools corresponding to each phase of profiling, analysis, and partitioning in Figure 5.1.

Figure 5.1: From left to right, Profiling: jip-osgi is hooked to the application; Java bytecode is instrumented and database queries are analyzed; Analysis: cost constraints and host configuration information augment the profiling trace; the level of granularity for the dependency model is decided and the dependency model is generated; Partitioning: the partitioner enforces the constraints and partitions the dependency model; and finally the result of partitioning is encoded into a distribution plan that is used by the deployment engine for a distributed deployment of the Java application.

5.2 The Software Profiling Tool

As mentioned in Chapter 3, for the purpose of profiling we developed a Java software profiler called jip-osgi [7] that allows for instrumentation and analysis of a given software system. To implement the profiling process, MANTICORE implements an eclipse launcher that hooks jip-osgi’s Java profiling agent to a target application. Figure 5.2 shows the implemented launcher. As shown in the VM arguments part of the window, upon choosing the Java Profile launcher (Figure 5.2B), the VM arguments are augmented with the information for the javaagent as well as other required parameters for MANTICORE in order to perform the profiling process (Figures 5.2C.1 and 5.2C.2).
5.2. The Software Profiling Tool

Figure 5.2: The screen shot for the eclipse launcher of the Java profiler. (A) User selects the target Java web application and chooses to modify the runtime environment. (B) The Java Profiler is selected in the runtime environment. (C.1) The required parameters get added to the set of runtime parameters for launching the `javaagent`. As shown in (C.2), the parameters include the path to the `javaagent` library as well as minimum and maximum memory requirements, and information for the database to be used by our modified database driver for collecting information on data exchange.

Once the `javaagent` is hooked to the target Web application, the process of collecting profiling data can be initiated. We refer to the complete profiling trace of the application as a snapshot which is managed using the Snapshot View in MANTICORe. The Snapshot View communicates with the Java profiler agent over a TCP connection and allows for monitoring profiling information on remote hosts as well as the local host. The host and port number for the TCP communication with the profiling server can be defined in the Snapshot view. Through the TCP communication, the Snapshot view can start or stop the profiling process and store the profiling traces of the execution for further analysis and partitioning.
5.3 The Tool for Analysis and Generating the Dynamic Dependency Graph

Upon collection of the profiling trace, the profiling trace is stored in a snapshot folder. Any of the profiling traces in the snapshot folder can be chosen for creating the DDG and analysis. Once a profiling trace is clicked, a dedicated editor view is generated for its dependency graph. However, prior to the generation of the dependency graph, the developer needs to specify the policy model (cf. Section 3.3.1) and the host configuration model (cf. Section 3.3.2) that need to be used when creating the dependency graph. As explained in Chapter 3, the policy model is used to decide on the set of software methods and database tables that need to be included in the DDG. On the other hand, the cost model is used to adjust vertex weights and edge weights in the dependency graph. To enable this selection of the policy constraints and the cost model, MANTICORE implements a Configuration View that allows the software developer to define these constraints and feed them to the system (Figure 5.3A). Both host configuration and the policy model files are XML files that can be imported into the configuration view. Figure 5.3B.1 shows that the Profile XML Trace references the trace file, the Mod Exposer XML references the policy model file, and the Host Config XML reference the host configuration file.

Prior to generating the dependency graph, MANTICORE also enables software developers to select the type of dependency modeling that they want to apply to the DDG of the target software. In Section 3.2, we described different models that we can use in order to generate a DDG of a target system (see Figure 5.3B.2). MANTICORE enables software developers to experiment with various models of generating the DDG and compare the analysis results. The developers can choose from the possible set of Request-Based Model, Static-Structure Model, or Context-Sensitive Model of Section 3.2 when deciding about which model of dependency analysis they want to apply to the collected profiling data. In the particular example of Figure 5.3, the Context-Sensitive Model is chosen as the target applicable model. Once all the proper options are selected, the dependency model can be created by clicking on the Generate Model button (Figure 5.3C) from the Configuration View.
5.4. The Tool for Partitioning the Dynamic Dependency Graph

Once the DDG is successfully created, the visualization of the graph will appear in the model view editor part of the plugin (cf. Figure 5.4). The dependency graph contains the list of methods calling one another as the nodes in the graph and edges showing the call order. The view also contains information on data exchange between methods in the target application (edge weights) and the execution time for each method in the graph (vertex weights). The values for the edge weights and the vertex weights for the graph shown in the editor is calculated based on measurements described earlier in Chapter 3.

