Fast Optimization for Scalable Application Deployments in Large Service Centers

By

Jim (Zhanwen) Li

A thesis submitted in partial fulfillment of the
requirements for the degree of

Doctor of Philosophy

Ottawa-Carleton Institute of Electrical Engineering
Faculty of Engineering
Department of Systems and Computer Engineering
Carleton University
Ottawa, Ontario, Canada, K1S 5B6

April 29, 2011

The undersigned recommend to
the Faculty of Graduate and Postdoctoral Affairs
acceptance of this thesis

Fast Optimization for Scalable Application Deployments in Large Service Centers

Submitted by

Jim (Zhanwen) Li. B.Eng.,M.Eng.
in partial fulfillment of the requirements for
the degree of Doctor of Philosophy in Electrical and Computer Engineering

_____

Chair, Howard Schwartz, Department of Systems and Computer Engineering

_____

Thesis Supervisor, Murray Woodside

_____

Thesis Co-Supervisor, John Chinneck

_____

External Examiner, Daniel A. Menasce, George Mason University

Abstract

Large complex service centers such as clouds must provide many services to many users with different service contracts. To deploy applications with many tasks across a cloud infrastructure, many goals must be satisfied simultaneously, which poses a large and complex optimization problem. The goals include quality-of-service targets, power minimization, respecting limited memory and software licences, and ability to track changing demands.

This thesis creates new approaches to task assignment in service centers that are QoS driven and fast, scalable and extensible to new objectives and classes of restrictions. The new approaches provide sound configurations across a large number of highly coupled decisions. These approaches can effectively handle online resource management in various difficult dynamic environments, increasing the stability of management.

The new approaches include (i) the combination of linear optimization with an analytical performance model to solve a complex nonlinear optimization problem via a series of LP solutions, (ii) a mixed-integer linear programming (MIP) model to address many interacting integer constraints, (iii) a new heuristic to approximately solve large-scale MIPs with greater efficiency, and (iv) strategies to increase robustness in the face of dynamic changes in demand.

ACKNOWLEDGMENTS

First and foremost, my greatest appreciation goes to my supervisors, Prof. Murray Woodside and Prof John Chinneck, for lending their extensive experiences and amazing broad vision as well as great deal of energy to this thesis work.

I owe a great deal of thanks to Prof Greg Franks, the principle designer of LQN. He offered many supports to improve the performance of LQNs for this research. I am also indebted to Prof Marin Litoiu for his valuable comments on the research projects. Other thanks are due to Prof Daniel Menasce for his thorough review on the thesis.

Many thanks to the friends from IBM and labs in school, including Tao Zhen , Yan Rui, Nan Jian, Michael Xiao, Thomas Reidemeister, Xiuping Wu, Pengfei Wu, Jacky Liu and Jin Xu et al, providing countless hours of discussion. I have had the pleasure of working with these talented people. Many thanks to my friends, including Jason Zhu, Jie Xiao, et al, who provided a constant source of entertainment and a nice accommodation in Ottawa.

Irene (Aiying) Chen, my beloved, helped me in more ways than I can possibly recount here. I am eternally grateful for her love and support that kept me going through the many ups and downs. I greatly appreciate my parents for their enduring patience and kind understanding throughout the course of my studies.

Moreover, I would like to thank Dr. Shiping Chen, my Master`s supervisor in CSIRO, for introducing me to the research on software engineering.

The financial support for the research was provided by IBM Toronto, through the Center for Advanced Studied.

Table of Contents

Abstract iii

Table of Contents vi

List of Tables x

List of Figures xii

Notation xiv

Chapter 1 Introduction 1

1.1 Motivation ... 1
1.2 Control Approaches ... 3
1.3 Contributions ... 4
1.4 Thesis Organization ... 5

Chapter 2 Background 7

2.1 Large-Scale Service Systems ... 7
2.1.1 Service System Metamodel ... 7
2.1.2 Cloud Computing ... 8
2.2 QoS in Service Systems ... 9
2.2.1 Performance Metrics and Relationships ... 10
2.3 Evaluation by Performance Model ... 11
2.3.1 Markov Modelling ... 11
2.3.2 Petri Nets ... 12
2.3.3 Queue-Based Performance Model ... 12
2.3.4 Layered Queuing Networks ... 14
2.4 Monitoring-Based Autonomic Computing ... 19

2.4.1 MAPE-K Architecture 19
2.4.2 Dynamic Management Approach 20
2.5 Optimization Approaches 23
2.5.1 Bin Packing 23
2.5.2 Hill Climbing 24
2.5.3 Reinforcement Learning 25
2.5.4 Flow Network Based Optimization 26
2.5.5 Mixed Integer Programming 26

Chapter 3 State of the Art in Optimal Deployment 28

3.1 Related Work 28
3.2 Weaknesses of Existing Work 31

Chapter 4 Problem Statement and Solution Overview 36

4.1 Problem Statement 36
4.2 Overview of the Solution 38

Chapter 5 Optimization Algorithms to Address Static Deployment (without Integer Constraints) 41

5.1 Optimization Architecture 41
5.1.1 Deployment Management in a Cloud 43
5.2 Overview of the Optimization Approach 44
5.3 Step 1: Network Flow Model for the Service System 47
5.4 Step 2: Solve the Optimization Model 54
5.5 Step 3 and Step 4: Insert Deployments and Solve the LQN 55
5.6 Step 5: Test for Convergence 55
5.7 Step 6: Linearized End-point Step (LEndStep) 57
5.7.1 Sensitivity Analysis of the Performance Model 57
5.7.2 LP Model to Seek Minor Adjustment 58
5.8 Case Study: Meet Response Time Goals at Low Execution Cost 60

Chapter 6 Discrete Optimization Algorithms for Static Deployment 65

6.1 Optimization Model with Integer Constraints 65
6.1.1 Mixed Integer Programming (MIP) Model 65
6.2 Solving the MIP 70
6.2.1 Algorithm: Heuristic Packing (HP) 71
6.2.2 Algorithm: Heuristic MIP (HMIP) 74
6.2.3 Evaluation of HP and HMIP 75
6.3 Revenue Model Supported by the Optimization 81
6.4 Case Study: Deployment with Multiple Goals with Contentions 82
6.4.1 Comparison with Other Approaches 85

Chapter 7 Management in Dynamic Environments 87

7.1 Re-Optimization with Persistence 89
7.1.1 Constraints on the Flow Rates 89
7.1.2 Reward/Penalize the Allocations 90
7.1.3 Constrain New Replicas for Some Specific Tasks 91
7.1.4 Limit New Replicas to a Small Number 93
7.1.5 Reduce the Number of New Hosts 94
7.2 Three Re-Optimization Strategies in Dynamic Environments 94
7.3 Experiments: Controlling the Scale of Changes 96
7.3.1 Case Study on Controlling the Costs of New Replicas for Specific Tasks 97
7.3.2 Case Study of Limiting New Replicas to a Small Value 99
7.4 Summary of Optimization with Persistence 101

Chapter 8 Case Study 102

8.1 Experimental Environment 102
8.2 Evaluate the Scalability of the Three Algorithms 103
8.3 Evaluation of the Stability of Management in Dynamic Environments 110
8.3.1 Dynamic Case I: Varying Workloads 112
8.3.2 Dynamic Case II: Host Failures and Repairs 119
8.3.3 Case III: Management for Applications Addition/Removal 125
8.3.4 Summary of the Effectiveness of the Algorithms in Dynamic Environments 132

Chapter 9 Conclusions **134**

9.1 Achievements 134

9.2 Limitations and Assumptions 136

9.3 Future Work 137

Reference **139**

List of Tables

Table 3.1 Overview of the Existing Solutions on Deployment Management 35
Table 5.1 Corresponding Entities in Different Views 49
Table 5.2. Host Resource Attributes in the Example 62
Table 5.3 Response Times of Classes in Each Iteration 63
Table 5.4 Host Multiplicity at each Iteration, and Final Utilizations 63
Table 5.5 Execution Cost and Number of Iterations Corresponding to Relaxation of the Goals 64
Table 6.1 Variables and Parameters used in the MIP 67
Table 6.2 Variables used in HP and HMIP 71
Table 6.3 CPLEX Configurations 75
Table 6.4 Host Information 76
Table 6.5 Pure MIP, Heuristic and HMIP Comparison (Without Contentions) 77
Table 6.6 Response Time and the Associated Cost in Each Iteration 84
Table 6.7 the Use of License of Each Task (HMIP) 84
Table 6.8 Comparison of the HMIP and Simple Packing 86
Table 7.1 Running Applications in the Cloud Infrastructure 97
Table 7.2 Re-Optimization with Limitations on the New Replicas per Task 98
Table 7.3 Re-Optimization with Limitations on the Percentage of Changes 100
Table 8.1 Evaluation of Pure MIP, HP and HMIP with Contention 105
Table 8.2 Average Response Time per Step (Varying Workloads) 113
Table 8.3 Average Cost per Step (Varying Workloads) 114
Table 8.4 Average Active Hosts per Step (Varying Workloads) 115
Table 8.5 Average Percentage of New Hosts per Step (Varying Workloads) 116
Table 8.6 Average Number of Replicas per Step (Varying Workloads) 117
Table 8.7 Average Percentage of New Replicas per Step (Varying Workloads) 118

Table 8.8 Average Costs per Step (Host Failures and Repairs)..... 121
Table 8.9 Average Number of Replicas per Step (Host Failures and Repairs) 122
Table 8.10 Average Percentage of New Replicas per Step (Host Failures and Repairs) 123
Table 8.11 Average Number of Hosts per Step (Host Failures and Repairs)..... 124
Table 8.12 Average Percentage of New Hosts per Step (Host Failures and Repairs).... 125
Table 8.13 Average Costs per Step (Application Addition/removal)..... 126
Table 8.14 Average Number of Active Hosts per Step (Application Addition/removal) 128
Table 8.15 Average Percentage of New Hosts per Step (Application Addition/removal) 128
Table 8.16 Average Number of Replicas per Step (Application Addition/removal) 130
Table 8.17 Average Percentage of New Replicas per Step (Application Addition/removal) 131

List of Figures

Figure 1.1 Application Deployment in a Cloud 2
Figure 2.1 Service System Metamodel 8
Figure 2.2 A Lab-Scale System with the Trade 6 Benchmark 16
Figure 2.3 An Example of a Layered Queuing Network Model of a Service System 18
Figure 2.4 MAPE-K Architecture, from [44] 19
Figure 2.5 Feedback control for QoS and optimization, from [86] 20
Figure 5.1 Sketch of model-based optimization architecture 42
Figure 5.2 Application Processes in a Cloud 44
Figure 5.3 Network Flow Model 48
Figure 5.4 An example of a web application 52
Figure 5.5 LQN Model of a Service Center 61
Figure 5.6 Fragment of Network Flow Model 62
Figure 5.7 Near Optimal Deployment for the Service Center in Figure 5.5 64
Figure 6.1 Linear Approximation of the Power Consumption against the Utilization of CPU 66
Figure 6.2 the LQN Model of an Application to be Deployed with Multiple Goals 83
Figure 6.3 the Use of CPU and Memory in Hosts (HMIP) 85
Figure 7.1 Deployment Scenarios 88
Figure 7.2 the number of new replicas of each task created in re-optimization (constraint on new replicas per task) 99
Figure 7.3 the number of new replicas of each task created in re-optimization (Ptg Control) 101
Figure 8.1 Response Time of the Application with Varying Workloads 113
Figure 8.2 Total Energy and License Cost Subject to Varying Workloads 114
Figure 8.3 Total Number of Hosts subject to Varying Workloads 115

Figure 8.4 the Percentage of New Hosts in Use subject to Varying Workloads 116
Figure 8.5 the Total Number of Replicas in Use subject to Varying Workloads 117
Figure 8.6 the Percentage of New Replicas subject to Varying Workloads 118
Figure 8.7 the Variation of the Size of the Host Pool in Host Failures and Repairs Environments 119
Figure 8.8 the Total Energy and License Cost Required by Each Approaches subject to Host Failures and Repairs 120
Figure 8.9 the Total Number of Replicas in the Host Failures and Repairs Environment 122
Figure 8.10 the Percentage of New Replicas in the Host Failures and Repairs Environments 123
Figure 8.11 Active Hosts in the Host Failures and Repairs Environment 124
Figure 8.12 Active Hosts in the Host Failures and Repairs Environment 124
Figure 8.13 the Total Energy and License Cost subject to Application Addition/removal 126
Figure 8.14 the Total Number of Hosts in Use subject to Application Addition/removal 128
Figure 8.15 the Percentage of New Hosts subject to Application Addition/removal 128
Figure 8.16 the Total Number of Replicas subject to App Addition/removal 130
Figure 8.17 the Percentage of New Replicas subject to the App Addition/removal 131

Notation

acronym	
SLA	Service Level Aggreement
MIP	Mixed-Integer Linear Programming
LP	Linear Programming
HP	Heuristic Packing
HMIP	Heuristic MIP
QoS	Quality of Service
NFM	Network Flow Model
LQN	Layered Queuing Network
Arc *arc* used in the NFM	
$Arc_{src\text{-}dest}$.	the *arc* can send flows from source node *src* to destination node *dest*
l_{arc}	the lower flow bound of the *arc*.
u_{arc}	the upper flow bound of the *arc*.
C_{arc}	the Cost per unit of the flow using *arc*.
p_{arc}	the reward/penalty on the *arc*.
α_{ht}	variable, the operation demand rate from host *h* assigned to task *t*
β_{ts}	variable, the operation demand rate from task *t* assigned to service *s*
γ_{sc}	variable, the operation demand rate in service *s* assigned to user class *c*
User Class *c*	
f_c	request rate of UserClass c, requests/sec
$\boldsymbol{S}_c$	the set of services used by class *c*
Y_{cs}	meanRequests by UserClass *c* to Service *s*, during one response to a user request. This includes the effect of indirect calls to other services at rate meanCalls
Z_c	the average time the user in class *c* spends between receiving one response and making the next request (sometimes called a think time)
N_c	Population of users in class *c*, called a *closed* workload situation
RT_c	mean response time of class *c*

$RT_{c,SLA}$	target response time of class c written in the service level agreement (SLA)
$f_{c,SLA}$	the minimum required value of $f_{c.}$ written in the service level agreement (SLA)
Host h (a host is a processor or other device)	
Ω_h	the capacity limit of host h, given as a processing rate relative to a "standard" host used to calibrate execution demands of services. The saturation Capacity of host h is given in the same units, and the utilization to provide a margin
M_h	Memory Capacity of host h, memory available for application tasks
Ω^+_h	remaining execution demand capacity of host h
Ω^*_h	The used capacity of host h, $\Omega^*_h = \Omega_h - \Omega^+_h$
M^+_h	remaining memory space of host h
M^*_h	The used memory space of host h, $M^*_h = M_h - M^+_h$
φ_h	expected upper utilization of host h. $\varphi_h = \Omega_h$/saturationCapacity of h
θ_h	expected lower utilization bound of host h . assigned demand /saturation capacity$\geq \theta_h$
C_h	Execution cost of host h, a cost factor for a unit of execution on this host. In a homogeneous cloud these are all equal.
C_{fh}	Fixed Cost of host h.
C_{ht}	The cost of the host h shared by the task t
S_h	The selection of host h, binary
A_{ht}	Binary variable, to indicate if the task t is allocated on the host h
R_h	Start-up cost of the host h
Service s (a service is an operation carried out by a task (process))	
d_s	mean hostDemand of Service s, per invocation, in CPU-sec. d_s can be obtained by measurement on the reference processor type
$d_{s,SLA}$	the processor demands for service s (regardless of the impacts due to contentions) ,$d_{s,SLA} = \Sigma_c f_{User,c,SLA} (Y_{cs} d_s)$
Task t (a Task has one or more Services)	
$d_{t,SLA}$	the sum of the processor demands for task t $d_{t,SLA} = \Sigma_c f_{User,c,SLA} \Sigma_{S \in S_{User,c} \cap S_{Task,t}} (Y_{cs} d_s)$
d_{ht}	the processing demands of host h are used by task t.
m_t	Memory Requirement of task t, in order to execute (assumed the same for all nodes in this research)
L_t	maxLicenses available for task t (these are the concurrent licenses)

d^+_t	remaining execution demand of task t
L^+_t	Remaining available licenses for use
L^*_t	Total number of license in use, $L^*_t = \Sigma_h A_{ht}$
L_t'	Extra number of licesne in use, integer
S_t	the set of services provided by task t
C_{Lt}	pay-per-use cost of every extra license for task t, beyond L_t
Parameters used in LQN	
D_e	processing demand of the entry e
Parameters used in optimization framework	
e^i_c	at iteration i, let the throughput of user class c be $f^i_{c,LQN}$, and define the shortfall in throughput to be e^i_c: $e^i_c = f_{c,SLA} - f^i_{c,LQN}$
e^i_{sc}	at iteration i, define the shortfall in service demands to be e^i_{sc}: $e^i_{sc} = \Sigma_c d_s e^i_c$
δ^i_s	at iteration i , for service s. Over all users, the flow adjustment to service s is $\delta^i_s = \Sigma_c e^i_{sc} = \Sigma_c d_s e^i_c$.
Δ^i_s	The total surrogate flows (virtual demand) of serivce s at iteration i , $\Delta^i_s = \Sigma^{i-1}_{j=1} \delta^j_s$
δ^i_t	at iteration i , over all users, the flow adjustment to task t is $\delta^i_t = \Sigma_{S \in S(t)} \delta^i_s$, $S(t)$ are the services at task t.
Δ^i_t	The total surrogate flows (virtual demand) of task t at iteration i , $\Sigma_{S \in S(t)} \Delta^i_s$, $S(t)$ are the services at task t.
C_{HP}	The total energy and license cost retuned by the heuristic packing
$BigC$	A very large constant
Parameters used in Sensitivity with LP	
E	an entry in the application template, instantiated in task replicas as replica entries denoted as e
ϕ	A vector to store the throughputs of entries
$\delta\phi$	A vector to store the the change of throughputs
$\overline{\phi}$	A vector to store the baseline throughputs returend from the iteration loop
J	Senstivity matrix, showing the change on outputs due to the change of parameters
δy	a vector, each element is a varaible to represent the change of the request rates
$\overline{y}$	A vector of the baseline of the request rates

σ_h	A fraction to control the size of the remaining capacity that can be used. $\sigma_h \leq 1$.
$Perf_i$	The value of the *ith* performacne
$Mpar_j$	The performacne model parameter *j*
Parameters used in Sensitivity with LP	
C_R_t	Cost of each new replica for task *t*
S_R_t	The number of replicas above the limit of task *t*
ptg_T	percentage of running tasks that could be migrated
Slack_T	the extra number of migrations

Chapter 1 Introduction

1.1 Motivation

New enterprise service centers need to deliver efficient, flexible and scalable services that respond rapidly to shifting business requirements. The ideal service infrastructure can help businesses make sound decisions in real time and can create opportunities for businesses to respond instantaneously to evolving demands. These new enterprise service centers will give the advantage of flexible deployment as needs change, hide management details from the user and the service provider, and require payment only for resources used [43] [40].

In a new enterprise service center, there are many applications comprised of many interacting tasks. Each task has one or several replicas deployed in the service center or the Cloud infrastructure, which consists of many heterogeneous machines, as shown in Figure 1.1.

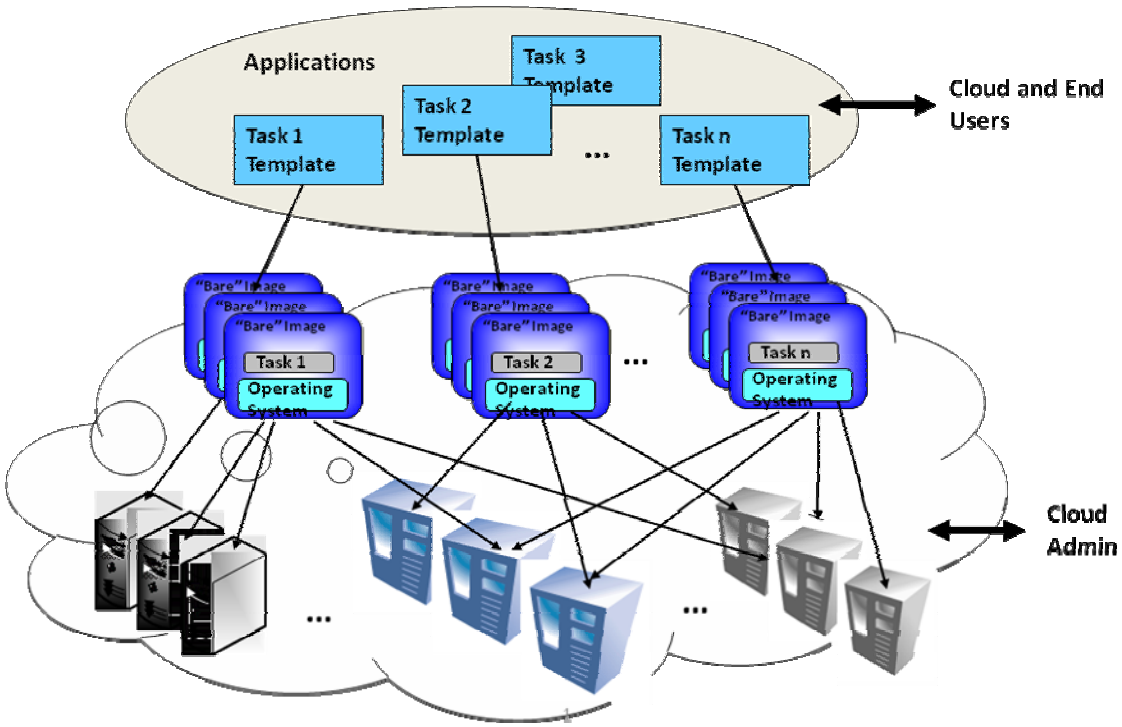


Figure 1.1 Application Deployment in a Cloud

A deployment method must scale up to thousands of services running on thousands of hosts, and must be cheap enough to re-run frequently as loads and requirements change. An integrated management viewpoint of deployment should consider numerous constraints, objectives and relationships between many interacting elements with higher-order nonlinear terms, which include the performance and quality of service (QoS) goals of each application, resource use (CPU cycles and memory space) and available capacity, license availability, software and hardware queuing delays, economic and latency issues, robustness in dynamic conditions, number of machines in use and energy consumption etc. Moreover, the ideal management system must be able to account for numerous constraints in a virtual environment and keep the models and decisions up to date in response to the changing workloads and computing resources, so as to minimize the risks of transitions and ensure system performance during certain critical periods.