Figure 5.3: The screen shot for selecting the profiling trace, the policy constraints, and the host configuration constraints prior to generating the dependency graph. (A) The user select a given snapshot from the snapshot view on the right pane of the framework for analysis. With the snapshot selected, the user provides information on (B.1) collected profiling trace (i.e., snapshot), the constraints files, and the host configuration file, as well as (B.2) the type of coarsening to be applied to the collected profiling trace in the snapshot file. (C) With all the information provided, the user clicks the button to generate the model.

5.4 The Tool for Partitioning the Dynamic Dependency Graph

Once the DDG is successfully created, the visualization of the graph will appear in the model view editor part of the plugin (cf. Figure 5.4). The dependency graph contains the list of methods calling one another as the nodes in the graph and edges showing the call order. The view also contains information on data exchange between methods in the target application (edge weights) and the execution time for each method in the graph (vertex weights). The values for the edge weights and the vertex weights for the graph shown in the editor is calculated based on measurements described earlier in Chapter 3.
5.4. The Tool for Partitioning the Dynamic Dependency Graph

Figure 5.4: The screen shot for selecting the execution cost model, the communication cost model, the partitioning algorithm, and applying partitioning. (A) The dependency graph is generated, (B) The user can provide information on applying charges to the edge weights and node weights in the dependency graph based on the type of cost model selected (i.e., monetary costs or performance), and also the user can select the partitioning algorithm. (C) With all the partitioning information selected, the user can click the button to partition the model.

Once the dependency graph is generated, the Configuration View is augmented with a sub-view on partitioning. In this view, the software developer is able to choose the proper encoding of resource usage costs into the dependency graph. For example, if the software developer is interested in minimizing the execution time of the hybrid system, the costs for edge weights and vertex weights can be set to be the communication time and the execution time of the components in the graph. Alternatively, if the software developer chooses to minimize the monetary costs of a hybrid deployment, edge weights and vertex weights can be set to reflect on monetary costs of deployment (cf. Figure 5.4B). Upon selecting the proper encoding for edge weights and vertex weights in the DDG, the formulations from Section 3.4 are used in order to update the weights in the graph. The details on how these edge weights are calculated can be found in Chapter 3.
Finally, the last step in the partitioning phase is to choose the partitioning algorithm. As discussed in Section 3.5, the work in this thesis is primarily concerned with two models of binary linear programming (BIP) algorithms, i.e., the symmetric and the asymmetric partitioning algorithms.

For the BIP algorithms we used the off-the-shelf integer programming solver lp_solve [10]. The results of the partitioning process are then reflected both on the dependency graph and in the form of a distribution plan. The visualized dependency graph in the plugin is updated with nodes going to the cloud colored in green and nodes staying on premises colored in red (cf. Figure 5.5). This provides a visual view of what nodes are separated from one another and what edges need to be cut. Besides the visualization results, the plugin also generates the partitioning output in the form of a distribution plan shown in the console of the plugin. The distribution plan highlights what modules in what request types in the application are separated from one another and how they need to be distributed. The partitioner also generates an XML version of the distribution plan. Figure 5.5A.1 highlights some of the nodes in the dependency graph whose placements are set to be the private premises (colored red in the figure) and Figure 5.5A.2 shows some of the nodes set to be in the cloud (colored green). Figure 5.5B shows the output of the partitioning and placements algorithms.

Further to the visualization of the partitioning results and the distribution plan, MANTICORE also generates statistical information on the number of modules placed on each host, the cost of execution, the cost of communication, and the total cost of deployment as calculated by the partitioning algorithm. Figure 5.6 shows how this information is presented to the user.

5.5 Distribution and Deployment

The distribution plan generated from partitioning of the dependency graph serves as the input to the Distribution Engine. Since the DDG is augmented with both code and data interdependencies, a cut in the DDG may separate code entities from one another, code entities from data entities, and data entities from one another. Separation of code and data is achievable by accessing the database engine through the database driver. Separating inter-code or inter-data dependencies requires some extra engineering work. For code entities, we have developed a bytecode rewriting
5.5. Distribution and Deployment

Figure 5.5: The screen shot for the results of partitioning and the generated distribution plan. (A) The output of the partitioning process is shown with, (A.1) nodes that are subject to placement on the private premises (colored in red) and (A.2) nodes that are subject to placement in the cloud (colored in green). (B) A text-based representation of the distribution plan is also generated both in plain text and XML to be used for distributed deployment.