Problem Statement: The goal of this research is to develop an efficient heuristic to optimize deployment problems for these large service centers. The new approach must be able to:

1. Satisfy many simultaneous goals
2. Handle may interacting constraints, such as minimum contracted quality of service, many resource and economic constraints *etc*.
3. Scale well to support many classes of users, many interacting tasks and many heterogeneous hosts on the Cloud infrastructure
4. Provide sound reconfiguration in response to change of loads and requirements
5. Extend to other objectives and constraints.

1.2 Control Approaches

Analytical performance models and measurement based reconfigurations are two ways to deliver the required performance, such as QoS, for service systems [62]. Performance models give a way to estimate the value of performance metrics, which are often used for capacity planning purposes. For example, they can help find bottlenecks [29] or compare competing alternatives [57]. By including tracking methods to update performance models at runtime, an analytical model can be extended for use in dynamic management [70][71]. Monitoring-based reconfigurations are dynamic control approaches for real-time management in response to the changing workloads and environment. In general a dynamic controller uses monitors to collect system information and then applies control policies to adjust the system configurations to achieve the desired outputs [36] [37]. Control approaches capable of supporting performance management in general can be classified into a few categories, as shown below:

1. *Threshold Control:* As some important performance value reaches a certain threshold, controllers execute operations according to the policies [18][95][104].
2. *Control Theories:* Based on a system control model, controllers attempt to achieve the desired effect on the outputs by manipulating the inputs to the system. For example, in [1][2], the system adjusts the acceptance rate to ensure multi-class QoS for web servers; in [25][26][27][28][35][37], control theory is used to manage the performance of various web servers and databases by regulating the thread pool size or workload distributions. In [59][102], control theory is used to control large-scale event-driven service systems.
3. *Optimizations:* An optimizer looks for the best (or near-optimal) solution among a set of available alternatives in an optimization domain. Optimization includes a variety of approaches such as heuristic packing [109][88][16][13][21], hill-climbing [99][113], machine learning [80][97], genetic algorithms [67][79], nonlinear optimization [24], mixed integer programming [15] [66] [98] etc.

With the emergence of new service systems, new requirements bring new challenges to existing solution methods. Rapid optimization of large-scale application deployments is one of the new control properties required by the evolution of service systems [1][43].

1.3 Contributions

This thesis develops an effective optimization approach that is able to rapidly solve large-scale deployment problems for service centers. The main contributions include:

1. A heuristic approach that combines linear programming (or mixed-integer programming) and nonlinear performance analytical model to optimize nonlinear deployment problems. The approach is fast, scalable and extensible

to new objectives and can satisfy a diversity of simultaneous goals, which may be nonlinearly coupled.

2. An algorithm that improves the scalability of the mixed-integer programming (MIPs) for task assignment in service centers. In comparison with a pure mixed-integer program, the new algorithm can handle larger number of configuration options and obtain a good quality solution with much greater efficiency.
3. The application of optimization with persistence for dynamic task assignment. The new approach accounts for the risks and costs associated with changes and helps high-quality solutions to persist, and performs very well in a variety of dynamic environments, including varying workloads, the addition and removal of applications, and the failure and repair of host machines. The approach balances the risks of impacting running tasks against the increase of the share of resources subject to various requirements.

1.4 Thesis Organization

This thesis is organized as follows. Chapter 2 introduces the background of performance management for service systems, including cloud computing, MAPE, performance models and various optimization approaches. Chapter 3 reviews related works on deployment optimization. Chapter 4 provides the problem statement and gives an overview of the solution. Chapter 5 proposes an optimization algorithm to address nonlinear contentions with a heuristic loop. Chapter 6 presents a MIP model to consider integer constraints and new algorithms to address large scale MIPs with greater efficiency. Chapter 7 proposes several approaches to provide optimization with

persistence for advanced deployment management in response to various dynamic changes. Chapter 8 evaluates the effectiveness and computational cost of the algorithms.

Chapter 2 Background

2.1 Large-Scale Service Systems

Large complex service systems, such as enterprise data centers [43], need to offer many services to many enterprise users with separate contracts for quality of service. Resource virtualization is a core technology to support resource sharing and consolidation, system management and environment isolations, so that computing environments can more quickly and reliably be dynamically created, expanded, shrunk or moved in response to the requests. Virtualization enhances flexibility and agility. [34]

2.1.1 Service System Metamodel

We view a service system as comprising UserClasses, Services, ServerTasks, Resources and Hosts, related as sketched in [107] and illustrated through a UML class diagram in Figure 2.1. A UserClass is a group of users which requests services from outside the system, and these services request other services inside or outside the system (exploiting the concepts of Service-Oriented Architecture), forming a web of inter-service traffic. Services are implemented by Applications (ServerTasks), which run as system tasks or thread pools, which may have limited capacity. UserClasses have throughput and delay requirements expressed by their SLAs. Resources, such as ServerTasks and Hosts, make up the system and are shared among many running Applications. Hosts have constraints due to limited memory or processing capacity.

ServerTasks can have license constraints. Thus, an Application can run its ServerTasks within commercial Data Base Management Systems or in Application Servers and there are upper limits on how many instances of those can be deployed.

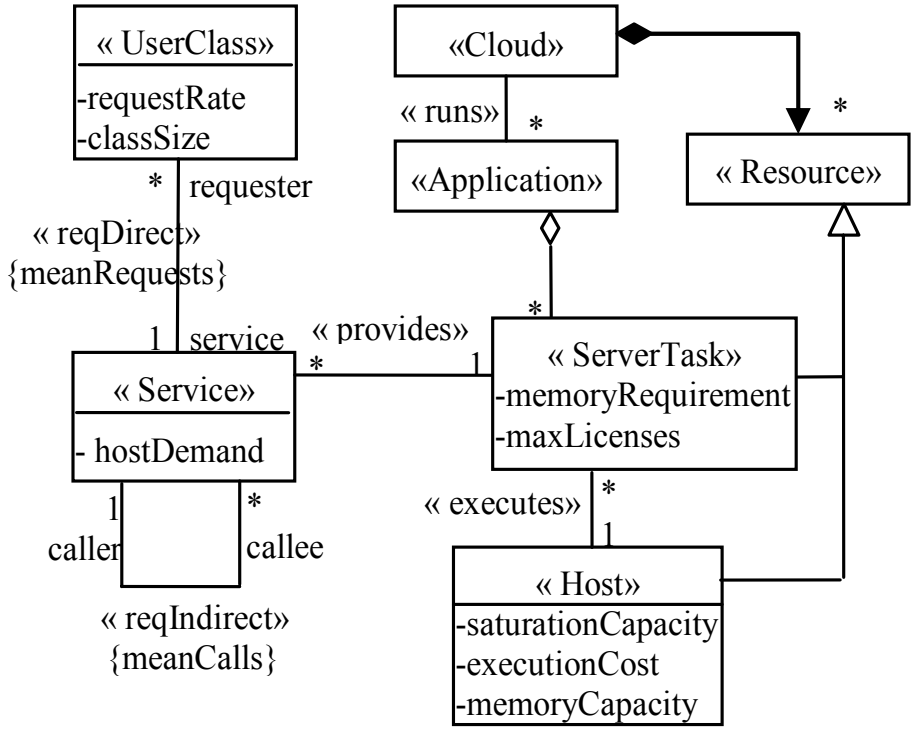


Figure 2.1 Service System Metamodel

2.1.2 Cloud Computing

Cloud computing is a term used to describe a style of computing for next generation service centers where massively scalable service-oriented IT-related capabilities are dynamically delivered to multiple external customers. A cloud may host a variety of services, that include Web applications (i.e. Software as a Service (SAAS) [84]), legacy client-server applications, platforms (i.e. Platform as a Service (PAAS) [39]), infrastructure (i.e. Infrastructure as a Service (IAAS) [40]), and information services [41][43].

A strength of cloud computing is the infrastructure management. Each process (task) has its own virtual machine. Each application in a cloud sees a virtual environment dedicated to itself, such as virtual machines for its deployable processes and virtual disks for its storage. The cloud management allocates real resources to the virtual resources by, for instance, giving a share of a real processor or memory to a virtual machine, or by deploying several virtual machines with replicas of application processes. The application offers services, and also uses services offered by other applications. Cloud computing provides high quality management as a service, allowing users to reduce time and skill requirements on lower-value activities and focus on strategic activities with greater impact on the business [38] [41][40].

Clouds can be classified as public and private. A public cloud is a collection of computers providing services at retail, where users pay for services they use (processing cycles, storage or higher level services), and do not worry about the mechanisms for providing these services. A private cloud, say within a company, may expose more mechanisms and provide more control to its users. Cloud management is responsible for all resources used by all the applications deployed in the cloud, and the opportunity for global resource optimization is a major driver for implementation of clouds [43].

2.2 QoS in Service Systems

A service system needs to deliver ensured quality of service, including service availability, performance, accessibility, integrity, reliability, regulatory and security [45], and should be able to give scalable services to increasing volumes of requests with satisfied latency and throughputs.

2.2.1 Performance Metrics and Relationships

Quality of Service is often stated in terms of response delay or throughput [83]. Response time constraints can be rewritten as equivalent throughput constraints, based on finite user populations, as described below.

A finite population (closed workload) is preferable for the performance calculations because the results are never unstable due to overloaded hosts, which can happen when throughput is fixed (open workload). Suppose UserClass c has a fixed population of N_c users (called a *closed* workload situation), and has throughput f_c and mean response time RT_c, then Little's identity [63] states that

$$f_c = N_c / (RT_c + Z_c) \quad (1)$$

where Z_c is the average time the user spends between receiving one response and making the next request (sometimes called a "think time"). If N_c is specified in the SLA, and a target response time $RT_{c,SLA}$ is given, then the target throughput is given by:

$$f_{c,SLA} = N_c /(RT_{c,SLA}+ Z_c) \quad (2)$$

When Z_c is unknown the worst-case value of 0 can be taken. If both $f_{c,SLA}$ and $RT_{c,SLA}$ are specified, a throughput is computed from the latter using Eq. (2) and the larger throughput is used.

If on the other hand the throughput is assumed fixed (called an *open* workload situation) and the SLA specifies response time, then appropriate values of N_c and Z_c are chosen to approximate the open situation by a closed one, with N_c and Z_c chosen to give the target $f_{c,SLA}$ according (2).

Percentile response time is another important performance metric, which indicates the percentage of requests that can achieve the response with a specified latency. It can be estimated by queuing model analysis with the distribution of response time [32].

2.3 Evaluation by Performance Model

Analytical performance models give efficient evaluation of the performance of a system, and they are widely used for capacity planning [65] and configuration evaluations [12] [10][70][71][95]. Markov models, Petri-Nets and Queuing models are some of the most typical performance models for software systems.

2.3.1 Markov Modelling

A Markov chain is a stochastic process that consists of a number of states and some known rates of moving from one state to another. In a Markov chain, future states depend only on the present state, and are independent of past states. In other words, the description of the present state fully captures all the information that could influence the future evolution of the process.

However, Markov models have three major problems which limit their use. First, some Markov models are incapable of being scaled. The state space may explode for all but the smallest of models, though Markov models that have a regular structure (such as a birth-death Markov chain) may be handled by a closed form solution. Second, large differences in transition rates may cause numerical instability. Finally, Markov models cannot model systems correctly when a system has purely deterministic services times or service time distributions without a rational Laplace transform. Of the three, the state explosion problem is the one that most often limits the use of this technique [50].

2.3.2 Petri Nets

A Petri Net is a graphical and mathematical modeling approach used to describe information processing systems that are characterized as concurrent, asynchronous, distributed, parallel, nondeterministic and/or stochastic [78]. A Petri Net consists of places, transitions, and directed arcs. Each arc connects a place and a transition. Traditional stochastic Petri Nets are unable to describe scheduling strategies with Petri Net elements. Some more recently developed Petri Nets, such as Queueing Petri Nets (QPNs) [49], can address this problem.

One major weakness of Petri Nets is the "complexity problem" – Petri Net models of even modestly-sized systems are too large for analysis. Similar to Markov models, Petri Nets also suffer state space explosion as the system scales up.

2.3.3 Queue-Based Performance Model

Queuing models rest on queuing theories (e.g., Mean Value Analysis) to provide mathematical analysis of many queue-based performance problems. Queue-based performance models have many variants, which include queueing network (QN) models and layered queueing network (LQN) model. A queuing model is a nonlinear function. Queuing models provide efficient calculations of performance measures with state probabilities and can easily be scaled to large systems [31].

Queue-based performance models are widely used to analyze static performance for capacity planning. They can help to find optimal selection of CPU speed, device capacities and file assignments [93]. Lazowska et al. [52] gave a thorough discussion of computer system performance analysis with queueing network models. In [68] [69] Menascé et al. extended the use of queuing models for performance analysis of service

systems. Queuing models are effective in providing analysis of the performance impact of software and hardware resource contentions. This capability helps to find management policies to achieve satisfactory performances [100]. Queuing models can be used to predict performance metrics in nonlinear dynamic systems and help to reconfigure the systems [65]. Optimization techniques can be combined with a queuing model to conduct dynamic management. For instance, the works in [70] and [71] respectively presented a use of queuing models to seek optimal services in distributed systems, and in [57] a performance model is applied to maximize system workloads.

However, the difficulties of determining model parameters (e.g. the CPU demands of particular operations) from real world systems at runtime have discouraged the use of queuing models in dynamic systems. Recent works on parameter estimation proposed some solutions to address this issue. Liu et al. inferred the service demands of queueing models by minimizing the estimation errors of the end-to-end response times [64]. Zhang et al. used regression-based approximation to estimate the service demands of different transactions, then used a queueing network model to predict performance metrics with different transaction mixes [112]. Pacifici et al. [77] leveraged multivariate linear regression to estimate dynamic CPU demands on the basis of the time-varying nature of the request traffic. In [114] Zheng and Woodside et al. showed that a Kalman filter can quickly estimate parameters in real time and indicate that such a filter is capable of tracking the changing parameters. In [106] Woodside bridged a gap between the practice of measurement and the practice of modeling with statistical concepts, and presented a framework for estimating a model by nonlinear regression. This provides a basis for making software performance models available in dynamic systems.

2.3.4 Layered Queuing Networks

Resource contention increases delays. If software resources are ignored, then all waiting occurs at processors and where throughputs are fixed (open workloads) contention may in principle be controlled by limiting processor utilizations to some chosen amount such as 80%. However this does not provide an estimate of delay so one cannot address the SLA for delay using this solution. This is why, in e.g.[70][71], a performance model for the processors is introduced. However, there is increasing evidence that software resources are also important, and for this a more structured performance model is required. A layered queueing network (LQN) is an extension of queuing models [30][82][108] capable of addressing software contention effects [56].

A *Layered Queueing Network* (LQN) model of a service system is a simplified view of its structure, emphasizing its use of resources. This is illustrated by a small system shown in Figure 2.2. The users (UserClasses) are represented in the LQN by *userTasks*, in which userTask *c* has population N_c. A userTask does not receive any requests, but rather cycles forever, waiting for a think time Z_c given as their demand (e.g. [1000 ms]), and then making a set of requests for service shown by directed arcs to the services. The arcs or arrows are labelled with mean counts of requests, per operation of the requester, e.g. (1). Services are represented by *entries*, which have processing demands D_e and make requests to other entries. Where a Service is provided by a ServerTask, the entry forms part of a corresponding resource called a *task*, and is deployed on a *processor*. Tasks and processors have a multiplicity {*m*} (e.g. {50}, modeling multiple threads or a multiprocessor). As discussed in [30], other software resources such as buffer pools may also be modeled as *tasks*.

A LQN incorporates services with nested requests for other services. Three types of communications are supported by LQN: synchronous call, asynchronous call, and forwarding call.

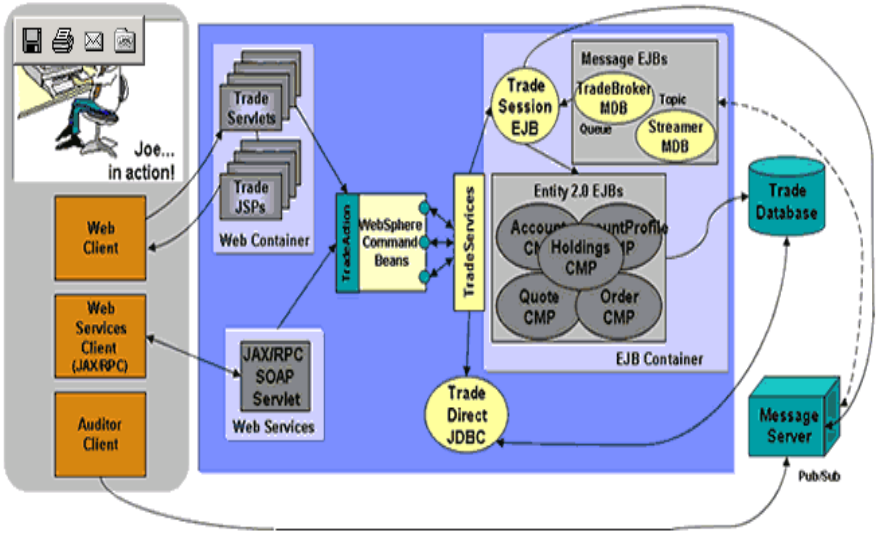
Synchronous call: This is a pattern of Remote Procedure Call (RPC). Sender blocks when waiting for a reply from receiver.

Asynchronous call: This is a non-blocking pattern. Senders can send new requests without the need to wait for replies of previous requests.

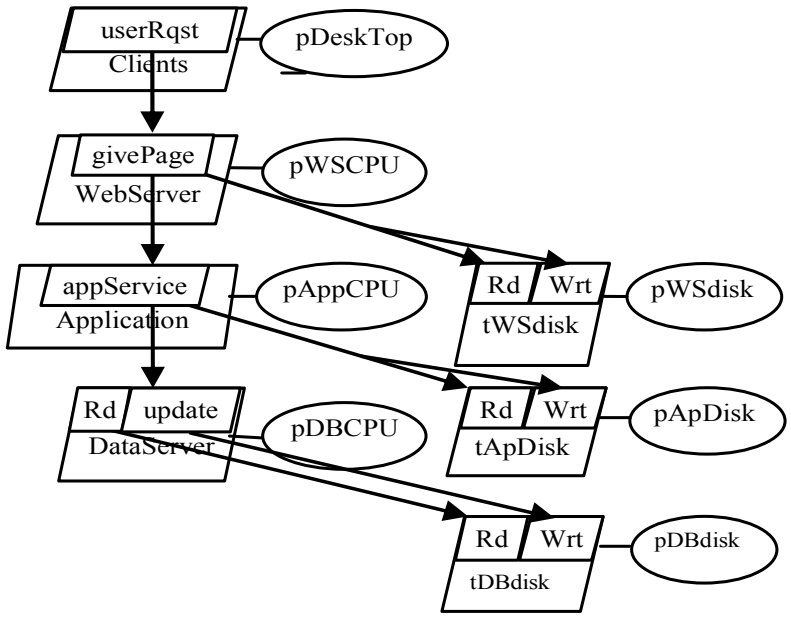
Forwarding call: The receiver can forward a synchronous call to a third task and the third task might reply or forward it further. Receiver does not block after forwarding and the final receiver sends a reply to the blocked sender directly.

These communication patterns can model multiphase types of software service with synchronous (in-rendezvous) and asynchronous (post-rendezvous) phases. They have been successfully applied to many applications.

Based on queuing analysis, an LQN solution determines throughputs at users, entries, tasks and processors, delays including queueing for requests, and resource utilizations. Number the userTasks as $c = 1...C$; entries as $s = 1...S$, tasks as $t = 1...T$, and host processors as $h = 1,...H$. Then throughputs at these entities are f_c, $f_{ENTRY,s}$, $f_{TASK,t}$, and $f_{HOST,h}$ respectively. Task and processor utilizations are $u_{TASK,t}$ and $u_{HOST,h}$ respectively, and for a *multiple* resource, full utilization makes the utilization equal to the multiplicity m.