...engine as well as an HTTP remoting library that takes the partitioning plan generated by the analyzer, and splits the target application based on the generated plan. Splitting the application happens by identifying methods that will be separated from one another in the distributed deployment of the application, injecting remote invocation (RMI) calls from the caller methods to the callee methods, and serializing data between the callers and the callees. The RMI calls are then given the host addresses where the private and the public portion of the application are deployed.
5.6. Discussion

Figure 5.6: The screen shot for statistical information on partitioning results. The figure shows information on the total size of data going from the private premises to the cloud, total data going from the cloud to the private premises, the overall execution time for the code in the cloud, and the total execution time for the code on the premises. For each data point, the monetary costs are also calculated based on the information provided in the host configuration file.

In order to allow for distribution of data entities, we have taken advantage of Oracle’s distributed database management system (DDBMS) [11]. This allows for tables remote to a local Oracle DBMS, to be identified and queried for data through the local Oracle DBMS. This is possible by providing a database link (@dblink) between the local and the remote DBMS systems. Once a bidirectional dblink is established, the two databases can execute SQL statements targeting tables from one another. Using the established dblinks we can provide distribution plans from our analyzer system to perform vertical sharding at the level of database tables. It is important to note that the distributed query engine acts on the deployment of a system after an intelligent decision about the placement of tables has been made by our partitioning algorithm.

5.6 Discussion

MANTICORE’s integration into an interactive development environment (IDE) provides a unique opportunity for hybrid deployment analysis to be weaved into the development process of a web application. We envision that through using MANTICORE, developers will be able to constantly assess and revise their decisions regarding a distributed deployment of their target applications and work towards having deployments that better meet cost and performance optimality requirements.
5.7. Summary

We conducted some informal pilot studies of MANTICORE with developers and software architects at the University of British Columbia. Our early analysis revealed that the developers found the design intuitive and useful. They expressed interest in using MANTICORE to assist them with their design and optimization decisions. Following the three goals of flexibility, modelling, and automation in the design of MANTICORE, during our study we asked system architects to play with the tool and modify their cost or constraint models, explain the results from the generated models and articulate the effectiveness of the automated deployment that MANTICORE provides. The received feedback highlighted the fact that modelling could significantly impact and improve how a distributed system is realized. The developers found it beneficial to be able to see how distribution and database roundtrips could affect performance and work towards performance improvements. They also found it effective to have automated tool for final distribution and deployment of the application. One of the developers stated that: "typically system architects suffer a lot when it comes to the final distribution of an application and it is really helpful if I can program my distribution model into the tool and let it take care of the process".

Even though the current user study was informal and limited in scale, the comments were encouraging to continue on building the platform and moving towards a fully-fledged analysis and user study of the platform. This is part of our future work to use MANTICORE during the development lifecycle of a web application and see how it can help developers with making informed decisions about their hybrid cloud deployments.

5.7 Summary

In this chapter we presented our implementation of MANTICORE as a framework that allows for profiling, partitioning, and distribution of a monolithic software system into a hybrid version of it deployable to a hybrid cloud. In describing MANTICORE, we discussed how its implementation was concerned with the design decisions and requirement analysis we had discussed earlier in this thesis. In Section 5.2, we discussed how MANTICORE in combination with jip-osgi allows for profiling and collecting traces of a software system. In Section 5.3, we described how the results of the profiling process are used in order to generate the DDG of the application and how
the $DDG$ is affected by policy and cost models defined and how different models of generating the $DDG$ can be selected prior to generating a $DDG$. Finally in Section 5.4, we showed how the generated $DDG$ can be used during the partitioning phase, how execution and communication costs are taken into consideration, and how the partitioning algorithms are selected. Moreover, we showed how the generated results are presented both in visualized form for software developers and architects to understand as well as machine interpretable XML files that can be used by the distribution engine. Manticore is open source and can be downloaded from github [12].
Chapter 6

Conclusion & Future Work

While there are advantages to deploying Web applications on public cloud infrastructure, many companies wish to retain control over specific resources [38] by keeping them at a private premises. As a result, hybrid cloud computing, has become a popular architectural model where systems are built to take advantage of both public and private infrastructure. This allows hybrid application deployments to meet privacy, confidentiality, elasticity, and scalability requirements all at the same time. However, architecting an efficient distributed system across these locations requires significant effort. An effective partitioning should not only guarantee that privacy constraints and performance objectives are met, but also should deliver on one of the primary reasons for using the public cloud, a cheaper deployment.

For multi-tier web applications, the challenge of hybrid cloud deployments manifests itself primarily in the partitioning of application- and database-tiers. While there is existing research that focuses on either application-tier or data-tier partitioning, they usually focus on a single tier and at a higher level of abstraction (e.g., the application servers). In this dissertation we showed that optimized deployment of web applications benefits from partitioning at a finer level of granularity (i.e., software functions and database tables), with both tiers being considered simultaneously during the partitioning process. We presented our research on a new cross-tier partitioning approach to help developers make effective trade-offs between performance and cost in a hybrid cloud deployment. We showed that for our benchmark applications (i.e., RUBiS [4], DayTrader [5], and JForum [6]), our approach can result in up to 54% reduction in monetary costs compared to a premises only deployment and 56% improvement in execution time compared to a naïve partitioning where application-tier is deployed in the cloud and data-tier is on private infrastructure.
6.1 Primary Research Contributions

The work presented in this dissertation has made the following four primary contributions:

6.1.1 Context-Sensitive Dependency Modeling

Existing research on application partitioning provides techniques to determine the optimal mapping of software functions to network hosts (e.g. client or server) [43, 63, 78]. Such research only supports a simple one-to-one mapping of functions to hosts. However, in our research we have found this simple mapping to be inadequate because the optimal placement of a function depends on the context in which the function is used. In short, sometimes it is better to execute a particular function in the public cloud and sometimes it is better to execute it on premises. We called such distinction, context-sensitive partitioning, and described our solution to this problem in Chapter 3. We showed that context-sensitive dependency modeling would improve both cost and performance in a hybrid deployment by as much as 40%.

6.1.2 Flexible Cost Modeling

At its core, application partitioning is a method for applying mathematical optimization to distributed software development. The primary objectives for optimization are performance, i.e. request processing latency, and the monetary cost of deployment. However, previous work did not provide a means for developers to explicitly make trade-offs between these two objectives. In our research, we provided a flexible cost modeling technique which allows developers to clearly specify their expected trade-offs between latency and monetary costs when analyzing an application for partitioning and hybrid deployment. Details for our flexible cost modeling were described in Chapter 3 and examples are provided in Appendix C. We introduced the notion of $\gamma$ as a parameter to relate cost of deployment to the level of perceived latency. We demonstrated that a higher $\gamma$ would value lower latency while a lower $\gamma$ would value cheaper deployments. We showed that choosing the right value for $\gamma$ can improve deployment costs by up to 51% if cheaper deployment is the focus. Similarly, in a situation where achieving lower latency is the goal, choosing the right value for $\gamma$ can improve the overall application throughput by almost 25%.
6.1.3 Cross-tier Partitioning

When distributing a system across a hybrid architecture, developers need to simultaneously minimize data transfer, maximize use of public cloud resources, and honor privacy constraints for data. Achieving this goal requires intelligent design for both business logic execution and data placement. Thus it is important to not only analyze the context of code execution, but also to profile relational operations at the level of database queries. We have investigated combining code partitioning and data partitioning to guarantee an optimal hybrid cloud deployment. The work presented here is the first of its kind in tackling the problem of combined code and data partitioning. Our experiments on benchmark applications (i.e., RUBiS [4], DayTrader [5], and JForum [6]) revealed that combined partitioning of code and data entities to a hybrid cloud can provide a cost improvement of more than 55% compared to a naïve hybrid deployment, and around 40% compared to when only code (and not data) is partitioned. The details of our cross-tier partitioning approach are presented in Chapter 4.

6.1.4 Asymmetric Data Exchange Costs

For partitioning and distribution algorithms to be effective within the context of hybrid cloud deployments, we need to consider financial and operational characteristics of the host cloud. Our analysis of cost schemes for various cloud providers revealed that public cloud providers impose asymmetric charges for data to/from their infrastructure [2, 3]. Purposefully enough, these asymmetric data transfer charges encourage pushing data to the cloud by making cost of data transfer to the cloud considerably cheaper than data retrieval from the cloud.\footnote{Amazon and RackSpace have no charges for data transfer into their cloud infrastructure while they charge between $0.12 to $0.18 for every GB of data leaving the cloud.} Exploiting this asymmetric cost model, we proposed an optimization formulation for software partitioning that further reduces the cost of a hybrid deployment. We showed that, by exploiting this asymmetry in data costs, we were able to reduce the monthly costs of hybrid deployments by 11% compared to when this asymmetry in communication costs is ignored. Details for our strategy in dealing with asymmetric cost models are presented in Chapter 3.
6.2 Secondary Research Contributions

Further to the primary contributions discussed above, we also made the following secondary contributions:

6.2.1 Simulation and Real-world Evaluation

To validate the effectiveness of the developed solutions, we conducted thorough analyses of our developed techniques on several open source software systems (i.e., RUBiS [4], Apache DayTrader [5], and JForum [6]). For all the analyses, we first conducted simulations using our implemented framework, Manticore, and then validated the results of our simulations by conducting real-world deployments of the benchmark software systems on Amazon Web Services platform (AWS). Through conducting the evaluations, we were able to demonstrate that the cost and performance were improved for a hybrid deployment of the applications, when the deployment model has been generated using our automated partitioning algorithms. Details for these evaluations can be found in Chapters 3 and 4.