(a) The service system



(b) The LQN performance model

Figure 2.2 A Lab-Scale System with the Trade 6 Benchmark

The more complex LQN in Figure 2.3 indicates the potential of the LQN framework, with the main features of a shopping service application. The two topmost user tasks represent the two classes of users, with 250 and 100 users. The pUsers processor

represents the user desktops. Arrows represent requests for services (labeled by the mean number of calls, e.g. (2)), with a filled arrowhead indicating a synchronous request (the requester waits for a reply), and an open arrowhead, an asynchronous request. There are databases for inventory and customer information. Entries are named beginning with "e" in Figure 2.3 and carry labels (e.g. [1]) for the mean execution demand D on the host in ms. Processors are shown as ovals, linked to tasks deployed on them; a processor entity may represent a multiprocessor. Processors and tasks are labeled by a resource multiplicity (e.g. {100}). For a user task the multiplicity is the number of users in the class. Pure delays without contention are represented by infinite-multiplicity tasks and processors. Some additional details: a device like a disk is modeled by a task with entries to describe its services, and a processor representing the physical resource. Delay for an external service not modeled in detail can be represented by a surrogate task with a pure delay (infinite task) and entries for its services, as for the Payment Server and Shipping Server.

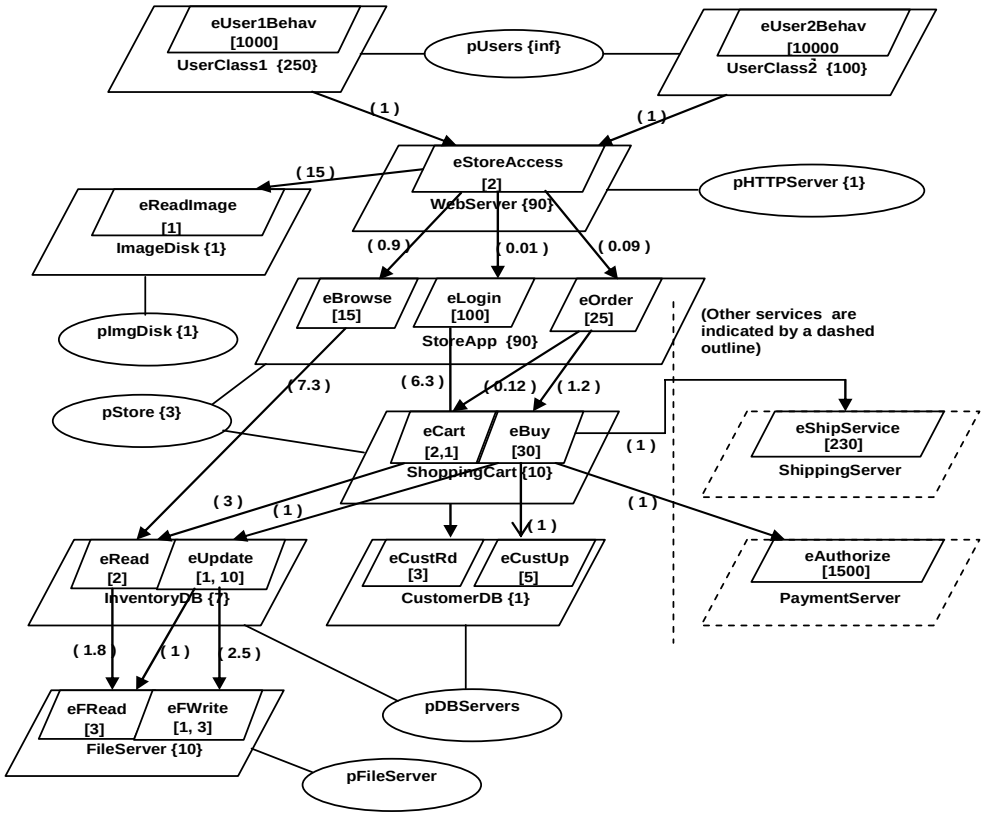


Figure 2.3 An Example of a Layered Queuing Network Model of a Service System

In practice the performance model is derived from ordinary measurements on the running system. The structure of tasks and entries participating in each service is found either from the system design or by tracing some representative requests as described in [114]. The parameters are determined by profiling, by regression techniques [118] or by using a tracking filter [106][117]. In practice these models are not perfect, because of statistical estimation errors, and delays in computing the parameters (during which the system may change). The references above discuss how this inaccuracy may itself be estimated and controlled.

2.4 Monitoring-Based Autonomic Computing

2.4.1 MAPE-K Architecture

Monitor, Analysis, Plan, Execute and Knowledge (MAPE-K) are the five basic parts to consist an autonomic manager for controlling either a system or a component, proposed by IBM with the following architecture [44].

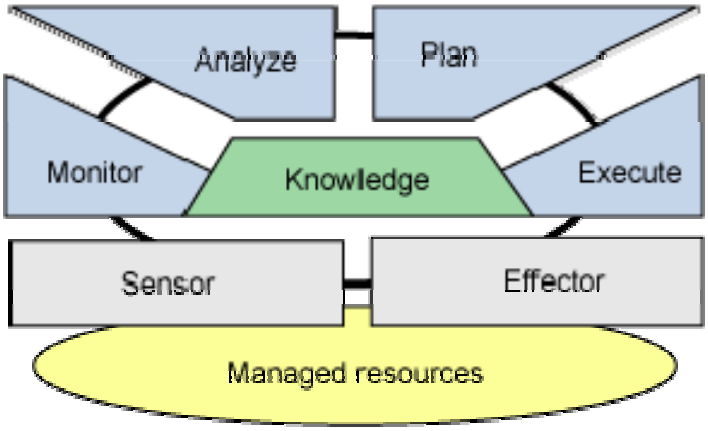


Figure 2.4 MAPE-K Architecture, from [44]

In the control loop, monitor, analysis, plan and execute communicate and work together with one another and exchange appropriate knowledge and data.

- Monitor: collects system information from sensors;
- Analyze: correlates and models complex situations;
- Plan: constructs the actions to achieve the desired targets;
- Execute: performs the necessary changes to the system via the effectors.
- Knowledge: maintains the data shared by the above four parts.

Under the MAPE-K architecture the self-optimization loop for service systems can have the structure shown in Figure 2.5. Performance models are the core knowledge inside the loop in the approach used in this thesis. Monitors collect system information to

construct and update the performance models. Analysis components such as filters are used to estimate the parameters that cannot be observed but needed in modelling. Based on the performance models and the QoS goals, optimization is conducted to seek the best decisions in response to varying conditions. These decisions are finally executed by the system effectors. The optimization feedback loop gives a generic solution to support performance-driven self-management on autonomic systems. This loop enables the developer (or system administrator) to modify the code using intuitive mental models for performance tuning, perhaps guided by principles such as given in [86].

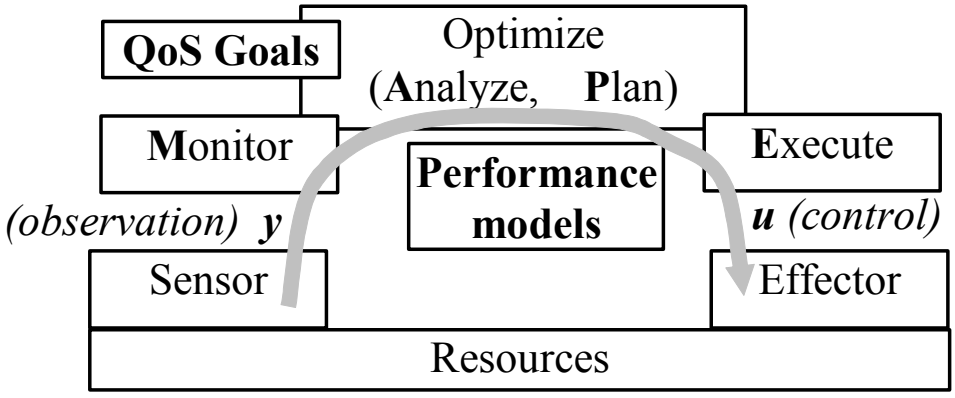


Figure 2.5 Feedback control for QoS and optimization, from [86]

2.4.2 Dynamic Management Approach

Many monitor-based control approaches have been developed for dynamic system management in response to runtime changes. The approaches in general can be grouped into threshold-based control, dynamic control theories and optimization.

- **Threshold Control**

Threshold control makes decisions based on the values of monitored variables, such as response time or utilization rates. When the value reaches a certain threshold, then the

system controller executes policies to change configurations. Threshold control is widely used because of its simplicity. For instance, in multi-tiered systems threshold control can give a straightforward way to adaptively adjust capacity provisioning in terms of response time [95][84] or to improve performance for overloaded systems [19]. Admission control and congestion control are typical examples of the use of threshold management. In admission control, response latency determines the acceptance or rejection of the new arrival requests. Welsh et al [104] demonstrated that admission control can be used to ensure percentile response time on such complicated systems as SEDA.

A threshold controller is driven by control policies. However it is difficult to create good adaptive policies for complex systems under many constraints and uncertainties. Threshold control is limited to controlling one parameter from one measurement variable. In service systems, many activities and configurations interact and many decisions have some alternatives, and threshold control is inadequate.

- **Dynamic Control Theories**

The motivation of control theory is to manipulate the inputs to a system to achieve the desired effects on the outputs. Control theory views a service system as a dynamic system, modelled with *H(s)*, which describes the relationships between the input configurations *X(s)* and the output performance *Y(s)*. With the use of a transfer function shown in Equation 1, the desired performance *Y(s)* can be obtained by regulating *X(s)*. *H(s)* could be a large matrix consisting of many interacting elements with higher-order nonlinear terms. The control approach seeks a configuration *X(s)* that provides the expected output *Y(s)*.

$$Y(s) = H(s)X(s) \quad (3)$$

Linear and nonlinear controllers have been used to adjust configurations in a variety of software systems. For instances, Hellerstein and Diao et al. applied linear control theories such as linear-quadratic regulator (LQR) to control many multiple-input and multiple-output (MIMO) commercial software systems, including workload management on web servers [1][25][27] and memory management in databases [60][89] as well as connection control in communication software [28]. Abdelzaher et. al. applied proportional-integral (PI) control on workloads to obtain the desired response time [2]. Chen et al. deployed control theories to manage the replication of databases in a data center [17]. The current author employed feedback control to deliver quality performance for stage-event-driven systems [58][59]. Xu et al. used predictive control to control dynamic resource allocation in data centers [102].

The effectiveness of control theory relies on the accuracy of system modelling. The complexity of a service system hides many uncertainties, which make it difficult to explicitly describe the relationship between outputs and inputs. Controllers are driven by reference outputs, but the desired values of some outputs are unachievable in some service systems; for instance the maximum capacity of a service center is hard to estimate accurately in advance. A decision maker in a service system must be able to coordinate many decisions with numerous classes of restrictions. These requirements are beyond the capability of control theories. Moreover, the system models could be quite different to address different dynamic problems, limiting the persistence of the management.

- **Optimization**

An optimization method in general includes an optimization model and optimization algorithms. Optimization can automatically seek the best (or near-optimal) solution in terms of diverse constraints and objectives. There are many optimization approaches developed for software system performance, such as bin-packing, hill-climbing, machine learning, genetic algorithms etc., surveyed below.

Almost every commercial enterprise product has optimization mechanisms to give the required system performance. With the emergence of new styles of computing, higher quality optimization approaches are needed to satisfy new requirements. For example, rapid optimization algorithms that can address complex and scalable configuration issues are urgently needed by large enterprise service systems [43]. Dynamic optimization periodically seeks new optimal solutions to adapt to the changes. In each period, a static optimization is solved.

2.5 Optimization Approaches

Following are some of the optimization approaches used in deployment management. Each approach takes a particular view of the service system and this view conditions the approach.

2.5.1 Bin Packing

Bin-packing views the deployment problem as placing a set of tasks, each taking up a certain amount of volume, into a set of hosts, that each has a limited available capacity (space or time) to host tasks. Tasks and hosts are respectively regarded as items and bins in the algorithms. The goal of bin packing is to place these items into the smallest number of bins. An item must be packed into a bin without violating the space constraints.

Though the solution cannot be guaranteed to be optimal, bin packing algorithms are widely adopted in many cases because of efficiency and simplicity.

In the simplest version of bin-packing, a set of items, each having a single dimension of "size" (e.g. length) is given, and the task is to pack all of the items into the smallest number of identical bins, each of which has a limited capacity. This problem is known to be *NP-complete* (e.g. [33]), hence it is usually solved in practice by heuristics which return approximate solutions. Some heuristics provide tight bounds on the optimal solution.

Multi-dimensional bin-packing is an extension of bin-packing to satisfy more packing constraints. In *multi-dimensional bin-packing*, items and bins have more than one dimension, e.g. height, width, and depth, to account for additional requirements. In online *bin-packing* the set of items is not known in advance: items arrive in a stream and must be packed upon arrival. In dynamic bin-packing the items stay in the system for only a certain time period and then disappear. See Coffman et al [22] for a survey of bin-packing methods.

Though deployment problems have some similarities to bin-packing, a significant difference between them is an application deployment problem allows subdividing processing capacity into parts of any size for allocation, which means a deployment would have more allocation choices than standard bin-packing problems and the problem is more complex.

2.5.2 Hill Climbing

Hill climbing views optimization as exploring decisions on a static surface where each response is mapped onto a set of coordinates. The strategy of hill climbing is to

iteratively improve the current state by varying each variable one at a time, comparing its function costs with its neighbours, and moving to the neighbour state with the best function cost value. The process terminates when the function cost value cannot be improved anymore [81]. The quality of hill-climbing can be improved by multi-start if searching for a global optimum [103].

Hill climbing is a simple and popular search algorithm that is used to find a local optimum. However, it cannot guarantee the global optimum unless the surface of the function cost is concave. The algorithm may encounter problems if the surface of the function cost has ridges or a plateau. Expensive cost function evaluation and the state explosion problem are the most common limitations on the application of hill climbing.

2.5.3 Reinforcement Learning

As a sub-area of machine learning, reinforcement algorithms attempt to derive best configurations for the specific system states by a trial-and-error methodology. Basic reinforcement learning contains system states, actions and scalar rewards. At each step, an action is taken to transit a state and assign rewards to update the value function. Best configurations to achieve maximum rewards are obtained when the objective function converges [90]. Reinforcement learning is a "knowledge free" approach, meaning that configuration parameters are mapped onto evaluation functions without the need of system modelling.

A weakness of RL is that the learning phase takes a long time, and may not react to dynamic changes in a timely way. Furthermore, some of the machine learning algorithms (e.g. MDP) may suffer state explosion when handling large-scale optimization.

2.5.4 Flow Network Based Optimization

Optimization with flow networks is a sub category of graph theory. A flow network takes a view of a service system as a network. A classic flow network includes a collection of nodes and directed arcs that connect pairs of nodes. Each arc is labeled with a triple of parameters [*l, u, c*]: the lower flow bound l , the upper flow bound u and the cost per unit of flow c. A flow in an arc must satisfy the lower and upper bound of the capacity. Ordinary nodes are of three types, and are shown as circles in a network diagram. *Source nodes* introduce flow into the network and *sink nodes* remove flow from the network, at rates given by the input and output arcs attached to them, called *phantom arcs*. Ordinary nodes simply balance flow between their input and output arcs (total input = total output) [20]. In addition, a special type of network flow model (NFM) called a *processing network* [20] has at least one *processing node* which has fixed ratios of the flows in its incident arcs. Processing nodes are shown as squares labelled with the fixed proportion of flow at the attachment point of each incident arc.

The strength of the flow network model is it can model many distribution and assignment problems. By optimizing the distribution of flows across the network, optimal configurations can be achieved. Flow networks have many variants, such as circulation and max-flow min-cost problems etc. However the effectiveness of flow networks stands on the basis that the problem can be modelled with flows and there are effective algorithms that can solve the flow model.

2.5.5 Mixed Integer Programming

Mixed Integer Programming (MIP) extends linear programming to account for some variables that must take integer values. A special case of MIP is binary integer

programming, in which the integer variables are binary-valued. Some advanced algorithms, such as branch and bound, can be used to address MIP problems. But MIPs are generally NP-hard, meaning that the solution time is non-polynomial and hence slow to solve a problem having many integer variables. An effective bounding function can significantly improve the efficiency of a MIP solution, but it is a creative work subject to the requirements of specific problems.

Chapter 3 State of the Art in Optimal Deployment

3.1 Related Work

Optimization approaches can seek better deployment decisions to improve system performance. Bin-packing is probably one of the most commonly used approaches to find optimal deployments. It has been widely used to pack execution requirements [21], execution and communications requirements [109], and memory; all of which have been combined in multidimensional bin-packing [16]. More recently, the use of bin-packing is extended to optimize task allocations in virtual environments [13].

Bin-packing motivates many new packing approaches. For example, Karve [48] and Steinder et al. [88] created heuristics to distribute workloads across virtual nodes in a virtual computing environment, and Tang et al. [91] combined max-flow algorithms and heuristics to manage large-scale resource allocations on Websphere XD. However these packing approaches oversimplify the deployment problems in that blocking delays and the availability of task replicas are important to system performance, but cannot be handled by these packing approaches.

Hill climbing has been applied to maximizing system workloads in service systems [61], minimizing replicas to provide satisfied QoS [115], seeking optimal QoS components for distributed systems [71], and finding resource allocations for DBMS management [113]. Expensive evaluation costs and the state explosion issue limit the

application of hill-climbing on medium or large scale systems. In existing applications hill climbing can support only a few tens of variables.

Reinforcement learning (RL) was first applied by Vengerov et al. [97] to seek optimal resource allocation decisions in terms of changing workloads. Rao et al. [80] made RL available to optimize configurations in virtual environments. In [87], Soror et al. successfully used RL to configure database workloads on virtual machines. In current works the RL is successfully used to support systems with a few tens of configuration parameters.

Flow networks were applied as early as 1980 by Bokhari et al. [9] who discussed the use of flow networks to solve a partitioning problem in multicomputer systems. In [91] Tang et al. presented a combination of max-flow algorithms with a heuristic to allocate varying workloads on a large scale commercial system such as WebsphereXD. This approach can handle thousands of application deployments at the same time, but it ignores the impacts of resource contentions on QoS, and thus cannot deliver ensured QoS for multiclass users. Toktay and Uzsoy [92] leverage a flow model to maximize resource utilizations and demonstrate that a heuristic can achieve almost the same result as a mixed integer programming (MIP), but with much more efficiency. However, this approach ignores the minimum execution demands required by each task and the QoS constraints.

Knapsack optimization can be used to address some combinatorial optimization problems. A multiple knapsack problem is a variant used to address general resource allocation problems that account for the allocation of n items into m knapsacks. Bilgin et al. [7] demonstrate that the Knapsack allocation problem can be modelled by a simplified flow network. In [48] Karve et al. present a heuristic approach for optimizing application allocations in terms of a knapsack model. Similarly, Zhang et al. [111] present a

combination of a queuing model with a nonlinear integer optimization to determine the number of hosts required by a multi-tier server network subject to some QoS requirements. However, this approach only considers the number of machines in use but ignores many performance issues such as how to improve the machine utilization and the constraints imposed by memory, and it is incapable of optimizing the allocation/creation of replicas. Since these algorithms cannot accommodate many optimization constraints and variables, they are unable to coordinate a large number of decisions at the same time.

MIP is effective to model the allocation problems with such integer constraints as memory and host activity etc. In [15], MIP is applied to manage VM deployments in a cloud infrastructure. It optimizes VM allocations taking account of computation power, electric power, storage and network bandwidth, but it does not consider QoS constraints, service allocation and workload balance across VMs. In [98] and [66] MIP was used to conduct power optimization, which is applicable to handle fixed power consumption and execution power consumption in terms of freezing deployments.

Nonlinear programming is applicable to address complicated deployment problems with nonlinear terms. It leverages an iterative process to explore the optima with methods like gradient descent. Many researchers used nonlinear optimization to handle complex deployment problems. For example, in [24] the authors proposed a combination of nonlinear optimization and control theory to solve a revenue problem subject to QoS requirements and dynamic changes. This approach optimizes allocations across a set of heterogeneous hosts by solving an optimization model that includes the nonlinear calculation for performance. This approach is effective for small scale problems with a few servers and hosts, but does not scale for large problems with many optimization

options, because search methods for a large-scale nonlinear problem are very time-consuming. In general a nonlinear optimization model is only available for specific problems, but it is not a generic solution extendable to address new objectives and constraints. Modelling a nonlinear optimization model subject to many coupled relationships is complicated.

Utility functions can be used in some autonomic computing systems [47][85][73][75]. Utility functions are the objective functions for optimization, mapping each possible state into a scalar value. A utility function can be constructed in several ways, such as by ranking the objective functions in importance, or using a weighted combination of the objectives. Optimization techniques are applied to seek solutions for the goal of the utility functions [101]. Menasce *et al*. used hill climbing [6][75][76] and beam search [5][72] for resource management in service systems. These approaches apply analytical performance models to evaluate the utility functions for decision making. However a utility function alone cannot guarantee the constraints are met. A returned solution may achieve a good utility value, but violate some constraints. And the quality of management with utility functions is strongly affected by the optimization techniques.