6.2.2 Comprehensive Tooling

All the algorithms and tools described in this thesis are implemented under a framework for application partitioning and distribution to the cloud, MANTICORE [67]. MANTICORE helps software architects make informed decisions with cost-, data-, and performance-effective deployments of their applications. MANTICORE has been implemented on top of the Eclipse integrated development environment, and can be used within the same development environment where web applications are developed. This close integration with the development environment is intended to facilitate the overall process of software development and allow for easier deployment of web applications to the hybrid cloud. Details on the usage of MANTICORE were presented in Chapter 5.

6.3 Thesis Review and Observations

As discussed in Chapter 1, we started our research following the thesis below:

We can develop semi-automatic partitioning and distribution algorithms and tools to facilitate migration of OLTP-style web applications to a hybrid cloud deployment. This is achieved through identification of
6.4 Challenges and Future Work

Based on the contributions in this work, we demonstrated that:

- developing semi-automatic partitioning algorithms is feasible through our cross-tier partitioning algorithm;
- cost and data placement can be captured through our flexible cost modeling scheme;
- code and data dependencies can be analyzed using our context-sensitive dependency model;
- dependency models can be formulated into a BIP optimization problem; and finally
- cross-tier partitioning is proven feasible through extensive tooling and detailed evaluation of benchmark applications.

Following the results of our research, we feel confident to claim that this research work validated our hypothesis.

In the next section we go over some of the challenges we faced throughout our research work, and discuss plans for our future work.

6.4 Challenges and Future Work

Similar to any other research work, the hypotheses discussed and validated in this dissertation had its own shortcomings and challenges. In this section, we discuss some of these challenges and shortcomings and propose how they can be improved in the future.

6.4.1 Semi-automatic Partitioning

While our approach simplifies manual partitioning for hybrid cloud partitioning, it requires some input from a developer. First, we require a representative workload for profiling. Second, a developer may need to provide input about the impact that atomic transactions have on partitioning. After partitioning, a developer may also want to consider changes to
the implementation to handle some transactions in an alternative fashion, e.g., providing forward compensation \[53\]. To ameliorate some of this burden, our tool provides an interactive visualization of partitioning results, as shown in Figure \[4.1\], to allow developers to work through different “what-if” scenarios. Also as noted, our current implementation and experience requires some manual intervention and is limited to Java-based web applications and SQL-based databases. However, as part of the future work, we plan to increase the level of automation and extend the tool to work with web applications utilizing other programming languages and database systems as well.

Furthermore, as described in Chapter \[3\], we make assumptions about the statelessness feature of web applications. Given that there are possibilities for functions in web applications to keep internal state, we plan to extend our research to do static analysis and symbolic execution combined with dynamic profiling and data dependency analysis to detect shared state in application functions and allow for replication or restriction of affected functions in such a way that the overall behaviour of the target application stays intact.

### 6.4.2 Heterogeneous vs. Homogeneous Machines

Following the points in Section \[3.7\] of Chapter \[3\] in our current implementation of the algorithms, our costs models and host configuration information assume homogeneity of all machines in the private premises and the public cloud. However previous research shows that the cloud environment is mostly a heterogeneous environment with machines of the same category having different capabilities. As part of our future work we plan to revise the cost schemes and host configuration information in such a way that a multi-way partitioning of the dependency models would allow for deployment across machines with heterogeneous capabilities.

### 6.4.3 Multi-way and Online Partitioning

As shown in this dissertation, our formulation of an asymmetric cost model works well for a two-way partitioning of a given application. However, as discussed in Section \[3.7\] we are not aware of any efficient algorithm that would allow for multi-way partitioning of dependency graphs with asymmetric cost models. This remains an open problem for us to investigate. Furthermore, we plan to develop tools that allow for dynamic repartitioning of code and data to happen through online monitoring of application
6.4. Challenges and Future Work

particularly for the variations of query plans, currently we are limited to the set of potentially incomplete suggestions provided by the database engine. As described in Chapter 4, our data model is built by consulting the ORACLE database engine. As a result, our data dependency model is only as good as the statistical knowledge, collected by the database engine, of predictions on database query executions. This clearly could result in a suboptimal partitioning of data entities due to suboptimal prediction of query plan executions.