3.2 Weaknesses of Existing Work

The emergence of large-scale service computing results in three main challenges for existing optimization approaches. Increased complexity is the first, which requires the optimization to be able to satisfy many coupled objectives and constraints at the same time, scalability is the second, and the third is the dynamic changes of the requirements. New solutions are needed to meet these new challenges.

Bin-packing cannot address the complexities. A standard bin-packing algorithm seeks placement decisions on the basis of the available host capacity and the size of each item, but the algorithms cannot address packing problems that allow merging or dividing items into arbitrary size, and the algorithms are poor at finding an optimal packing for items whose volumes interact with each other. In a service system there are many resource issues that cannot be described by bin-packing. For instance, processing capacity can be subdivided into parts of arbitrary size for allocation, and this is not a standard bin-packing problem. Another weakness of bin-packing is that it cannot ensure QoS in a straightforward manner. Bin-packing algorithms will terminate at some local optimal point, but this might be far from any global optimum.

Hill climbing requires evaluation of every candidate decision. Expensive evaluation costs and the state explosion problem make hill climbing suitable for small systems only (see Section 2.5.2). At present hill-climbing approaches can support only one or two dozen configuration parameters. Such capability is far from the requirement of large-scale service systems that need to adjust over a thousand variables at the same time.

Machine learning is particularly suitable to long-term running systems. Nonetheless as the system varies significantly, the learning process may be incapable of finding good up-to-date configurations. And the limitation of the number of states means that the algorithms do not scale up. Thousands of runtime configuration parameters are beyond the capability of current reinforcement learning algorithms. (See Section 2.5.3)

Pure flow networks ignore complex performance issues, such as blocking delays due to resource contention, which are in general nonlinear and NP-hard. Oversimplifications make for an optimistic flow network solution, which may be very different from reality.

A flow network can model resource provisioning and consumption using a flow representation; however, there is a gap between resource utilization and QoS. How to bridge this gap has not been addressed in the existing research. To solve scalable flow networks, rapid algorithms are required, but further research is needed to accommodate this requirement.

MIP is another approach used for deployment optimization, which can be applied to handle more practical problems than LP. However, MIP is NP-hard. It does not scale to solve large problems. And MIP cannot account for nonlinearities. This limits the use of MIPs in problems with nonlinear coupled goals. Current research on deployment often only focuses on how to model the problem in MIP, and then use general algorithms via solvers to seek the optimal configurations, but ignore the weakness of MIP solvers, which may limit the efficiency and scalability. Other algorithms could be more effective than the algorithms in MIP solvers to address some specific MIP problems.

Nonlinear optimization is an approach capable of accommodating nonlinear terms in the optimization model. It gives greater flexibility to model the problem for optimization. Nonlinear problems are solved via an iterative process with gradient descent. Because this process could be time-consuming for complex problems with higher-order nonlinear terms, nonlinear optimization is ineffective for allocation problems with many simultaneous highly coupled goals. Moreover, for QoS management contention must be considered, but the complexity of the calculations for contention makes it hard to model for an optimization solution. Therefore, it is difficult to use nonlinear optimization to deliver QoS-ensured performance.

Table 3.1 summarizes the strengths and weaknesses of the above solution approaches for deployment management. It compares the decision quality, scalability, efficiency, and the effectiveness for satisfying nonlinear coupled goals, including multiclass QoS constraints, and the extensibility for new requirements, ease of modeling, as well as the performance in response to dynamic changes.

Table 3.1 Overview of the Existing Solutions on Deployment Management

Approach	Decision Quality	Scalability	Efficiency	QoS Constraints	Coupled goals	Extendibility	Ease of Modeling	Dynamic Performance
Bin Packing	Non Optimal	Yes	Yes	No	Weak	Weak	Easy	Weak
Hill Climbing	Local Optimal	No	No	Probably	Good	Good	Easy	Weak
Control Theories	Good	----	Yes	Yes	Probably	Weak	Hard	Good
Machine Learning	---	----	No	---	----	Weak	---	---
Flow Networks	Global Optimal	Yes	Yes	No	Probably	Good	Depends	Good
MIP	Near Optimal	No	No	No	Good	Good	Depends	----
Nonlinear Optimization	Local Optimal	No	No	Probably	Weak	Weak	Depends	----

- Decision Quality: the accuracy of the solution, measured by the results that are global optimal, near optimal or non optimal
- Scalability: the size of the problem can be handled
- Efficiency: the speed to solve a problem
- QoS Constraints: capability to ensure the quality of service, such as response time, throughputs, including the account of QoS loss due to resource contention
- Coupled Goals: the capability to address multiple interacting goals at the same time
- Extendibility: good extension to accommodate new objectives and constraints.
- Ease of Modeling: the difficulties to build up a control/optimization model
- Dynamic Performance: the stability of management and the quality of optimization during a dynamic process.

Chapter 4 Problem Statement and Solution Overview

4.1 Problem Statement

The goal of this research is to develop algorithms that provide advanced performance management for large service centers, including the ability to satisfy many simultaneous goals. They should optimize numerous coupled configurations at the same time. These new algorithms must:

1. Scale up to provide the minimum contracted quality of service (QoS) for many users subject to many resource and economic constraints, which include:
    - Multiclass performance targets described in service contracts, (e.g. response time, number of users, capacity given as arrival rates),
    - Constrained software and hardware queuing delays,
    - Constraints on the number of replicas of each service,
    - Constraints on the number of hosts in use,
    - Economic targets (e.g. cost budgets, power consumption, profit targets).
2. Operate quickly enough to provide frequent adjustment as loads and requirements change.

- The new algorithms must be capable of making high quality decisions for possibly thousands of simultaneous configurations across a large-scale system within a few minutes.

3. Give high-quality robust decisions in response to shifting circumstances.
    - Stability in various dynamic environments (e.g. varying workloads, the addition and removal of applications, and the failure and repair of host machines),
    - Control the costs/risks associated with changes,
    - Control creation of replicas.
4. Coordinate decisions subject to numerous simultaneous constraints, objectives and relationships between many interacting elements with higher-order nonlinear terms, which include,
    - The selection of hosts,
    - Computing power consolidation,
    - Allocation of service replicas,
    - Workload balancing and distribution.
5. Provide extensibility of the solution.
    - The solution must be extendable to satisfy new objectives and classes of restrictions.

Existing approaches can give some of these control properties, but not all at once. Bin packing cannot address multiple goals at the same time, and optimization quality is not guaranteed. Hill climbing cannot respond to changes rapidly. Modelling a controlled

system for a large-scale service system is challenging, though control theory performs well for dynamic problems. Machine learning requires a great deal of history data, and is weak in addressing new requirements. Flow Network Optimization and MIP are effective for modelling some issues, but cannot address nonlinear problems. Nonlinear optimization is limited by the efficiency of the iterative gradient search.

The new optimization algorithms developed in this research are required to deliver all these properties at the same time. They must be applicable to managing a set of applications to share a cloud infrastructure in an optimal manner, ensuring multiclass users the expected performance at low cost. They should be able to provide high quality solutions that persist, taking into account the risks and costs associated with changes.

4.2 Overview of the Solution

The solution proposed in this thesis provides dynamic management by solving a sequence of static deployment problems as conditions change. It includes an optimization approach for static deployment based on a snapshot of the system requirements and state, and a persistent control mechanism, giving stable/persistent management in a variety of dynamic environments. This solution models the problem with these goals by an optimization model comprised of an objective function (which addresses multiple goals by weights, rewards and penalties on the objectives) and a set of constraints.

For each static deployment problem, the approach leverages an analytical performance model and scalable heuristics to seek near-optimal deployments which deliver the required performance for each class subject to the cost and quality constraints. The optimization process iterates between a sub-optimization problem and a layered

queuing network (LQN), in which the optimization uses a network flow model (NFM) or a mixed integer programming (MIP) problem. Rapid algorithms are developed to solve these optimization models either exactly (in the case of NFM) or approximately (in the case of MIP). The optimization solution returns high quality deployments allocating host reservations to tasks. The performance model (like LQN) predicts the effect of contentions, which reduce the throughputs. The optimization model is then adjusted by introducing surrogate flows at the services, which account for the additional capacity required to overcome contentions, and the sub-optimization problem is re-solved.

This iteration loop provides an effective approach to seeking sound deployment decisions while considering the effect of resource queueing, including logical resources modeled by extended queueing [56], thereby satisfying constraints on multiclass average response times. This was extended to find a minimal change to accommodate one new application, in [55]. In [57] the optimization approach was extended by combining heuristic packing (HP) and linear programming (LP) to account for memory and license constraints. This approach is fast and scales well.

The algorithm is further improved by combining heuristic packing with mixed integer programming (MIP), to increase the quality of optimization and satisfy more goals, including minimizing the energy consumption, number of hosts in use, and maximizing robustness in dynamic conditions and persistence of solutions subject to changing circumstances. In comparison with a pure MIP via the CPLEX tool, the heuristic MIP (HMIP) can solve much larger optimization problems and obtain a good quality solution with greater efficiency. Contention linearization via sensitivity analysis and linear programming is applied to reduce costs due to over-allocated resources. [53]

During a dynamic process, soft and hard constraints can be added on related variables in the optimization model in order to take account of the change of the environment, providing persistent solutions that reduce the risks/costs associated with changes. In addition, a new model capable of accommodating constraints to limit the scale of changes and control new replicas to specific tasks is described. These persistence mechanisms can deliver advanced robust deployment management in response to a variety of dynamic changes, guaranteeing system stability [54].

The solution is evaluated by a set of experiments in static and dynamic environments. The unit of time in all experiments is milliseconds, and the processor speed, memory space and costs considered in the optimization functions are relative to a standard host. These experiments assume the performance model of each application is up-to-date and accurate. Based on an assumption that the time interval between optimization solutions is long enough that the system will actually settle down to steady state, the results returned by the performance model LQN are steady-state and the mean-value LQN model can be used to predict performance on the real systems.

Chapter 5 Optimization Algorithms to Address Static Deployment (without Integer Constraints)

This chapter presents the fundamentals of the static deployment algorithm, which combines network optimization and contention calculations, effectively solving a nonlinear optimization problem by iterative LP solutions. The returned solution allocates host reservations to tasks, and divides request traffic between multiple task replicas, where applicable.

5.1 Optimization Architecture

In a large-scale service system, such as a cloud, the system incorporates several elements in order to provide the feedback control shown in Figure 2.5; the resulting architecture is sketched in Figure 5.1. Monitoring of resources provides utilization information at the level of the physical processor, virtual machine, and other logical resources. Monitoring of user requests gives measures of throughput and response time. The performance model tool stores a model of each application and its deployment, and is connected to estimation tools for updating the model parameters periodically from the monitoring (the Model Tracker). When a new application is loaded, an initial performance model is supplied by the application provider, derived either from the application design (as described in [110]), from other knowledge of the application, or by tracing its behaviour (as described in [115]). Finally some deployment effector tools must

be included to load and initialize VM images on host processors, as indicated by the optimization.

A management platform might include one or several optimizers to satisfy different requirements. Each optimizer has an optimization model to describe its part of the problem and a corresponding solver to seek the optimal solution in terms of the optimization model. In this research, the optimization aims to satisfy performance targets.

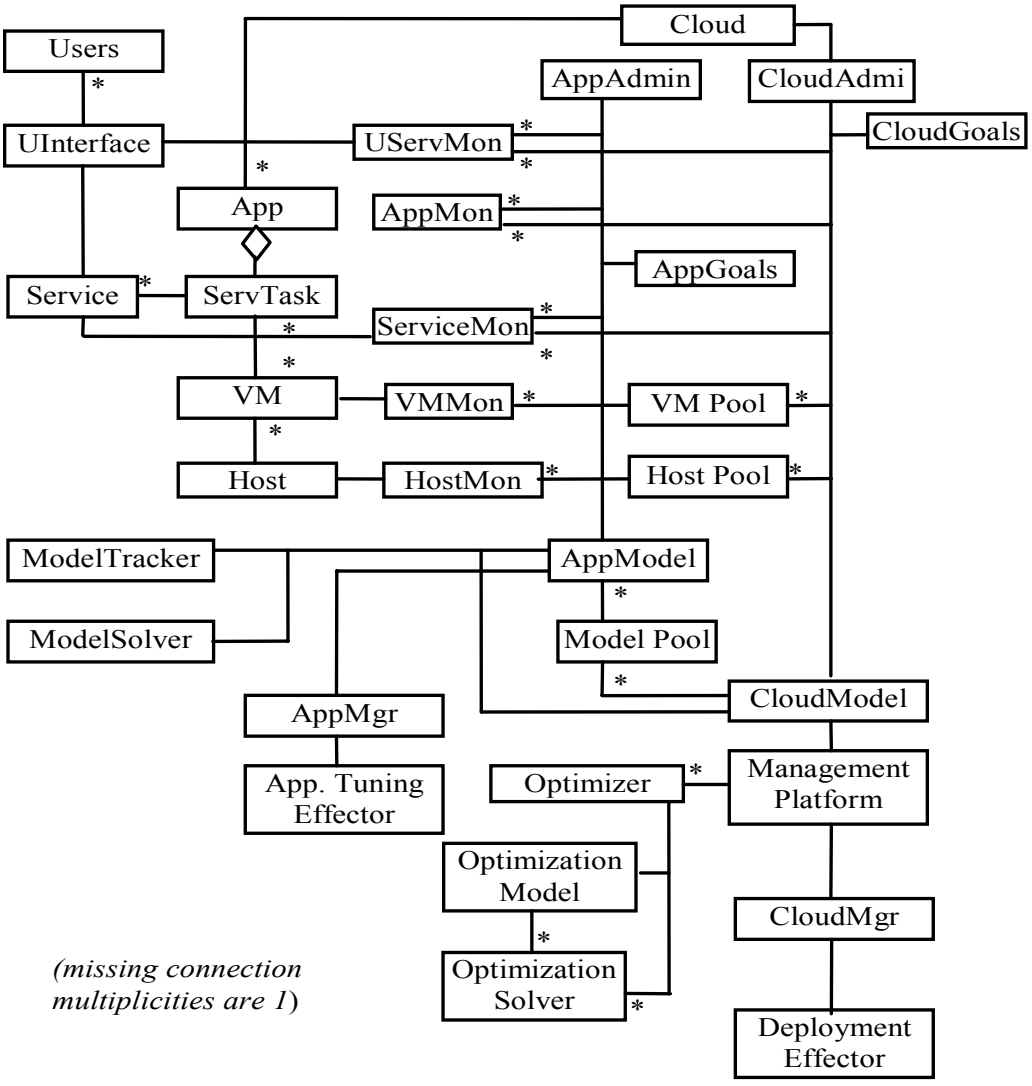


Figure 5.1 Sketch of model-based optimization architecture

5.1.1 Deployment Management in a Cloud

Based on the architecture of Figure 5.1, a scenario of the deployment management is shown in Figure 5.2. When initializing an application deployment, a performance model is constructed from the software specification or from previous operational data. Deployment decisions are created by the optimizer based on predictions by the performance model. Dynamic changes in the system synchronously update the performance models, which then keep the corresponding optimization models up to date. New optimization solutions are sought periodically by solving the optimization models in response to changes.

Virtualization of processors makes it possible for separate applications with separate virtual machines (VMs) to safely share a physical node, and a virtual machine monitor can control the rate of processing provided to each VM. During the runtime deployment is adjusted in terms of the tracked performance model in response to the changes.

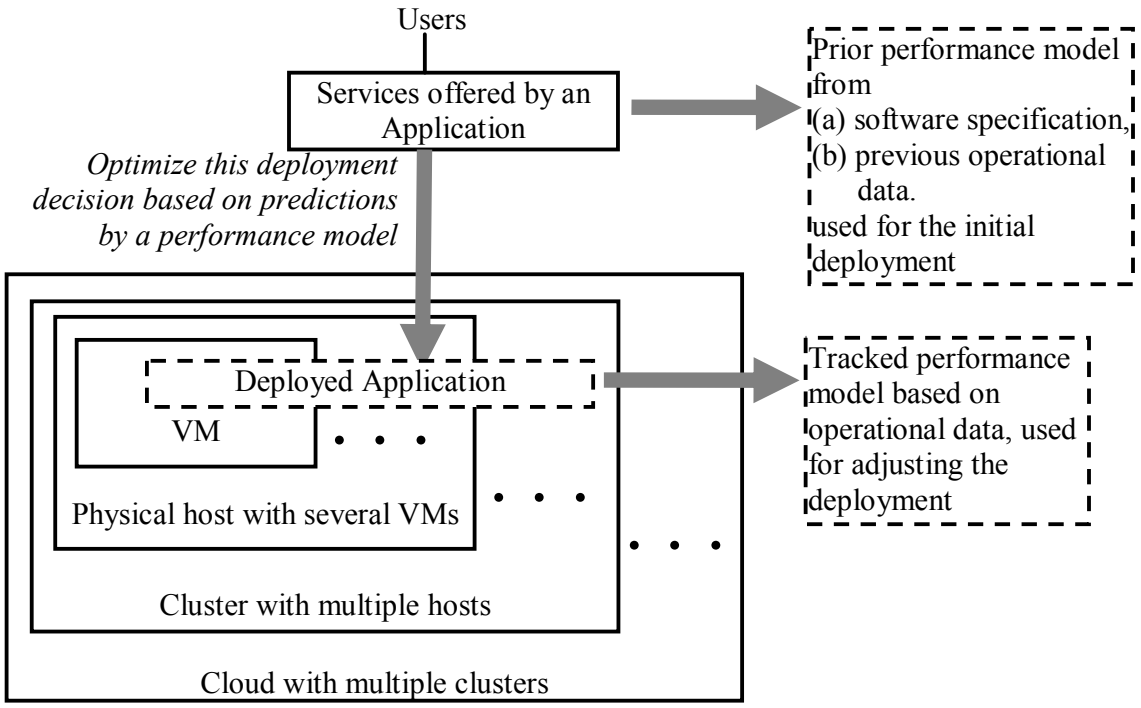
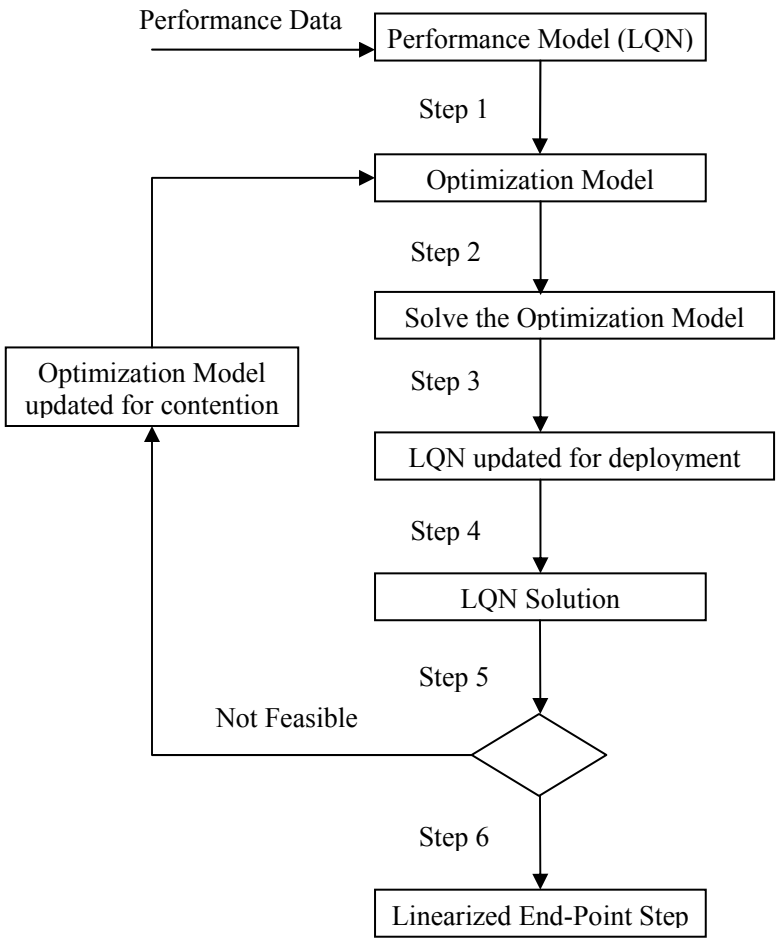


Figure 5.2 Application Processes in a Cloud

5.2 Overview of the Optimization Approach

The optimization algorithm consists of an optimization loop and a Linearized End-point Step (LEndStep). The optimization loop seeks near-optimal feasible solutions. It iterates between a linear (or MIP) sub-optimization problem and a nonlinear cost calculation to find a near optimal solution. Then the LEndStep solves a linearization of the entire problem a single time, improving the quality of the final solution, constrained by the deployment decisions in the feasible solution. The LEndStep tunes the loading of different tasks to minimize cost subject to the QoS constraints. A high-level algorithm describing the execution of the approach is shown as Algorithm I.