As part of the future work, we plan to develop techniques to allow for more exhaustive analysis of query plans by either i) deploying all distribution models for data entities to identify the optimal query plans, or ii) integrating our optimization techniques more closely with those of database engines to more effectively report on distributed query plans. We also intend to investigate how providing more adaptivity in relocating data and code entities can result in better performing and more cost effective deployments of applications.

6.4.4 Data Dependency Analysis

As discussed in Section 4.7, our current implementation of data-tier partitioning relies on leveraging the distributed query engine from a production database (i.e. ORACLE). In some environments, relying on a homogeneous integration of data by the underlying platform may not be realistic. A given monolithic OLTP-style web application may use ORACLE as its database engine, however when it comes to a distributed deployment in a hybrid cloud, conditions may enforce different choices of database for different parts of the distribution. As an example, a hybrid cloud offering of a MySQL database may happen to be cheaper than its ORACLE counterpart or a cloud provider may lack proper licensing to support an ORACLE database. In situations like this, it is important to allow for heterogeneous engines to manage data for different parts of a hybrid deployment, e.g., ORACLE for the premises deployment and MySQL for the cloud deployment. We are currently working to automatically generate REST interfaces to integrate data between the public cloud and private premises rather than relying on a SQL layer. In future work we plan to support a more loosely coupled service-oriented architecture to accommodate a distributed version of the partitioned applications.
6.4.5 Security and Privacy Requirements

Following the discussions in Section 4.7, our current work mostly focuses on security and privacy of data storage when dealing with hybrid cloud deployments. This implies that, for our deployments we are mostly concerned about where the data is stored rather than how the data flows. Consequently, our analysis and partitioning algorithms deal with constraints on data placement and not the flow of data. While this allows for addressing scenarios concerning regulations on data storage, it falls short in dealing with more security and privacy critical scenarios where the flow of data is also important. Our current approach does not guarantee that leakage of privacy sensitive data from the private infrastructure to the public cloud will be prevented. For the future work, we plan to develop techniques that would allow for analyzing the data flow and tracking privacy critical data. We intend to combine this work on data flow analysis into our approach to provide a better guarantee on protecting highly sensitive user information.

6.5 Summary

To summarize, we have shown that, in the presence of code and data placement constraints, the use of automated profiling, analysis, and partitioning techniques would lead to optimal deployment of web applications to the hybrid cloud. We showed how this can be achieved through a combination of context-sensitive dependency modeling, flexible cost model, cross-tier partitioning, and asymmetric data exchange models. We also conducted detailed analysis of benchmark applications to validate the overall improvements in cost and performance when doing a cross-tier partitioning of web applications for hybrid cloud.

Overall, our findings showed significant improvements in hybrid deployment of web applications to the cloud using the proposed automated approach. Nonetheless, the research is only scratching the surface when it comes to (semi-)automated application deployment to the cloud. Web applications, while one of the major candidates, are not the only type of applications for hybrid cloud deployment. With NoSQL databases, map-reduce type of applications, scientific CPU intensive web applications, batch processing applications, etc., the problem of (semi-)automatic partitioning and deployment of applications to the cloud, remains an open problem. We believe the correct strategy for a hybrid deployment really depends on the context in which the application is utilized. As such, any other type of ap-
6.5. Summary

Application deployed to the hybrid cloud, requires detailed research and evaluations for the identification of a proper deployment strategy to the hybrid cloud.
Bibliography


[27] Personal Correspondence with Microsoft Principal Architecture, Doug Franklin.

[28] Personal Correspondence with RackSpace Engineer, David Campos.


Bibliography


[75] Microsoft. The Economics of the Cloud. USA, November 2010.


[104] F. Yang, N. Gupta, N. Gerner, X. Qi, A. Demers, J. Gehrke, and J. Shan-


Appendix A

Appendix on Hybrid Cloud Solutions

As discussed in Chapter 1, three infrastructure elements contribute to a hybrid cloud solution: i) the public cloud infrastructure, ii) the private cloud infrastructure, and iii) the network infrastructure connecting the two. Public cloud infrastructure involves resources and services offered by cloud provider companies, in the form of computation and storage resources as well as accompanying services enabling resource usage elasticity and pay-per-use cost models. Similar to public cloud infrastructure, private infrastructure also involves computation and storage resources with the distinction that these resources are fixed in size and are privately managed.

Despite existence of various points of variability across different hybrid cloud solutions, in this appendix, we distinguish them based on their type of network connectivity; mainly because the network connection can connect any two types of public and private infrastructures in a hybrid cloud solution. Options for network connectivity range from software-enabled virtual private networks (VPNs), to more reliable hardware-enabled VPNs, to highly complex low latency and high-capacity direct network connections between public cloud and private data centers which vary in their offered resources and pricing models. Next we review each hybrid cloud solution as well as the respective provider companies.