Algorithm I. ***Generic High-Level Iterative Algorithm***



In the algorithm, the optimization loop is made up of 5 main steps and iterates until finding a near optimal configuration capable of achieving the required QoS. The steps are:

Step 1. *construct the Optimization Model for the service system,*

Step 2. *solve the Optimization Model to find the suggested deployment decisions.*

Step 3. *reconfigure the Performance Model to incorporate the deployment decisions given by Step 2.*

Step 4. *solve the Performance Model*

Step 5. *test the feasibility of the solution. If not feasible, incorporate the queueing delays into the Optimization Model, and repeat from (2).*

Step 6. *If the solution is feasible, a linearized version of the entire problem is solved to give a final solution (the Linearized End-Point Step (LEndStep) described in Section 5.7). If this final step is ignored the solution may not be as good as it could be.*

In this research two optimization models are developed to satisfy different goals in Step 2. The first is the Network Flow Model (NFM) of the deployment problem. NFM uses linear programming (LP) to optimize the distribution of execution demands, but ignores some nonlinear and integer restrictions. Because LP is solvable in polynomial time, this optimization is scalable and fast. The second is Mixed Integer Programming (MIP), which can accommodate such integer constraints as memory demands and availability, power consumption due to host activity and license availability. The design of the MIP model will be introduced in the next chapter.

The solution given by the optimization model is a deployment,

(1) allocating host reservations to tasks,

(2) dividing request traffic between multiple task replicas, where applicable, and

(3) minimizing cost.

Since the solution of either NFM or MIP by itself ignores the effects of contention for resources, this makes the predicted performance optimistic. These are important practical aspects of the deployment problem, which cannot be addressed by flow optimization via NFM or MIP alone.

Contention introduces additional delays and reduces the actual flows, in a way that can be estimated by a performance model (a LQN in this work). The total capacity required to process an entry is being increased by adding the pseudo-flow; this excess capacity is required to reduce contention (by reducing utilization of the assigned resources). The optimization model then is adjusted by the surrogate flows to describe the new resource requirements, and re-solved. A fixed-point iteration (Steps 2 – 5 in the Generic Iterative Algorithm above) is used to adjust these, terminating at a converged solution (where the delays, including contention, do not violate the QoS constraints).

5.3 Step 1: Network Flow Model for the Service System

A NFM is a graph with arcs which carry flows and nodes which operate on the flows, as illustrated in Figure 5.3 and discussed in Section 2.5.4. Each node shown in Figure 5.3 is representative of a set of nodes, with *H* nodes in the Host column, *T* nodes in the Task column, *S* nodes in the Services column, and *C* nodes in the Class column. The arcs show which flows between nodes may be non-zero, and by the conventions of modeling with the NFM they flow into the hosts, and out from the user classes. Each arc has a flow quantity, defined as

flow quantity = demands for CPU-sec of processing, transferred per sec between nodes

and initially the CPU-sec for any operation will be assumed to be the same on all hosts (hosts are of uniform speed); this is generalized below.

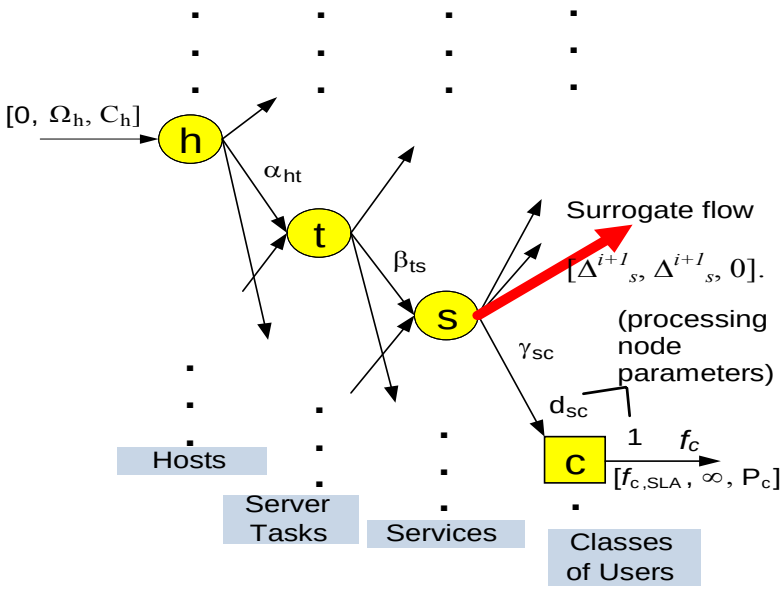


Figure 5.3 Network Flow Model

The deployment problem is formalized with a flow network model (NFM). We consider the flow of execution of services of task t by host h (i.e. α_{ht}), as part of the solution of NFM. The unknown flows α_{ht}, β_{ts}, γ_{sc} comprise the variables in the model. Each arc is labeled with a triple of parameters [l_{arc}, u_{arc}, c_{arc}]: the lower flow bound l_{arc} (default 0), the upper flow bound u_{arc} (default infinity), and the cost per unit of flow c_{arc} (default 0). The parameters are not shown where all take the default values.

An NFM can be derived from the LQN performance model by considering the flow of demands for CPU work implied by the request arcs in the LQN. An NFM host node h =1...H is created for LQN processor h; a task node t=1...T is created for non-user task t; a service node s=1...S is created for entry s. These are *ordinary nodes* which relate demand flow on each host to demand flows by services. The NFM may include additional processors which are not used but which could be used in an optimal deployment. For each *userTask c* there is a *processing node* for user class c in the NFM, which converts a

flow of user requests into CPU demands by services. Table 5.1 summarizes the entities defined for service systems in general, with their corresponding representations in the LQN and NFM models.

Table 5.1 Corresponding Entities in Different Views

Service System	Network Flow Model (NFM)	Layered Queueing Network (LQN)
Processor *h*	Host node *h*	Processor *h*
UserClass *c*	User class node *c*	UserTask *c*
Service *s*	Service node *s*	Entry *s*
ServerTask *t*	Task node *t*	Task *t*
Resource	...	A Task or Processor
Activity	...	Activity (within an entry)

The input arcs on the left in Figure 5.3 represent the total flow $f_{HOST,h}$ at host h, and are labelled with $[0, \Omega_h, C_h]$ meaning that $f_{HOST,h} \geq 0$, the host capacity limit is $f_{HOST,h} \leq \Omega_h$, and the cost is C_h per unit of flow (meaning the cost per execution demand of the host h). For a set of processors of equal speed, and flows given in CPU-sec/sec, the capacities are all 1.0. There is also an arc:

- from host h to each task t which is permitted to be deployed on h, with flow α_{ht} (the demand rate executed on host h, to satisfy the needs of task t). If multiple replicas of a task are deployed, it will have non-zero flows from multiple processors, which will optimally divide the execution flow between them.
- from task t to each service s offered by task t, with flow β_{ts} (the demand rate from the service). In the LQN each service (entry) is associated with just one task.

- from service s to each user class c which causes s to be executed, with flow γ_{ts}. γ_{ts} is the total CPU demand triggered at service s by requests made by class c.

Notice that when a task is replicated on different processors, there is one task node for all the replicas, but one host node for each replica.

These arcs relate demands at processors to demands from user requests, and express the software structure and the constraints on deployment of tasks. Omitted arc labels default to [0, ∞, 0].

The output arcs at the right have a flow which is the requested throughput of the user class. The user class node c is a processing node with flow ratio parameters which convert the class flow f_c at the right in Figure 5.3, in units of user requests/sec, to demand flows γ_{sc} for services. For each single user request by class c, a demand of d_{sc} CPU-sec is required for service s, giving this flow proportionality:

$$\gamma_{sc} = d_{sc} f_c$$

The value of d_{sc} can be determined by profiling the system for each user class request type, or from the LQN model. In the LQN, let Y_{cs} be the total direct and indirect mean requests to entry s for one request from user class c, and let y_{es} be the mean requests made directly from any entry e to entry s. For this purpose user class c will be defined to have an entry numbered $S + c$, and $y_{S+c,s}$ is the mean number of requests made directly to entry s for one user response (in Figure 5.3, there is exactly one request to a particular service entry point, but it can be more general). Then assuming there are no request cycles, Y_{cs} can be computed by setting $Y_{c,S+c} = 1$ for all c, and using:

$$Y_{cs} = \sum_{e=1}^{S+C} Y_{ce} y_{es} \ , \quad s = 1..S$$

Using the parameter D_s from the LQN, for the CPU demand per execution of entry s, we obtain $d_{sc}=Y_{cs}D_s$.

Figure 5.4 shows an example of a service system with two user classes and six services/applications, labeled with the request rates from one service to another. Other labels, such as the size of the user classes and the host demands of the services, are not shown here. In the present deployment analysis, only the total host demand for each application (task) is used. It requires the total direct and indirect requests for a service from each user class, found by following all paths from the user class to the service. For example, from Users1 to DB2Serv it is:

$Y_{user1DB2Serv}$ = 1 x 0.3 x (0.7 + 0.3 x 1.4) = 0.336 requests/user response.

The service architecture in Figure 5.4 can be modeled in detail by a layered queueing model (see, e.g. [30] for more information), which can be calibrated and tracked from operating data [113].

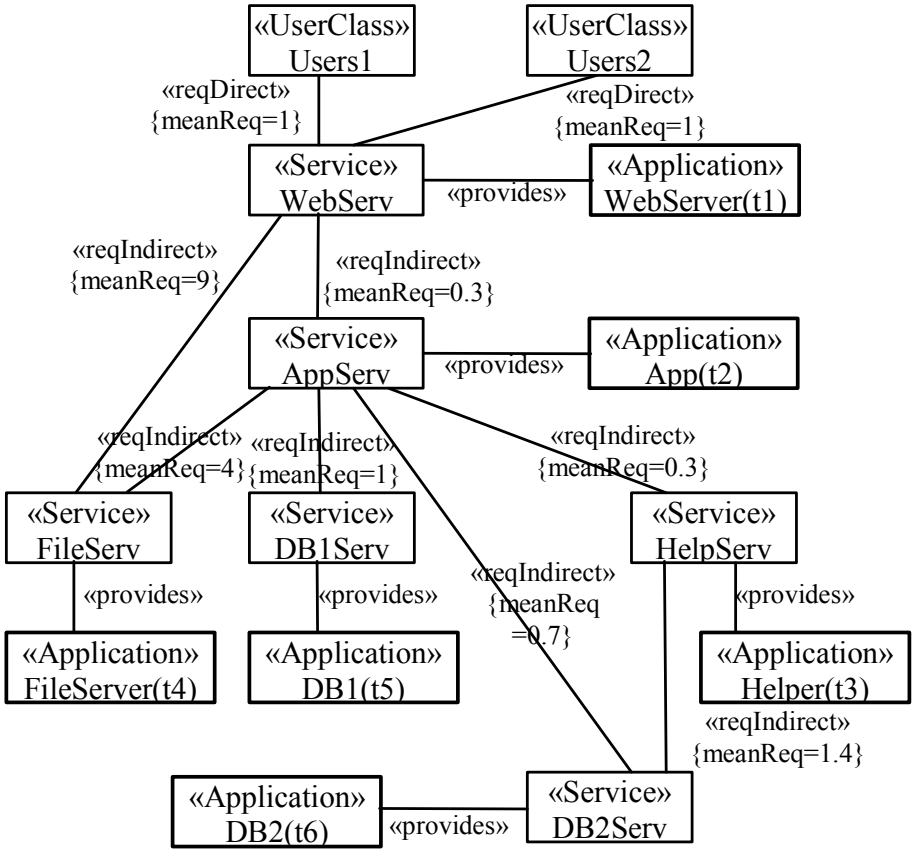


Figure 5.4 An example of a web application

The performance of a service system is affected by contention for resources. Because contention reduces the throughputs and increases the latency, it must be considered in deployment configurations. Limiting processor utilizations to some chosen amount such as 80% [3] is a common way to reserve resources to reduce the effects of contentions. However, this does not evaluate the effects of contentions so it cannot give the ensured performance required by the SLA.

The surrogate flows to offset contentions are indicated by the big arrows attached to the service nodes in the NFM, shown as Figure 5.3. They represent reservations for processing capacity needed to reduce the contention delays. The size of these surrogate

flows (Δ^{i+1}) is estimated using the performance model (LQN) in terms of the deployment configurations given by the NFM-based optimization. Details will be introduced in Section 5.6 (Step 5).

If each task provides different services, the NFM can be simplified by

$$\Sigma_h \alpha_{ht} = d_{t,SLA} + \Delta^{i+1}{}_t$$

where $\Delta^{i+1}{}_t$ is the total surrogate flow of task t at iteration i calculated by adding the virtual demands of $\Delta^{i+1}{}_s$ of the services that are hosted by this task.

The NFM is solved via linear programming. Details of the algorithms are introduced in Section 5.4 below. As pointed out in Section 2.2, for a suitable closed workload population the satisfaction of the user throughput requirement implies satisfaction of the response time requirement.

The solution of the NFM gives the optimal flow rate in each arc, which shows how processing demands should be distributed from hosts to services. The allocation of demands includes computing power consolidations and isolations, the number of replicas of each task or service and the allocation of these services onto the virtualized nodes as well as the transaction flow rates etc. The solution can be converted from NFM to LQN. For details please refer to Step 4 in Section 5.5.

The generalization to a set of processors of different speeds is trivially made through the host capacities. In place of Ω_h = host multiplicity, we have Ω_h = host multiplicity × speed factor of each element. The speed factor is relative to the type of processor, regarded as the "standard" processor, for which the CPU demands are defined. Please note that this is a simplification, since the speed factor will actually be different for

different applications, depending on the processor architecture. If processor types are such that simple speed scaling is not possible, then the linearity of the problem is lost, and the NFM cannot be applied.

5.4 Step 2: Solve the Optimization Model

The resulting NFM model for execution demand optimization consists entirely of linear relationships, and with a linear objective function, such as execution cost, it forms a linear programming optimization problem. This makes the problem scalable.

Taking minimum execution cost as an example, an optimization model can be constructed as below. In the model every class of users has QoS requirements on response time (or throughputs) and populations; every host has constraints on the capacity utilization.

Optimization Model I. LP Model based on NFM

Objective: $\min \Sigma_{h,t} C_h \alpha_{ht}$

Constraints:

- Host computing capacity: for each h, $\sum_{t \in T} \alpha_{ht} \le \Omega_h$. If we wish to provide a safety margin, we can specify a maximum utilization fraction φ_h and require that $\sum_{t \in T} \alpha_{ht} \le \varphi_h \Omega_h$, for each h.
- Operations balance at each task: for each t, $\sum_{h \in H} \alpha_{ht} = \sum_{s \in S} \beta_{ts}$.
- Operations balance at each service: for each s, $\sum_{t \in T} \beta_{ts} = \sum_{c \in C} \gamma_{sc} + \sum_{s \in S} \Delta^i{}_s$
- Operations of service s used by user class c: $\gamma_{sc} = f_c d_{sc}$.
- Nonnegative operations: for all h, t, s, $\alpha_{ht} \ge 0$, $\beta_{ts} \ge 0$, $\gamma_{sc} \ge 0$.
- Throughput for each class exceeds the minimum specified in the service level agreement: for each c, $f_c \ge f_{cSLA}$.

The LP solution of the NFM can find the minimum execution cost for a deployment subject to processing capacity and user throughput constraints. It decides which tasks need replicas, where to allocate the replicas and how many requests are placed on the replicas.

5.5 Step 3 and Step 4: Insert Deployments and Solve the LQN

The optimal host-to-task flows in the optimization model determine the task deployments in the LQN. Where a task *t* has nonzero flow from a single host, this means it is deployed only on that host. However if it has non-zero flow from several hosts then task *t* is replaced by a set of identical replica copies (with the same set of entries), with the replica deployed on host *h* identified as task *t_h* and its replica of entry *s* identified as entry *s_h*. Each request to an entry of task *t* is split among the replicas in the same ratios as the NFM flows α_{ht}. To do this, each request arc to an entry of task *t* (say an arc from entry *e*) is replaced by a set of request arcs. The arc from entry *e* to entry *s*, labeled with y_{es} requests, gives an arc to entry *s_h* in task replica *t_h* labeled with y_{e,s_h} requests, with

$$y_{e,s\_h} = y_{es} \, (\alpha_{ht} / \Sigma_h \, \alpha_{ht}) \qquad (4)$$

and this is repeated for each replica of task *t*. The solution is found using an LQNS solver.

5.6 Step 5: Test for Convergence

If the NFM solution is overly optimistic after evaluation by the LQNS, then the surrogate flows must be adjusted, and the NFM must be run again. After a number of iterations, the NFM solution and the LQNS solution will converge to a solution that

satisfies all of the SLAs. This indicates that the surrogate flows have reserved enough additional capacity to account for contention. The result is a feasible solution.

Suppose an LQN has been solved at iteration i. The throughput of user class c in the LQN is indicated by $f^{i}_{c,LQN}$, and the shortfall in throughput (relative to the requirements) is e^{i}_{c}:

$$e^{i}_{c} = f_{c,SLA} - f^{i}_{c,LQN} \qquad (5)$$

When e^{i}_{c} is less than the allowed tolerance rate such as 1% of the $f_{c,SLA}$, it means that the optimal configuration for class c has been found, so the throughput and response times have converged to their target values. Iteration stops when every class has converged. However, convergence might be slow. Iteration can stop when the first feasible solution is found, which we will assume to be specified originally in terms of response times as $RT_c \leq RT_{c,SLA}$; This solution may not be optimal, but it is found relatively efficiently.

If the throughput given by the LQN solution does not meet the requirements, a new NFM is created, denoted NFM^{i+1} for the next iteration $i+1$, adjusted to deal with the shortfall. The shortfall in throughput is attributed proportionately to the demands for services, with an amount

$$e^{i}_{sc} = d_{sc}\, e^{i}_{c} \qquad (6)$$

for service s. The execution capacity provided for service s is augmented by this amount, by new surrogate flow

$$\delta^{i}_{s} = \Sigma_c\, e^{i}_{sc} = \Sigma_c\, d_{sc}\, e^{i}_{c}. \qquad (7)$$

at service s. The total surrogate flow at service s is represented by a fixed rate Δ^{i+1}_{s} used in the new NFM^{i+1}.

$$\Delta^{i+1}{}_s = \Sigma^{i}_{j=1}\ \delta^{i}_s \qquad (8)$$

This is indicated by an output arc (surrogate flow) from service node *s* with the label $[\Delta^{i+1}{}_s,\Delta^{i+1}{}_s,0]$. In the new NFM, the replicas of a task (if any) are treated again as a single task node. When the new optimization model NFM^{i+1} is solved, the total demand rate at the hosts will be increased by this amount. The iteration continues until enough resources are reserved to compensate for the performance lost due to contention, and the throughput requirement is met (or until the iteration limit).

5.7 Step 6: Linearized End-point Step (LEndStep)

The iteration of the optimization-LQN loop finds a feasible solution ($RT_c \leq RT_{c,SLA}$) quickly but may converge (i.e. $e^i_c \leq 1\% f_{c,SLA}$) slowly and not uniformly due to jitter in different deployment combinations. A feasible solution that has not converged tends to have a smaller-than-specified response time and over-allocated resources; a converged solution tends to have exactly the specified response time and less resource use. A final linearized analysis is used to achieve this goal.

LEndStep is conducted on the basis of the LQN model returned by the optimization loop. It freezes the deployments and linearizes the performance calculation with respect to the request rates between tasks, and then uses LP on the entire linear model (including linearized contention) to find a solution. In general this will save costs by reducing execution power.

5.7.1 Sensitivity Analysis of the Performance Model

In the LEndStep process sensitivity analysis estimates the change on throughputs due to the change of request rates. Sensitivity analysis provides a Jacobian matrix *J*, called a

sensitivity matrix, in which each element is the sensitivity of an entry throughput to a request rate, defined as:

$$J_{ij} = \partial \text{ (entry throughput\_}i\text{)}/ \partial \text{ (request rate\_}j\text{)} \quad (9)$$

Let J be the matrix, $\delta\phi$ be a vector of entry throughputs and δy be a vector of all the intertask request rate parameters. Each element $\delta\phi_e$ in the vector ϕ indicates the change of the throughput of entry e, due to changes in request rates. Then

$$\delta\phi = J\delta y \quad (10)$$

The sensitivity matrix $\delta\phi$ is approximated by the LQNX software (version 5.0 or later using finite differencing [105].

5.7.2 LP Model to Seek Minor Adjustment

Based on the above linear approximation via sensitivity analysis, the following LP model shifts request rates between replicas in order to reduce the execution costs.

Optimization Model II. LP Model in LEndStep

In the objective function, C_h is the cost per execution demand on the host h, ϕ_e is the throughput of an entry e, and D_e is the mean execution demand of the entry e.

$$\text{Objective: Minimize } \Sigma_e C_h \phi_e D_e \quad (11)$$

The objective aims to minimize the execution cost by optimizing the request rates across replicas.

Constraints:

1. $\phi = \overline{\phi} + J\delta y$ (12)
2. $\underline{\mathbf{1}}_E{}^T \phi \geq \phi_{E,SLA}$ (13)
3. $\overline{y} + \delta y \geq 0$ (14)
4. $\underline{\mathbf{1}}_{(e1, E2)} \delta y = 0$ (15)

5. $$0 \le \sum_{e \in E(h)} \phi_e D_e \le \Omega_h^* + \Omega_h^+ \sigma_h \qquad (16)$$

Constraint 1: the estimated throughput subject to the change of the request rates, calculated by the linear approximation with sensitivity analysis. In the constraint, $\overline{\phi}$ is a vector for the baseline throughputs returned from the iteration loop, $J\delta y$ is the variation on the throughputs ($\delta\phi$) due to the change of the request rates. δy are the variables in the optimization, and ϕ is the vector of resulting throughputs. The number of elements in ϕ is indicated by N_ϕ.

Constraint 2: $\phi_{E,SLA}$ is the SLA of the throughputs of entry E in the application template. A set of replicas initialized from the entry E is indicated by an indicator vector $\underline{\mathbf{1}}_E^T$, which has a 1 for the replica of the Entry E, 0 otherwise. The constraint requires that the new throughputs must satisfy the SLA requirement.

Constraint 3: limits the range of the change of request rate on each request arc.

Constraint 4: $\underline{\underline{\mathbf{1}}}_{(e1,\ E2)}$ is an indicator matrix consisting of $N_{Es}N_\phi$ rows and y columns. N_{Es} is the number of entries (E_s) that send requests which are collected in the vector y. Each row indicates the requests from a sender e_1 (where $e_1 \in E_s$) to the replicas e_2 which are the replicas of E_2 in the application template; the corresponding request is indicated by 1, otherwise 0. This constraint guarantees the change of request rates does not violate the request requirement in the template.

Constraint 5: CPU capacity constraint. E(h) are the entries deployed in the host h. The loadings on a host cannot violate the capacity of the host. This constraint is needed because a change of request rate may cause new contentions, which may violate the

linearization constraints, resulting in the performance failing to satisfy the requirements. The bound σ_h is introduced to maintain the validity of the linearization; σ_h must be no greater than 1. A large σ_h may help find a solution with lower costs, but it may cause big changes in the resource utilization, creating new contentions which may lead to some performance failing to satisfy the SLAs.

5.8 Case Study: Meet Response Time Goals at Low Execution Cost

This example looks for an economical solution giving the required response time for multiple classes of users. It simplifies the problem by only considering the cost of execution power and the capacity constraints, excluding the integer constraints imposed by memory, availability of licenses and fixed costs on the hosts. LEndStep is not applied.

For this case study we define a response time $RT_{c,\infty}$ as the response time that can be provided with infinite resources in the system, and define the target response time constraint as $1.02RT_{c,\infty}$, saying $RT_c \leq 1.02RT_{c,\infty}$. Solving the LQN with an infinite processor for each task estimates the value of $RT_{c,\infty}$ for every class. We use $f_{c,SLA}$, which corresponds to the $RT_{c,\infty}$, to update the throughput constraint in the above LP model.

Note that, in each round of optimization, the LP may think it is feasible to achieve the $f_{c,SLA}$ with finite resources because of underestimation of actual resources required by contention. The iterative process then keeps adding resources, LP thinks it is feasible, but the LQN solution shows that it isn't, until the response time is near to $RT_{c,\infty}$, with a difference less than the tolerance rate.

The decision algorithm was evaluated using a case study of a moderate-sized service system represented by the LQN in Figure 2.3, showing two classes of users. Class 1 has

250 users and class 2 has 100 users. The objective is to minimize the host computing costs while meeting the multiclass workload response time goals. Figure 5.5 shows the deployment of a single application.

The $RT_{c,\infty}$ is chosen to be the solution of the LQN with infinite threads and processors for each task, which makes $RT_{1,\infty}$ = 0.146 sec and $RT_{2,\infty}$ = 0.267 sec. The corresponding throughputs are $f_{1,SLA}$ = 250/(1 + 0.146) = 219.3/sec, and $f_{2,SLA}$ = 100/(1 + 0.267) = 78.9/sec.

The NFM optimization will determine the computing power needed to provide the best possible service on a certain set of hosts.

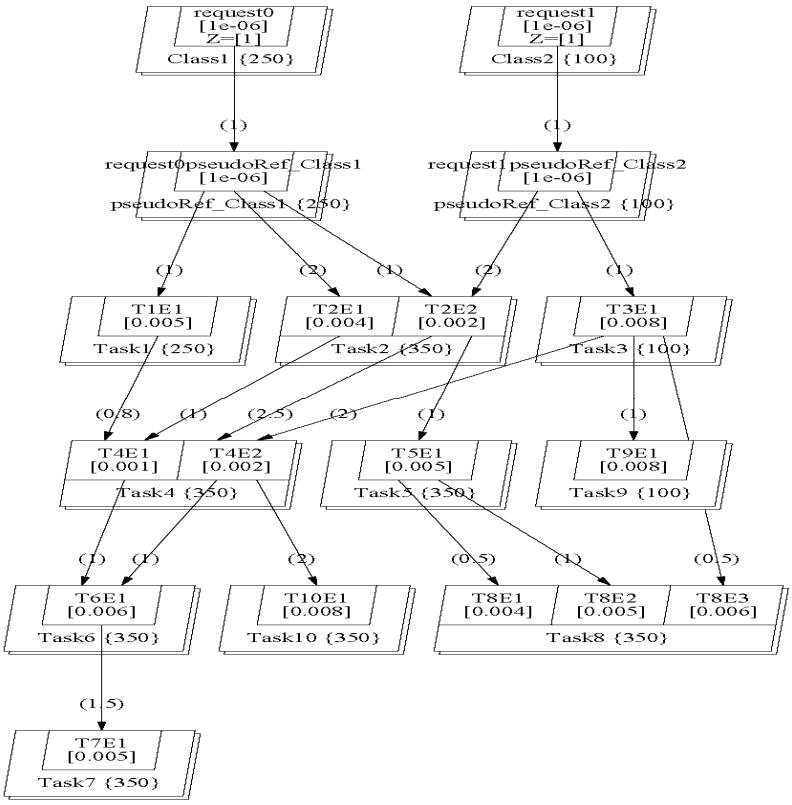


Figure 5.5 LQN Model of a Service Center

Table 5.2. Host Resource Attributes in the Example

Host h	m_h	φ_h	Speed Ratio	Ω_h	Cost C_h	Hostable Tasks
Host 1	20	80%	1	20	1	1,2,4,7
Host 2	20	80%	1.2	24	1.1	3,4,6,7
Host 3	20	80%	0.9	18	0.9	1,4,5,6
Host 4	20	80%	1.1	22	1.1	3,7,9,10
Host 5	20	80%	0.8	16	0.7	1,2,8,10
Host 6	20	80%	1.2	24	1.2	5,6,8,9

There are six hosts available with constraints as to the tasks that can be assigned to them. Demands are defined in CPU-seconds on a reference processor type, with a relative speed factor for each host. The resources at each node are described in Table 5.2. The cost C_h is relative to the standard host. The column headed m_h gives the multiplicity of each host. A fragment of the network flow model is shown in Figure 5.6. The thread pool size of each task in the LQN was set to handle 70% of the maximum possible demand rate at the task, a value found by experience to give good results.

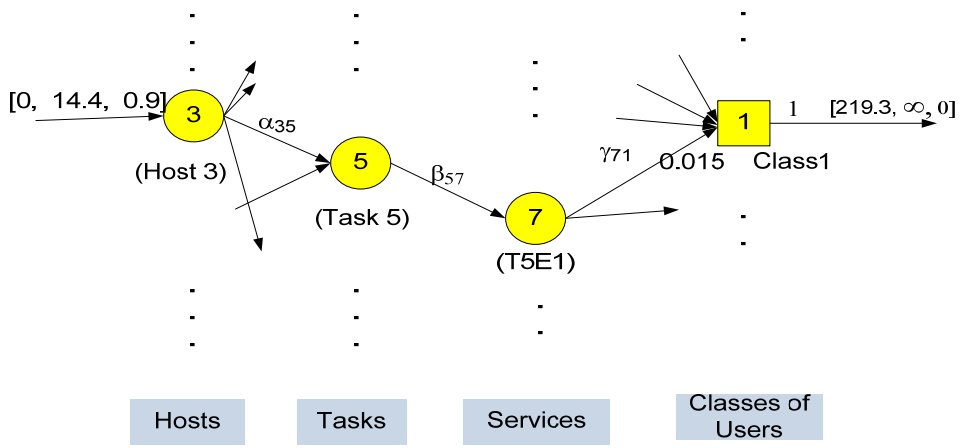


Figure 5.6 Fragment of Network Flow Model

The solution is found in five iterations. The performance of each class, the resource utilizations and the service allocation can be seen in Table 5.3 and Table 5.4. Table 5.3 shows the performance delivered to users in every class in each round. Table 5.4 shows the computing power consolidation in every node, in which the integer number indicates the required multiplicities of devices. Notice that a multiprocessor is fully utilized when its utilization equals its multiplicity.

Table 5.3 Response Times of Classes in Each Iteration

Class	Itn 1	Itn 2	Itn 3	Itn 4	Itn 5	Final	RT_∞	Excess
Class1	0.290	0.168	0.162	0.153	0.149	0.147	0.146	+0.68%
Class2	0.456	0.368	0.311	0.290	0.280	0.272	0.267	+1.87%

Table 5.4 Host Multiplicity at each Iteration, and Final Utilizations

Host	Itn 1	Itn 2	Itn 3	Itn 4	Itn 5	Final	Final Utilization
1	8	5	7	8	5	5	5 × 0.72
2	17	17	17	16	16	16	16 × 0.66
3	0	0	0	2	11	8	8 × 0.80
4	8	16	16	16	17	17	17 × 0.68
5	16	16	16	16	16	18	18 × 0.79
6	5	5	5	6	11	8	8 × 0.70

Figure 5.7 shows that tasks 6, 7, and 10 are replicated across multiple hosts, and that every host accommodates at least two tasks except host 5. The ratio of request flows divided between replicas has been determined by the relative flows, as described above.

Feasible goals can be provided by increasing the maximum response time limit. Factors of 1.1, 1.2,...,1.5 were applied to the required values of 0.146 sec for Class 1 and 0.267 sec for Class 2, to give the results in Table 5.5. The larger the response time limit,

the easier the problem. We can see that as the factor increases, the execution cost of the system required to meet the requirements decreases and the solution is found more quickly.

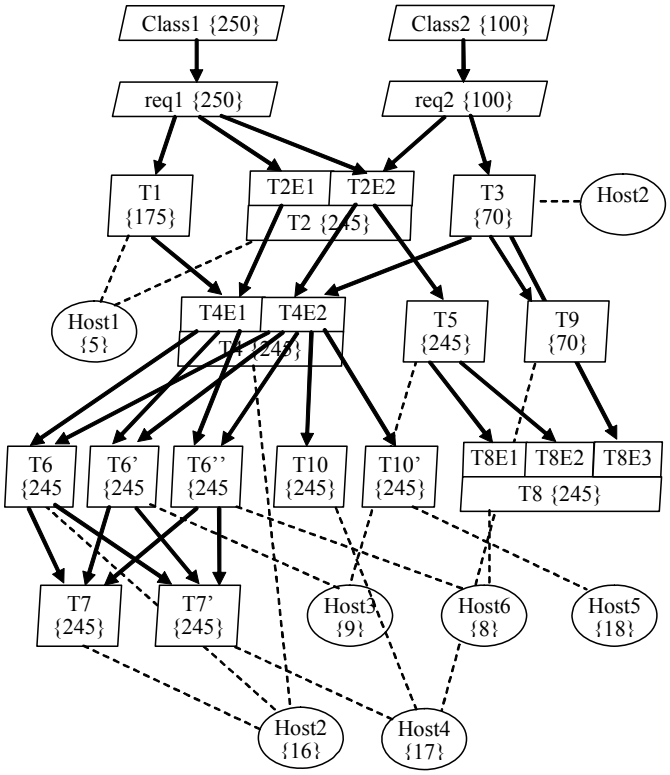


Figure 5.7 Near Optimal Deployment for the Service Center in Figure 5.5

Table 5.5 Execution Cost and Number of Iterations Corresponding to Relaxation of the Goals

Factor on Response Time	1.1	1.2	1.3	1.4	1.5
Execution Cost	61.07	57.31	55.66	52.91	51.69
Number of Iterations	7	4	4	3	3

Chapter 6 Discrete Optimization Algorithms for Static Deployment

The above simple optimization with NFM is an optimistic solution, which excludes several practical constraints such as memory requirement and availability, license costs and the power consumption associated with active hosts. To include the integer constraints, a MIP is formulated which replaces the LP step in the iterative scheme shown in Algorithm I in Chapter 5.

6.1 Optimization Model with Integer Constraints

6.1.1 Mixed Integer Programming (MIP) Model

Memory requirements, license availability and energy cost must be considered together for application deployments on a commercial service center. For example, on a Cloud infrastructure there is an assumption that every task has a VM to itself, and each VM needs a specific memory space. Each host to accommodate a VM must offer enough memory space to accommodate the VM. Commercial tasks have license constraints. Additional licenses must be purchased if the number of replicas exceeds the agreed

maximum number, so the availability of software licenses and the associated costs must be considered in a deployment decision. The Power cost (P_h) associated with the host activity affects the total cost of deployment. Power cost can be simplified as the total of execution and fixed costs. Following the literature [46], execution cost can be approximated as being linearly related to CPU utilization, although the power cost is actually a nonlinear function of load. Fixed costs are additional, representing some fundamental operations of the machine. Fixed costs can be defined as a cost per active host regardless of the size of workloads running. Because the CPU utilization is proportional to the capacity used (Ω^*_h), reusing the NFM model, the total power cost of a host can be mathematically described with linear approximation, shown as below,

$$P_h = C_h \Omega^*_h + C_{fh} \quad (17)$$

where C_h and C_{fh} are model specific constants, which can be estimated by learning techniques such as linear regression. Because power cost is the rate of energy use, the energy cost during a period of time of duration Δt is the total power consumption calculated by $P_{avg}\ \Delta t$, where P_{avg} is the average power consumption. Figure 6.1 shows an example of linear approximation of the power costs subject to the CPU utilization[46].

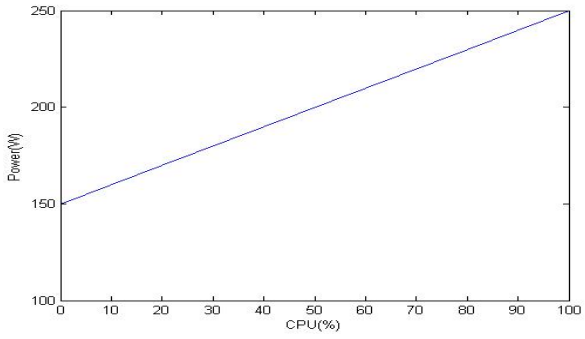


Figure 6.1 Linear Approximation of the Power Consumption against the Utilization of CPU

Let a binary variable A_{ht} indicate an allocation of tasks to hosts, shown by arcs having positive flow.

$A_{ht} = 1$ if $\alpha_{ht} > 0$, $A_{ht} = 0$ otherwise

which allows modeling the problem as a mixed integer program, as shown below. The model assumes that execution cost is charged per second of actual execution, including capacity reserved to reduce contention. The solution returned by the MIP is the optimal workload distribution across the system. It accounts for the workload balance and the effects of contentions, as well as deciding which tasks should be duplicated and where to place the replicas. The solution seeks the optimal configuration that has the minimum costs for energy and license while satisfying multiple goals.

This model considers memory as a hard constraint and the limits on the number of licenses as soft constraints. This means that a task only can be deployed on a host that can provide sufficient memory space, but extra licenses can be added at extra cost when additional replicas are needed. The variables and symbols used by the MIP are defined as below,

Table 6.1 Variables and Parameters used in the MIP

L_t'	Extra number of license in use, integer
L_t	The number of avialble licenses
p_{arc_ht}	the reward/penalty on the *arc* connecting host *h* and task *t*, default $p_{arc\_ht} = 0$
C_h	Execution cost of a host *h*, a cost factor for a unit of execution on this host
C_{Lt}	Pay-per-use cost of every extra license for task *t*, beyond L_t

C_{fh}	Fixed cost of host h, associated with host activity
$BigC$	A very large number
m_t	Memory requirement of task t, in order to execute (assumed the same for all nodes)
M_h	Memory Capacity of host h, memory available for application tasks
Δ^i_s	The total surrogate flows (virtual demand) of service s at iteration i . For the calcualation please refer to Section 5.6
$T(h)$	The collection of tasks hostable by host h
A_{ht}	Binary variable, to indicate if task t is assigned to the host h
S_h	Binary variable, to indicate host activity

Optimization Model III. MIP Model

Objective function:

Minimize: $\Sigma_{ht}\, C_h\alpha_{ht} + \Sigma_{ht}\, p_{arc\_ht}\, A_{ht} + \Sigma_t\, L_t'\, C_{Lt} + \Sigma_h\, S_h C_{fh}$ (18)

over A_{ht}, L_t' S_h, α, β, γ subject to constraints:

Constraints:

1. for each host, capacity of host h: $\Sigma_t \alpha_{ht} \le \Omega_h$ (19)
2. for each task, flow balance at node t: $\Sigma_h \alpha_{ht} = \Sigma_s \beta_{ts}$ (20)
3. for each service, add surrogate flows at node s: $\Sigma_t \beta_{ts} = \Sigma_c \gamma_{sc} + \Sigma_s \Delta^i_s$ (21)
4. for each class, flow proportion at node s: $\gamma_{sc} = f_c d_{sc}$ (22)
5. set A_{ht}=1 for arcs having positive flow: $\alpha_{ht} \le A_{ht}.\ BigC$ (23)
6. memory space at h: $\sum_{t \in T(h)} m_t A_{ht} \le M_h$ (24)
7. license constraint: $\Sigma_h\, A_{ht} \le L_t + L_t'$ (25)
8. Set S_h = 1 if any tasks are assigned to this host, for each h: $A_{ht} \le S_h.$ over all t (26)

where Δ_s^i is determined by the LQN part of the iteration and is initially 0.

Variables:

1. *Integer variable*, licenses used in excess of L	$L_t' \geq 0,$	(27)
2. *binary variables*	A_{ht} , S_h = 0 or 1	(28)
3. *continuous variables,* all flows are non-negative	$\alpha_{ht} \geq 0,\ \beta_{ts} \geq 0,\ \gamma_{sc} \geq 0$	(29)
4. *continuous variables,* SLA flow constraint	$f_c \geq f_{c,SLA}$	(30)

In the objective, the first term stands for the execution cost, the second term prioritizes the use of arcs based on a reward/penalty scheme, the third term accounts for the license costs and the last one is for the fixed cost.

Some constraints need further explanation:

Constraint 3: Operations balance at each service, including surrogate flows: for each s. Δ_s^i is the amount of surrogate flow returned by the LQN model in the *ith* iteration, indicating additional capacity that should be reserved to reduce the contention delays in the $(i+1)th$ MIP optimization. $\Sigma_S \Delta_s^i$ is the size of the surrogate flow at the *ith* iteration.

Constraint 5: Constraint to determine A_{ht}. *BigC* is a very large positive constant. When the arc is used ($\alpha_{ht} \geq 0$), A_{ht} must be 1 to satisfy the constraint; otherwise, A_{ht} will be 0. Though the constraint can be satisfied with $A_{ht} = 1$ when $\alpha_{ht} = 0$, this is penalized by the objective value, so $A_{ht} = 0$ is chosen when $\alpha_{ht} = 0$.

Constraint 7: Soft constraint on license. L_t' indicates the number of extra licenses used.

Constraint 8: Constraint to decide the value of S_h. S_h is a binary variable to indicate the activity of a host. When an allocation is used (A_{ht} is 1), then S_h must be 1 to satisfy the constraint and the fixed cost of the host is counted in the objective function; otherwise S_h is 0.

In the model, p_{arc_ht} is a reward or penalty on arc ht. Rewards have negative values to increase the priority that an arc is chosen, and penalties have positive values of p_{arc_ht} to discourage the use of the arc. Smaller values of p_{arc_ht} imply higher priority. The value of p_{arc_ht} is determined by how difficult it is to install a new replica. p_{arc_ht} is an effective tool to improve robustness in the face of dynamic changes. The strategy for adjusting p_{arc_ht} will be introduced in Section 7.1.2.

6.2 Solving the MIP

The MIP model can be solved with advanced algorithms via such MIP solvers as CPLEX [23], which provides APIs to construct MIP models, solve the model and return the optimal value for each variable. In the MIP described in Optimization Model III, CPLEX looks for the optimal values for the continuous and discrete variables subject to the goals and constraints.

However, since solving MIPs is NP-hard, it could be very time consuming to solve a large-scale deployment problem with many tasks and hosts. To address this issue, two heuristics are introduced to permit the solution of large models: Heuristic Packing (HP) and Heuristic MIP (HMIP). The number of variables in the MIP model is determined by the numbers of tasks and hosts. Assuming each task is hostable by every host, then n hosts and m tasks creates $n \times m$ arcs, giving $n \times m$ variables representing flow rates, $n \times m$

binary variables indicating the allocations, and m variables for licenses, as well as n variables to account for power consumption. These variables allow over $2^{n \cdot m}$ allocation options. As a result, the problem size explodes combinatorially with the increase in the number of tasks and hosts as the model grows.