Software-enabled VPNs. Hybrid clouds using software-enabled VPNs utilize the existing network infrastructure of the Internet and do not require any extra hardware to enable secure connectivity between the public cloud and private data centers. This reduces costs in the overall setup of these hybrid cloud solutions. However, establishing such hybrid connectivity is complex and may suffer unstable data transfer rates. Vyatta [13] and CloudReach [14] are examples of companies offering such hybrid cloud deployments connecting Amazon public cloud to arbitrary private infrastructure. Google also connects its Compute Engine [24] and
Appendix A. Appendix on Hybrid Cloud Solutions

AppEngine \[23\] to arbitrary private data centers using a service called Secure Data Connector (SDC) \[15\]. In both these examples, capabilities and cost of utilizing public cloud resources follow the cost models for Amazon AWS and Google Compute Engine / AppEngine respectively. On the private cloud however, it is the responsibility of companies utilizing these services to establish, configure, and manage their private resources. The model is particularly suitable for software systems that do not require significant amount of data exchange between the public cloud and the private infrastructure and can tolerate occasional connectivity interruptions.

Hardware-enabled VPNs. Hybrid clouds with hardware-enabled VPNs provide faster and more secure connectivity at a higher cost. Costs of using a hardware-enabled VPN usually fall above regular network appliances of a cloud provider. However, their security and reliability offerings as well as configurable network setups make them attractive options for hybrid deployments. Amazon Web Services (AWS) offers such hardware-enabled VPNs via their Virtual Private Cloud (VPC) \[16\] service to be combined with arbitrary private infrastructure. VPC is utilized by companies such as Kiip \[1\] and Karma \[17\] to build hybrid connectivity with increased security and confidentiality \[18\] \[19\]. Hardware-enabled VPNs can replace software-enabled VPNs in systems where more reliability and security is expected. However, in terms of connectivity and bandwidth, such systems also rely on the underlying network infrastructure which could limit data exchange rates and cause occasional network disruptions.

Direct Data Links. Finally, hybrid cloud with direct network connections between a private data center and a cloud data center offers the most consistent network performance of all hybrid deployments. On the upside, clients of such services are offered low latency and high capacity for data exchange, making such deployments suitable for traffic heavy cloud applications. On the down side however, there are only limited options for private infrastructure that would allow for such type of connectivity to cloud data centers. Furthermore, these setups require complex network hardware and result in deployment costs often three times as much as a software-enabled VPN connectivity. As an example, Amazon Direct Connect \[20\] with its more reliable high-capacity network topology provides the ideal infrastructure for companies requiring high throughput and data-intensive interactions between their private infrastructure and the public cloud. NetApp \[21\] and Zadara Storage \[22\], offering cloud-enabled content-delivery networks and data management, are among the customers
for Amazon Direct Connect. From the set of customers to a direct data link solution, it can be seen that this model of deployment is particularly effective in systems where extensive data exchange over a reliable network is expected between the cloud and the private infrastructure. The offered reliability and network capacity justifies the higher costs of deployment over a direct data link solution.

With all the existing solutions, choosing the right hybrid cloud solution is a matter of identifying requirements for performance and reliability versus cost and complexity of the deployments. This is the job of the system architect to decide about the right balance for all these factors when choosing which hybrid cloud solution to adopt.
Appendix B

Appendix on Dynamic Dependency Graph
<frame bn="jdbcwrapper" cn="Util$TraceItem"
  mn="nz.jdbc.Util$TraceItem:print"
c="1" t="663702" dfp="23444" dtp="123324" cfp="1" ctp="1"/>
<frame bn="jdbcwrapper" cn="Util"
c="1" t="12712" dfp="53" dtp="56" cfp="1" ctp="1"/>
<frame bn="jdbcwrapper" cn="RollingPrintWriter"
  mn="nz.jdbc.RollingPrintWriter:println"
c="1" t="547975" dfp="0" dtp="0" cfp="1" ctp="1"/>
<frame bn="jdbcwrapper" cn="RollingPrintWriter"
  mn="nz.jdbc.RollingPrintWriter:print"
c="1" t="124318" dfp="0" dtp="0" cfp="1" ctp="1"/>
<frame bn="jdbcwrapper" cn="RollingPrintWriter"
  mn="nz.jdbc.RollingPrintWriter:write"
c="1" t="90654" dfp="0" dtp="0" cfp="1" ctp="1"/>
<frame bn="jdbcwrapper" cn="RollingPrintWriter"
  mn="nz.jdbc.RollingPrintWriter:write"
c="1" t="57759" dfp="0" dtp="0" cfp="1" ctp="1"/>
<frame bn="jdbcwrapper" cn="RollingPrintWriter:ensureOpen"
c="1" t="12452" dfp="0" dtp="0" cfp="1" ctp="1"/>
<frame bn="jdbcwrapper" cn="RollingPrintWriter"
  mn="nz.jdbc.RollingPrintWriter:println"
c="1" t="352978" dfp="0" dtp="0" cfp="1" ctp="1"/>
<frame bn="jdbcwrapper" cn="RollingPrintWriter"
  mn="nz.jdbc.RollingPrintWriter:newLine"
c="1" t="194368" dfp="0" dtp="0" cfp="1" ctp="1"/>
<frame bn="jdbcwrapper" cn="RollingPrintWriter:ensureOpen"
c="1" t="11105" dfp="0" dtp="0" cfp="1" ctp="1"/>
Appendix C