HP uses an efficient packing heuristic to assign tasks to hosts. HMIP uses an initial loose packing to create a smaller MIP that is solved exactly; this is faster, but the guarantee of finding the true optimum is lost.

Table 6.2 Variables used in HP and HMIP

Ω^+_h	remaining execution demand space of host h
Ω^*_h	The used capacity of host h, $\Omega^*_h = \Omega_h - \Omega^+_h$
M^+_h	remaining memory space of host h
M^*_h	The used memory space of host h, $M^*_h = M_h - M^+_h$
d^+_t	remaining execution demand of task t
L^+_t	Remaining available licesne for use
L^*_t	Total number of licenses in use, $L^*_t = \Sigma_h A_{ht}$
L_t'	Extra number of licesnes in use, integer

6.2.1 Algorithm: Heuristic Packing (HP)

The heuristic packing (HP) algorithm is motivated by bin-packing. It includes two steps. Step I aggregates workloads on the lowest-cost hosts if the capacity can be fully used. But the constraints may limit the utilization of the selected hosts in Step I, making the selection non-optimal. In order to achieve a better solution, Step II then seeks other more suitable hosts to replace the hosts that are not fully utilized in Step I. The algorithm is shown below.

Algorithm II. Heuristic Packing (HP):

// allocation function

Allocate (*t, d, h*)

//allocate demand d for task t to host h, and adjust the remaining demand d^+_t and available memory M^+_h

1. Set $\alpha_{ht} = d$,
2. Decrement Ω^+_h by α_{ht},
3. Decrement d^+_t by α_{ht},
4. Decrement M^+_h by M_t
5. Increment L^*_t by 1

// HP algorithm

STEP I

1. Sort the tasks in decreasing order of d^+_t/L^+_t, subordered by decreasing order of $C_{Lt.}$ *//This gives priority to tasks with the most demand and fewest licenses.*
2. For each task *t* in order:
   a. Sort the hosts with M^+_h greater than M_t, by their CPU execution demand space Ω^+_h (largest first) and designate *h*(*i*) as the *i*th host in order, break tie by the total cost. Define these sorted hosts as ***I***.
   b. Set $i = 1$ (allocate first to host *h(1)*)
   c. if $d^+_t > 0$
      i. execute **allocate** (t, min(d^+_t , $\Omega^+_{h(i)}$), *h(i)*) *// maximize the use of host i.*
      ii. If $d^+_t > 0$ and there are available hosts remaining in ***I***, then increment *i*, repeat from Step 2.c, else, exit and return error message "not enough available hosts"
   d. if $d^+_t = 0$
      i. Sort the hosts with $\alpha_{ht} > 0$ by α_{ht} (largest first), and designate *h(j)* as the *j*th host in this order. Define these sorted hosts as ***J***
         1. $d = \min(\alpha_{h(j)t}, \Omega^+_{h(i)})$
         2. $C_{h(j)t} = \max(M_t/M_{h(j)}, d/\Omega_{h(j)})\, C_{fh(j)} + d \cdot C_{h(j)}$
         3. $C_{h(i)t} = \max(M_t/M_{h(i)}, d/\Omega_{h(j)})\, C_{fh(i)} + d \cdot C_{h(i)}$
      ii. If $C_{h(j)} > C_{h(i)}$ or ($C_{h(j)t} = C_{h(i)t}$ and host *i* is hosting at least one task but has spare capacity):
         1. If $d = \alpha_{h(j)t}$ *// allow migration*
            a. Then move task *t* from host *j* to *i*. increment *j* if there are hosts remaining in ***J***, repeat from 2.d *// move the task to the low cost host*
         2. If $d = \Omega^+_{h(i)}$ then:
            a. If $L^+_t > 0$ or $\alpha_{h(i)t} > 0$ or $C_{h(j)t} - C_{h(i)t} > C_{Lt}$
               i. Then **allocate** (*t,* $\Omega^+_{h(i)}$, *h*(*i*)), increment *i* if there are available hosts remaining in ***I***.

// add a replica of t on the host i, if spare license.

iii. Else

increment *j* if there are hosts remaining in ***J***, repeat from 2.d

e. increment *i* if there are available hosts remaining in ***I*** *// the new selected host has less cost than the ones in use*

STEP II

1. For each host *i* which is not selected in STEP I
   a. For each host *j* which has been selected to host tasks:
      i. If $\Omega^{+}_{h(i)} > \Omega^{*}_{h(j)}$ and $M^{+}_{h(i)} > M^{*}_{h(j)}$ and $C_{fh(i)} + C_{h(i)}\,\Omega^{*}_{h(j)} < C_{fh(j)} + C_{h(j)}\,\Omega^{*}_{h(j)}$ then:
      Move all tasks from host *j* to host *i*.

In HP, Step I includes a set of operations of migration, aggregation and replication. This step takes account of the fixed cost, execution cost and license availabilities. It approximates the share of fixed costs at a host *h* (C_{fh}) by tasks proportional to the share of the resource utilization, estimated with $\max(M_t/M_h, d/\Omega_h)C_{fh}$, though fixed cost ($C_{fh}$) is independent of the size of the load. Let C_{ht} be the costs due to task *t* on host *h*, consisting of the share of fixed costs (C_{fh}) and the execution costs ($d \cdot C_h$), in which C_h means the cost per demand in host *h*, shown as below. The goal of Step I then is to minimize $\Sigma_{ht}\ C_{ht}$, where:

$$C_{ht} = \max(M_t/M_h, d/\Omega_h)\ C_{fh} + d \cdot C_h \qquad (31)$$

The packing in Step I assumes each selected host will be fully utilized in the solution. However, this assumption is optimistic.

Step II considers hosts that have large spare capacity remaining after Step I. Other hosts may be able to handle their tasks with lower cost consumption. Step II addresses this issue by moving tasks from the under-utilized hosts to other hosts that can

accommodate the operations with lower costs. Step II assumes that all tasks currently deployed on a high-cost host are hostable in a low-cost host, so the new host can replace the old host.

6.2.2 Algorithm: Heuristic MIP (HMIP)

Heuristic Packing is efficient in finding a feasible solution, and in some cases, the quality of optimization is similar to that obtained using exact MIP solvers. However, the quality is not guaranteed. So the two are combined in a new algorithm called HMIP, shown as Algorithm III. HMIP simplifies the MIP problem by reducing the optimization options with HP first, gaining the ability to handle larger problems, and then using a MIP solver to optimize based on the subset of selected hosts.

Algorithm III. HMIP

1. Heuristic Packing (HP) performed as described in Algorithm II.
2. Construct and solve a MIP as in Optimization Model III, using only the hosts selected in Step 1, and allowing any task potentially to use any host in the selected subset.

The feasible solution returned by HP (C_{HP}) can be used as an incumbent in the MIP solution to reduce the optimization time. This extra information is added as a constraint in the MIP:

$$\Sigma_{ht}\, C_h \alpha_{ht} + \Sigma_t\, L_t' C_{Lt} + \Sigma_h\, S_h C_{fh} \leq\ C_{HP} \qquad (32)$$

6.2.3 Evaluation of HP and HMIP

A set of experiments was conducted to compare the performance of the pure MIP via an exact MIP solver with HMIP and HP. The test bed simulates a system with 1 to 50 applications to be deployed on a host pool that consists of 5 types of hosts, as shown in Table 6.4. The ratios between the C_h and C_{fh} are approximated based on the experimental results given in [96][24][42][46], ranging between 0.15 to 1; the fixed cost is about 40~60% of the total energy cost when a host is fully utilized. The value of the cost is relative to a standard host. Each application has 10 independent tasks, each of which has different randomly chosen CPU demand, memory requirement, license availability and cost. Each type of host in the pool has 1000 fully identical hosts, which can host arbitrary tasks. The branch-and-bound solver CPLEX is used to solve the MIP problems. Aggressive probing and strong branching are deployed to improve the efficiency and give good solutions for large and difficult MIP problems. The optimization solver CPLEX in the experiments in this thesis uses the default configurations except the following parameters.

Table 6.3 CPLEX Configurations

Aggressive probing	enable
Strong branching	enable
Relative MIP gap tolerance	0.01

The MIP terminates at the convergence rate of 1%, meaning the MIP optimization stops as soon as it has found a feasible solution proved to be within one percent of the optimal, or it terminates at 350 seconds if a solution cannot be found.

Table 6.4 Host Information

Type	Relative Speed	Relative Memory	cost per execution demand (C_h)	Fixed cost (C_{fh})
A	1.8	2	0.5	0.6
B	2.4	4	0.45	0.81
C	2.8	8	0.4	1.12
D	3.2	12	0.35	1.4
E	3.6	16	0.3	1.62

The quality and effort of optimization depend on how easy it is to fit the applications into the available processing resources. We will call a situation with just enough resources (or not quite enough) a "high-stress" situation, and one with plenty of resources, low stress. The *stress rate* indicates the ratio of the demands to the available total capacity for execution:

$$Stress\ Rate = \Sigma_t\, d_{t,SLA} \,/\, \Sigma_h \Omega_h \qquad (33)$$

Four values of the stress rate are used, from 0.25 to 0.9; in each case the number of hosts is adjusted to give a stress rate within 0.05 of the nominal shown. At each stress rate, there is the same number of hosts of each type.

The goal is to evaluate the scalability of the algorithms so we only use a single scenario. The evaluation in each case uses the same set of applications and hosts. Optimization quality and efficiency are the two measured metrics.

Table 6.5 Pure MIP, Heuristic and HMIP Comparison (Without Contentions)

Stress rate	**Very High (0.9±0.05)**			**High (0.7±0.05)**			**Medium (0.5±0.05)**			**Low (0.25±0.05)**		
	HP	MIP	HMIP	HP	MIP	HMIP	HP	MIP	HMIP	HP	MIP	HMIP
1 app	Host Pool Size: 8 Number of Tasks: 10			Host Pool Size: 10 Number of Tasks: 10			Host Pool Size: 14 Number of Tasks: 10			Host Pool Size: 25 Number of Tasks: 10		
objective	11.9	**11.48**	11.9	11.6	**11.48**	11.6	11.6	**11.41**	11.6	**11.24**	**11.24**	**11.24**
Solution time (sec)	**0.016**	0.11	0.125	**0.016**	0.172	0.188	**0.015**	0.25	0.109	**0.016**	0.125	0.093
# of variables in MIP	-	127	97	-	157	97	-	217	97	-	382	82
5 app	Host Pool Size: 12 Number of Tasks: 50			Host Pool Size:16 Number of Tasks: 50			Host Pool Size: 22 Number of Tasks: 50			Host Pool Size: 39 Number of Tasks: 50		
objective	23.89	**22.88**	**22.88**	23.92	22.87	**22.72**	23.2	22.75	**22.57**	**22.3**	**22.3**	**22.3**
Solution time(sec)	**0.015**	0.375	0.25	**0.015**	3.735	0.766	**0.016**	0.703	0.328	**0.015**	0.453	0.203
# of variables in MIP	-	887	887	-	1171	816	-	1597	816	-	2804	674
10 app	Host Pool Size: 17 Number of Tasks: 100			Host Pool Size: 23 Number of Tasks: 100			Host Pool Size: 32 Number of Tasks: 100			Host Pool Size: 57 Number of Tasks: 100		
objective	34.04	33.15	**33.13**	34.38	**32.82**	33.14	46.79	**32.65**	32.85	35.2	**32.2**	**32.2**
Solution time(sec)	**0.015**	0.39	0.562	**0.016**	6.969	0.422	**0.015**	6.344	1.734	**0.031**	0.891	0.422
# of variables in MIP	-	2467	2467	-	3313	2326	-	4582	2326	-	8107	2044
20 app	Host Pool Size: 25 Number of Tasks: 200			Host Pool Size: 33 Number of Tasks: 200			Host Pool Size: 46 Number of Tasks: 200			Host Pool Size: 86 Number of Tasks: 200		
objective	52.73	52	**51.94**	56.52	52	**51.99**	68.69	**51.34**	51.28	52.35	50.98	**50.88**
Solution time(sec)	**0.016**	1.641	2.219	**0.031**	4.281	3.203	**0.031**	25.953	7.735	**0.062**	68.657	2.562
# of variables in MIP	-	7165	7165	-	9413	6884	-	13066	7165	-	24306	6322
30 app	Host Pool Size: 41 Number of Tasks: 300			Host Pool Size: 53 Number of Tasks: 300			Host Pool Size: 75 Number of Tasks: 300			Host Pool Size: 139 Number of Tasks: 300		
objective	85.57	**84.43**	85.15	106.41	84.11	**84.06**	86.1	83.42	**83.33**	84.51	**81.95**	**81.95**
Solution time(sec)	**0.031**	4.031	5.469	**0.046**	6.266	7.625	**0.078**	51.687	9.672	**0.109**	21.422	4.328
# of variables in MIP	-	17471	17471	-	22523	16629	-	31785	16629	-	58729	14945
40 app	Host Pool Size: 50 Number of Tasks: 400			Host Pool Size: 64 Number of Tasks: 400			Host Pool Size: 89 Number of Tasks: 400			Host Pool Size: 169 Number of Tasks: 400		
objective	105.48	**102.75**	**102.75**	107.12	**101.71**	102.66	104.72	**100.76**	**100.75**	102.13	Out of Memory	**101.01**
Solution time(sec)	**0.047**	8.125	18.953	**0.063**	100.81	13.688	**0.094**	165.83	27.093	**0.141**		9.047
# of variables in MIP	-	28330	28330	-	36184	26647	-	50209	26647	-	95205	23842
50 app	Host Pool Size: 63 Number of Tasks: 500			Host Pool Size: 81 Number of Tasks: 500			Host Pool Size: 113 Number of Tasks: 500			Host Pool Size: 216 Number of Tasks: 500		
objective	133.94	**130.77**	132.2	134.86	130.91	**130.77**	135.21	**130.28**	130.32	130.58	Out of Memory	**128.56**
Solution time(sec)	**0.062**	42.578	28.296	**0.062**	Time out (350)	54.547	**0.094**	Time out (350)	57.343	**0.187**		19.359
# of variables in MIP	-	44513	44513	-	57131	41709	-	79563	42410	-	152103	37503

Results are given in Table 6.5 for the objective cost, solution time and the number of all MIP variables (MIPvar, including continues and discrete variables) comparison. The best numbers for each model are shown in boldface (lowest objective value, lowest solution time).

- The evaluation shows that all problems with less than 10,000 MIP variables can be solved by CPLEX in less than 10 seconds. This many variables corresponds to an HMIP formulations with about 20 applications on 100 hosts or a pure MIP formulations with 20 applications on 35 hosts.
- Most MIP problems that have 10,000~20,000 variables can be addressed in 10 seconds. In HMIP all problems with about 35 applications on 150 hosts are at this size. In pure MIP it corresponds to a problem with 30 applications on 50 hosts or 20 applications on 80 hosts.
- For the larger MIP problems with over 20,000 variables, the effort is highly variable.
- Low stress cases gave larger MIPs than high stress, and also have a bigger reduction in MIP size between the pure MIP and the HMIP. Roughly speaking there was little or no reduction in MIPvar at stress rate 0.9. A reduction of one third at 0.7, half at 0.5 and three quarters at stress rate 0.25, up to the point where the problem is too large for CPLEX.

- In most cases, the pure MIP gives the best objective function value, but it takes more solution time. When the size of the problem increases to extremely large, pure MIP either reaches the limit on memory or cannot find a solution within an acceptable time. Moreover, the increase in the number of hosts may result in an explosion in the size of the MIP problem, giving a longer solution time.
- HP is very fast, but it cannot ensure the quality of the optimization. In some cases it is within a few percent of the optimum, but with medium stress (0.5) and 10 or 20 applications, it is much worse.
- Looking at different problem sizes, the solution time for HP increases with decreasing stress (since there are more alternatives). For MIP and HMIP the solution time tends to be largest for intermediate stress cases (stress rate = 0.5).
- HMIP can give a solution with almost the same quality as pure MIP in all cases (less than 2% difference), and in most cases, it is much faster, especially when the stress rate is low. It can handle much larger problems than pure MIP, but the quality of HMIP is affected by the solution given by the initial Heuristic Packing step. When a large-scale deployment needs to have a high quality solution in a limited amount of time, HMIP is the best algorithm to use.

Note that in some cases the results given by HMIP are slightly better than pure MIP. This results from the termination criteria, which stops the optimization at a solution close

to the optimum, though better solutions may exist. The quality of the optimization is thus determined by the first solution that satisfies termination criterion, so HMIP may perform better than MIP in some cases.

Please also note that the pure MIP is faster than HMIP in a few cases, though the problem size in HMIP is much smaller. The number of variables and constraints is one of the factors affecting the solution speed. Reducing the number of possible solutions (indicated by the number of variables) can help to improve the efficiency, but not always, since MIP is NP-hard. The search strategy chosen by the optimization solver has a great impact on the optimization quality and efficiency. A search strategy that is efficient in addressing a pure MIP model might be slow for the corresponding HMIP model simplified from the original MIP problem. Though the current optimization mechanisms in CPLEX try to give the best strategy in terms of the MIP model, it cannot guarantee that the strategy chosen is the most suitable.

Comparing the objective costs shows that in all cases HMIP is able to give a high-quality optimization competitive to that returned by a pure MIP via CPLEX. Results show that the solution time of both algorithms may exponentially increase with the growth of the number of applications, but the solution time for the pure MIP increases faster than HMIP, which corresponds to the increase in the number of variables in MIP and HMIP. For high stress models, MIP starts to show an obvious latency for problems with 40 applications, while HMIP shows latency for problems with 50 applications. For problems with medium or low stress rates, the algorithms start to show different solution times when the number of applications scales beyond 20. The smaller MIPs solved by

HMIP allow it to solve problems with over 40 applications at 0.25 stress rate, but pure MIP reaches the memory limitations on these problems.

These results demonstrate that HMIP is able to give high quality decisions, nearly as good as pure MIP, but with much greater efficiency in most of the problems, especially those having medium or low stress rates or a large number of applications.

6.3 Revenue Model Supported by the Optimization

The approach presented above gives an approach to minimizing the total costs associated with deployments, and accounts for several goals, including execution demands, contentions, energy consumption, memory, and license availability. This model can be extended to support other optimization problems such as the revenue model.

In the revenue model, assuming:

- Each service class *c* offered to users has a price per response of P_c,
- Each application *App* to provide services has execution and fixed costs on hosts, as well as costs on extra licenses, described as

$$\text{Cost}_{App} = \Sigma_{ht}\, C_h \alpha_{ht} \; + \; \Sigma_{ht}\, p_{arc\_ht}\, A_{ht} + \Sigma_t\, L_t\, 'C_{Lt} + \Sigma_h\, S_h C_{fh} \qquad (34)$$

The *t* in the equation represents the tasks used by the application.

Then an application *App* has a profit

$$PROFIT_{App} = \Sigma_{c\ in}\, e_{App}\, P_c f_c \; - \text{Cost}_{App} \qquad (35)$$

The optimization for a host provider is to maximize the total profits

$$TOTAL = \Sigma_{App}\, PROFIT_{App}$$

subject to constraints described in Optimization Model III. When the total profit to the application providers is maximized, presumably the host provider share of this is also maximized, although the mechanism for sharing this is not considered here.

An objective function can be described as:

$$\text{Max } \Sigma_{App} \, PROFIT_{App} \qquad (36)$$

Moreover, because some classes may have a constraint on the maximum throughput resulting from the limit on the number of users, an upper bound on the throughput f_c may be needed for these classes.

6.4 Case Study: Deployment with Multiple Goals with Contentions

This example leverages a small application shown in Figure 6.2 to conduct an evolution to test the feasibility of the approach subject to a diversity of constraints. The application has 287 users, and is required to give an average response time described in the SLA of 28.98ms. The values of these QoS requirements are randomly generated for the testing. This test accounts for the goals of:

- Average response time in SLA
- Resource availability and requirements (CPU and Memory)
- License availability and license cost (the number of free licenses for each task is different, as shown in Table 6.7.)
- Power Consumption (execution and fixed costs)
- Effects of contentions

The evaluation is conducted on the host pool shown in Table 6.4 with a stress rate of 0.7.