Appendix on Cost and Policy Specifications for Partitioning

C.1 Policy Specification

As shown in Program C.1, the policy spec can include constraints on what code entities to include in the dependency graph. For example, the policy spec of Program C.1 constrains the DDG to contain only the information for the TradeServletAction:doQuotes business logic functionality of the DayTrader application. The policy spec can also define entities that can be ignored in the dependency model. This can involve code blocks that belong to the underlying platform on which the application is executed. For the example of Program C.1, the policy model excludes entities of the catalina component in the Tomcat JEE server from the DDG of the application and as such the created DDG will not contain any code entities from this component as part of its model.

The policy spec can also specify which entities can possibly be replicated or not replicated when creating the DDG. For non-replicated code entities in the DDG, the DDG ensures that there is only one single instance of each of these code blocks in the dependency graph. When the application is partitioned, these code blocks are constrained to be placed either on premises or in the cloud. On the contrary, for replicable code or data entities, the entity can co-exist both on the private infrastructure and in the cloud. For stateless code entities this code replication can easily be supported by simply having replicated instances of the code on public and private infrastructure. For stateful code entities or data entities, synchronizing the state of code instances or supporting consistency across different versions of data entities needs to be handled by the system under analysis. Given that the system can guarantee the consistency across
replicated instances, we can use the DDG to effectively decide about the optimal partitioning and distribution of the application across public and private infrastructures.

Program C.1 A sample policy specification filtering on given DDG; ignoring any code from the catalina jar in the dependency model; and making all the database tables non-replicable. The constraints are defined as regular expressions.

```
{
  "constraints": {
    "root": {
      "entry": {
        "id": "1",
        "entity": {
          "component": "^.*\(TradeServletAction:doQuotes\).*$",
          "target": "component" }
      }
    },
    "ignore": {
      "entity": {
        "component": "^\.(catalina).*$",
        "target": "component" }
    },
    "non-replicable": {
      "entity": {
        "component": "^\.(DBBundle:Oracle).*$",
        "target": "component" }
    }
  }
}
```

C.2 Cost Schema

As shown in Program C.2 the cost scheme can encode information on the CPU and memory capabilities of the machines in the cloud or on-premises as well as their associated costs for a given amount of time. All machine
C.2. Cost Schema

capabilities and associated monetary charges are configurable in the cost scheme. Similarly for the data exchange, the cost scheme can encode information on the bandwidth of the connection link between the cloud and the private infrastructure for both the inbound and the outbound communications. Separate monetary costs can also be configured for the inbound and outbound communications.
C.2. Cost Schema

Program C.2 An example cost scheme provided to the profiler with information on host capabilities and data exchange rates as well as resource usage costs.

```json
{
    "capabilities": {
        "hosts": {
            "host-num": "2",
            "host": {
                "id": "1", "default": "true",
                "cpu": {
                    "capability": { "scale": "GHz", "#text": "2.4" },
                    "cost": { "unit": "3600", "scale": "second", "#text": "0.50" }
                },
                "memory": {
                    "capability": { "scale": "GB", "#text": "3" },
                    "cost": { "unit": "1", "scale": "GB", "#text": "0" }
                }
            }
        },
        "exchange-rates": {
            "exchange-rate": {
                "from-host": "1", "to-host": "2",
                "capability": { "scale": "MB", "#text": "100" },
                "latency": { "scale": "second", "#text": "0.03" },
                "cost": { "unit": "1", "scale": "GB", "#text": "0.12" }
            }
        }
    }
}
```

// data for host2 is omitted

// data exchange for host2->host1 is omitted