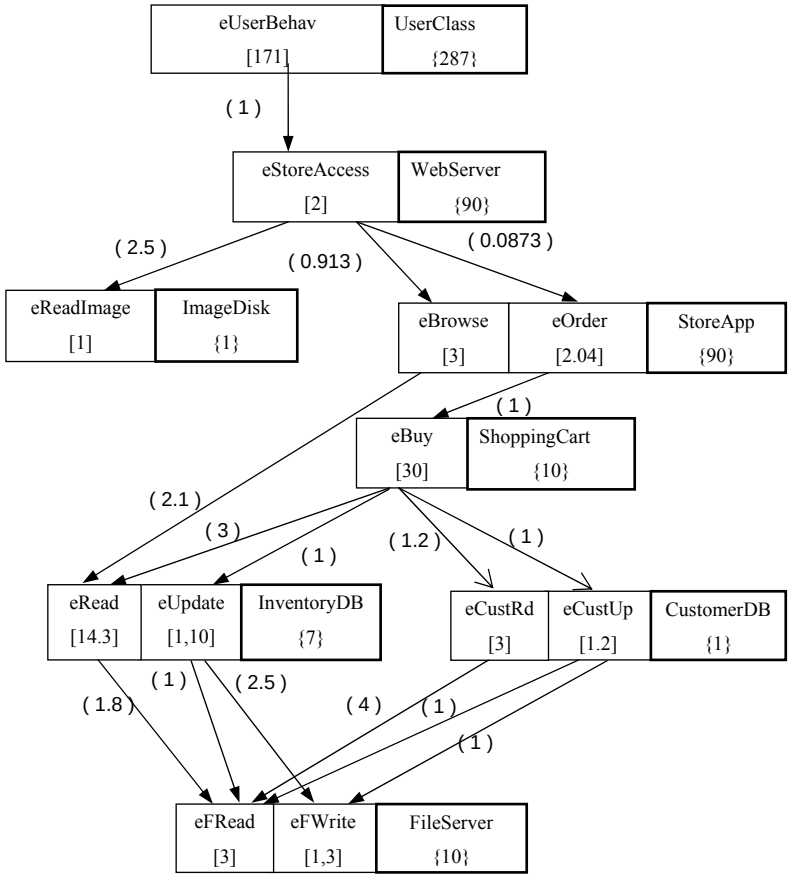


Figure 6.2 the LQN Model of an Application to be Deployed with Multiple Goals

Based on the LQN and the available hosts in the pool, an optimization model can be constructed as a MIP in the form of Optimization Model III, solved with HMIP. The MIP solver and the LQN solver are employed in the iterative loop described in Chapter 5. This process takes QoS as a goal of the optimization while accounting for contention.

The optimization results given below show that the optimization loop achieves a feasible solution in three iterations, giving a response time of 27.21ms. Adjustment of request rates with LEndStep tunes the loadings between replicas, achieving the final

response time of 28.74ms and saving 0.02 execution cost. This final result deviates from the goal by -0.83%. The total optimization process takes 8.5 seconds, in which 6.63 seconds are used by the sensitivity analysis.

Table 6.6 Response Time and the Associated Cost in Each Iteration

	Iteration 1	Iteration 2	Iteration 3	Sensitivity with LP	Goal	Error
Response Time	34.80ms	32.47ms	27.21ms	28.74ms	28.98ms	-0.83%
Execution Cost	4.93	4.83	5.07	5.05	-	-
Fixed Cost	6.58	6.86	6.88	6.88	-	-
Total Energy and License Cost	11.51	11.69	11.95	11.93	-	-
Solution Time	0.734	0.547	0.594	6.63	-	-

Table 6.7 shows the use of licenses for each task and the original license availability. Resource sharing based on power consolidation means that 13 replicas can be processed with 6 physical hosts. Figure 6.3 gives the utilization of memory and processors, showing that the resource utilizations are balanced between the hosts.

Table 6.7 the Use of License of Each Task (HMIP)

Task Name	Number of license used	Number of free license
TWebServer	1	3
TImgDisk	2	3
TInvDB	3	5
TFS	3	6
TCustDB	1	2
TCart	2	4
TStoreApp	1	6

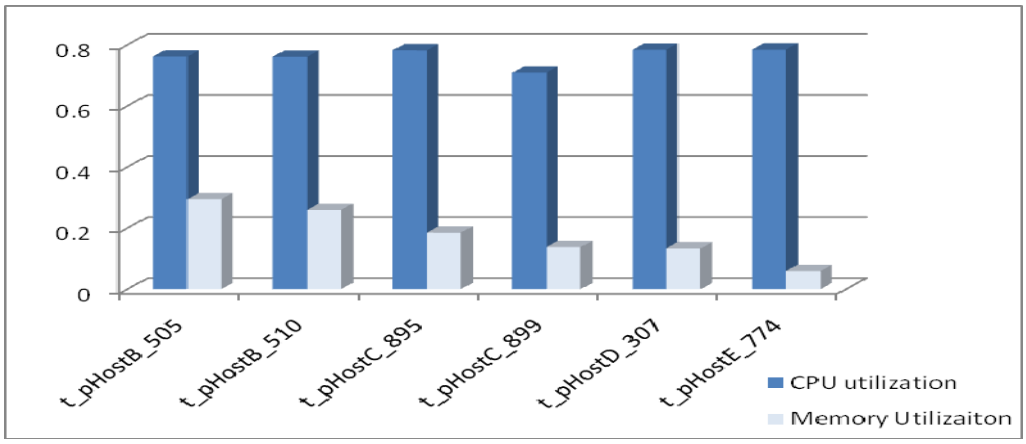


Figure 6.3 the Use of CPU and Memory in Hosts (HMIP)

This simple case study demonstrates that the approach can satisfy multiple goals at the same time. The optimization creates replicas for tasks, balances workloads, and optimizes allocations, achieving the required performance with an economical solution.

6.4.1 Comparison with Other Approaches

Many existing systems use packing approaches to handle task deployments. Experiments are conducted to compare the effectiveness of HMIP, the Power-minimizing Placement Algorithm (mPP) introduced in [96] and the simple greedy approach shown below,

Simple Greedy Approach:

T: a collection of tasks to be deployed.

I: the number of hosts

1. Sort hosts in increasing order by the maximum power consumption calculated as $C_{fh} + \Omega_h C_h$.
2. Set $i = 1$,
3. For each t in **T**
   - **3.1.** If $i \leq$ I then:
     - **3.1.1.** Execute **Allocate** (t, $\min(d^+_t, \Omega_{h(i)})$, $h(i)$) *// allocation function is the same as the one used in Algorithm II*

3.1.2. If t has remaining execution demand ($d^{+}_{t} > 0$) then increment i and repeat from Step 3.

3.2. Else ($i > I$) return error "out of hosts"

3.3. Next t and increment i

Table 6.8 compares the effectiveness of HMIP, Simple Packing and mPP with the same case studied above.

Table 6.8 Comparison of the HMIP and Simple Packing

	HMIP	Simple Greedy Approach	mPP
Response Time (Goal: 28.98ms)	28.74ms	34.10ms	41.78ms
Number of Hosts Used	6	13	10
Total Execution and Fixed Cost	11.93	14.28	12.34
Execution Cost	5.05	6.48	6.34
Fixed Cost	6.88	7.80	5.99
Average CPU Utilization Per Host	0.76	0.55	0.70

Because the simple greedy approach and mPP do not account for the QoS requirement, it is not surprising that the solutions violate the QoS constraint. In the simple greedy approach one task per host increases the number of hosts used and reduces the host utilization. As a result, the solution has higher energy costs and requires more hosts than HMIP. mPP allows resource sharing, thus fewer hosts are used than in the solution returned by the simple greedy approach. However, mPP cannot guarantee the solution is near the global optima because of the limitation of the packing strategy. Moreover, both the simple greedy approach and mPP do not consider the memory and license constraints, but this limitation is not shown in this simple example.

Chapter 7 Management in Dynamic Environments

The discussion so far has been in terms of *static conditions* for the applications and the cloud: a constant set of applications and user workload intensities, fixed application demands, and a fixed set of hosts. One goal of management of a cloud is to respond to a change in these conditions, with a new deployment if necessary. A host may fail, a new application may need to be deployed, and the parameters of existing applications may change. A new optimization may simply repeat the effort of the first one, but it is better to take into account the existing deployment and try to minimize the changes to be made. This makes the changes quicker and cheaper to install.

A common way to guarantee the performance subject to variations is to reserve specified resources and provision resources as long as the applications demand them. A critical downside of reserving resources is that the application only can use the particular resources, which limits resource sharing. In addition, it does not consider the effects of contention. Moreover, the deployment of new tasks into a Cloud can impact the performance of tasks already running on the machine. *Optimization with persistence* [11] is an effective way to provide online adaptive resource provisioning in terms of dynamic requirements, increasing resource utilization, limiting the impact on existing configurations, and providing dynamic robustness.

Figure 7.1 shows two deployment scenarios: (a) new application deployment (shown by the dark dashed arrow), and (b) redeployment (shown by the light dashed arrow) where all applications are optimally redeployed to adapt to changing conditions. The deployment optimization module computes the deployment plans and forwards them to a deployment engine (such as IBM Tivoli Provisioning Manager) which executes them. The optimization decisions are based on the state of the cloud which includes information about the applications and resources already allocated.

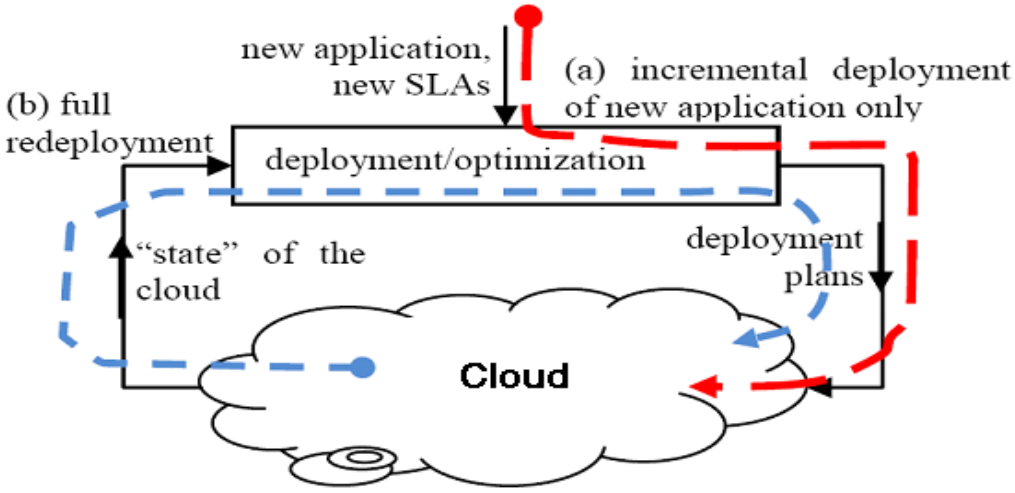


Figure 7.1 Deployment Scenarios

To enhance sharing, one node may host more than one application. In both the deployment and redeployment scenarios, (a) and (b) in Figure 7.1, the cloud uses a flexible licensing model. It owns a number of concurrent licenses (which means the owner can run a specified number of application instances at the same time) for each type of application. When that number is exceeded the cloud acquires additional pay-per-use licenses.

7.1 Re-Optimization with Persistence

A good re-optimization approach should increase resource sharing with persistence of existing placements, limiting the cost of changes. Brown et al. [11] stated that persistence can be achieved by using hard constraints to limit deviations from previous preferred values, or soft constraints to penalize the deviations. This implies that persistence in the optimization of task placement can be achieved by adding constraints on the flow rates or giving rewards/penalties on the arcs. This will help to control the changes to running applications when delivering new decisions in response to changing requirements.

7.1.1 Constraints on the Flow Rates

If an application arrives or parameters in the performance model are changed, some of the existing configurations to be preserved can be constrained with flow bounds, limiting the deviation of flow rates.

Constraint I. Constraints on the Flow Rates

Flow rates can be constrained with lower and upper bounds, which respectively guarantee the minimum and maximum execution demands to be assigned to the task from the host. When the upper and lower bounds are the same, this specifies the exact amount of the execution demands given to the task.

Constraints on the flow rates can be employed in the following ways:

- *Disable specific allocations*

If an allocation is not allowed, the arc can be disabled in the optimization by setting the upper bound of the flow rate at 0, At 0 (or equivalently, by removing the arc from the model).

- *Preserve existing configurations without changes*

If a running task must perform stably without any changes, its upper and lower flow bound are constrained at the current flow rate.

- *Reuse the allocation and allow the addition of extra workloads*

If a running task is needed to perform stably and to handle extra workloads, the upper bound can be set at infinity while the lower bound is set at the current flow rate.

- *Reuse the allocation but limit the resource utilization*

If an allocation can be reused but no more new workloads can be added, the lower bound can be set at 0 and the upper bound set at the current flow rate.

- *Reuse the allocation but limit the deviation of the performance*

If a running task can be performed with certain deviations, the upper and lower bounds are constrained with the maximum and minimum flow rates that describe the acceptable varying range.

Note that constraints on the flow rates are hard constraints, guaranteeing the size of the execution demands delivered to the allocation.

7.1.2 Reward/Penalize the Allocations

Another approach is to use the weights p_{arc} defined in Optimization Model III to apply a reward or penalty on the arcs. These control the priority of use of the allocation.

This approach can help to minimize the changes in the placements and preserve good configurations. For example, in order to drive the re-optimization to reuse the existing allocations, the arcs currently having no flows can be penalized, and the arcs that are in use can be rewarded. The value of the penalty and rewards can be determined by the cost of a new installation, risk of migration, and the importance of preserving some replicas. Unlike the hard constraints on flow rates, a reward or penalty is a soft constraint.

7.1.3 Constrain New Replicas for Some Specific Tasks

In dynamic environments, adding new replicas can help to give sound performance in response to the variation of conditions, but installing new deployments is associated with risks, increased operating costs and provisioning delays, and may change the system structure and parameters. In some cases, it is better to limit the creation of new replicas for some specific tasks in order to reduce the associated risks and costs.

The optimization approach provides this property by considering the requirements as constraints, shown as follows:

Constraint II. Constraints on Creating New Replicas for Specific Tasks

1. indicate each arc currently in use by setting a parameter $old\_arc_{ht} = 1$, otherwise 0.
2. for each task, the number of new replicas is

$$\Sigma_h (1 - old\_arc_{ht}) A_{ht} \qquad (37)$$

The new replicas here refer to the replicas to be installed, which include the new replicas created from duplication and migration and also newly arrival tasks.

3. when the number of new replicas of a task t is not allowed to exceed a limit (R_t), this condition can be described with

$$\Sigma_h (1- old\_arc_{ht}) A_{ht} \leq R_t \quad (38)$$

This constraint builds up a relationship that has the following characteristics.

- When the arc is already in use, $old\_arc_{ht} = 1$, the reuse of this allocation is not counted as a new replica. $(1- old\_arc_{ht}) A_{ht} = 0$
- When the arc is not in use, $old\_arc_{ht} = 0$, if $A_{ht} = 1$, which means it is a new replica. $(1- old\_arc_{ht}) A_{ht} = 1$
- When the arc is not in use $old\_arc_{ht} = 0$, if $A_{ht} = 0$, which means this allocation is not selected. $(1- old\_arc_{ht}) A_{ht} = 0$

For each task t, there is a cost C_R_t associated with each new replica. The value is determined by the cost of installation and maintenance of the new replica and the possible revenue lost due to the process of installation. This cost can be considered in the objective function described as below, in order to minimize the cost of changes.

$$C\_R_t(1- old\_arc_{ht}) A_{ht} \quad (39)$$

4. In some cases, for the task, the number of new replicas is expected to be below a certain limit, but this is not a hard constraint. The limit can be described with a slack (S_R_t), indicating that the extra changes are allowed if they are necessary.

$$\Sigma_h (1- old\_arc_{ht}) A_{ht} \leq R_t + S\_R_t \quad (40)$$

There is a penalty P_R_t associated with each extra replica. For each task, the total penalty can be calculated as below. This calculation can be considered in the objective function for minimizing the costs of extra replicas.

$$P\_R_t \cdot S\_R_t \quad (41)$$

7.1.4 Limit New Replicas to a Small Number

In order to improve the stability of the system in dynamic deployment management, changes on some running applications should be limited to a small range. For example, in each re-optimization update, a possible constraint is that at most 10% of running tasks may be migrated or given new replicas. To achieve this goal, a constraint can be included in the optimization as developed as below.

Constraint III. Constraints to Limit New Replicas to a Small Number

1. Reuse Σ_h (1- old_arc_{ht}) A_{ht} to indicate the number of new replicas of a task t, in the same manner as above.
2. Let ptg_T be the percentage of running tasks that could be changed, where T represents a set of running replicas. Assuming there are N elements in T, then the number of tasks allowed to be changed is the largest integer value that is no greater than ($ptg_T/100)N$, indicated by floor(($ptg_T/100),N$). Let $Slack\_T$ be the extra number of changes if needed, but these are not expected, since extra change violates the available range.
3. A constraint then can be created:

$$\Sigma_T \Sigma_h (1- old\_arc_{ht}) A_{ht} \leq (ptg_T/100)N + Slack\_T \quad (42)$$

4. $Slack\_T$ is included in the objective function where it is minimized.

This is still a MIP problem. The constraint guides the optimization to limit the scale of changes, offering system stability subject to dynamic changes.

7.1.5 Reduce the Number of New Hosts

Using a new host is associated with a start-up cost (R_h) to boot the computer. In some cases this cost should be taken into account in order to provide a more efficient adjustment in the face of dynamic changes. The optimization approach can provide this property by reducing the use of new hosts, as follows:

1. Indicate each host currently being used by setting a parameter old_S_h = 1, otherwise 0.
2. Account for the *Total Start-up Costs* calculated as below in the min-cost objective function.

   $$Total\ Start\text{-}up\ Costs = \Sigma_h\ (1 - old\_S_h)\ S_h\ R_h \qquad (43)$$

   In the Optimization Model III, when a host is used, S_h is constrained to be 1, otherwise 0. If the host is newly used (old_S_h = 0 and S_h =1), then the associated start-up cost R_h is added to the total costs. This guides the min-cost optimization to find a solution with a smaller number of new hosts.

7.2 Three Re-Optimization Strategies in Dynamic Environments

Three re-optimization strategies will be compared, one using full optimization in every period, and the other two using optimization with persistence:

- Full re-optimization: attempts to find the solution with the least cost, using the MIP model shown in Optimization Model III.
- Re-optimization with persistence: allows a new application to share resources with existing applications, but attempts to reduce the changes to existing

deployments. This strategy rewards the flows that are in use and penalizes the potential new flows.

The model for Re-Optimization with Persistence can be described as:

1. Reward the arcs that represent deployed tasks that are running. Set a penalty parameter p_{arc_ht} with a negative value, which indicates how important it is that the allocation should be preserved. A smaller p_{arc_ht} indicates that it is more important to preserve this allocation.
2. Penalize the arcs representing potential new deployments. p_{arc_ht} is assigned a positive value. The value of p_{arc_ht} is determined by the costs to install the new allocation. A greater penalty means larger costs.
3. The value of p_{arc_ht} is updated periodically to guide the re-optimization to choose new solutions.

- Simple Rule (re-optimization without sharing): This is a simple strategy used in practice. New deployments only can be added onto new hosts that are not being used. This strategy has no impact on existing deployments. This strategy imposes several constraints on the optimization model:
    1. It imposes a lower bound on each output arc that is in use equal to the current flow rate, labelled as [current flow rate, ∞, cost per flow rate]. This allows increasing loadings of the running tasks, but cannot reduce the loads, or terminate, or move the tasks.
    2. It places an upper flow bound of zero on the idle arcs ongoing from the running hosts, labelled as [0,0,0].

7.3 Experiments: Controlling the Scale of Changes

These experiments are conducted on the host pool described in Table 6.4. Each application has the structure shown in Figure 6.2, but with different parameters and requirements (chosen randomly). These ecommerce systems are to share a cloud. A 0.25 stress rate is used, and at most 80% of the CPU capacity can be used to avoid overloading. HMIP is used as the optimization algorithm and takes into account:

- Average response time in SLA
- Resource availability and requirements (CPU and Memory)
- License availability and license cost
- Power Consumption (execution and fixed costs)
- Effects of contentions

This experiment uses LQNS V5.1 as the performance model to conduct performance and sensitivity analysis and uses CPLEX v12.1 as the MIP optimization solver.

Three applications are running, which respectively are labelled as APP 17, APP 89 and APP 91. The information on the costs and resource utilizations of the running system is shown in Table 7.1, in which energy cost is the total of fixed and execution costs. A new application labelled APP 97 is added, which can share resources with the running applications. The “number of new replicas” stands for the replicas just installed in the re-optimization.

Table 7.1 Running Applications in the Cloud Infrastructure

# of applications	# of allocations	Total # of hosts	Total energy and license cost	Execution cost	Fixed cost
3	25	7	15.693	5.894	9.72

The evaluation measures the energy and license cost, the number of new allocations (including the allocations of new tasks and the migration of the old allocations), and the number of hosts in use.

The first experiment attempts to limit the number of replicas of some specific tasks (using the approach given in Section 7.1.3), and the next experiment tries to limit the changes to a small number (using the approach given in Section 0). These experiments will evaluate the feasibility of the approach and study the impact of the stress of the constraints.

7.3.1 Case Study on Controlling the Costs of New Replicas for Specific Tasks

This example is a simulation to evaluate the effectiveness of the re-optimization in terms of the number changes required relative to the cost of the changes. The number of new replicas (the replicas to be installed) per task is controlled in each re-optimization. One test is to let each task have at most one more free replica in the re-optimization (indicated by 1 rep per task), and the other allows each task to have two new free replicas (indicated by 2 rep per task). Each extra replica above the limitation is associated with an extra cost (set C_R_t=0, P_R_t = 1). These results are compared against full optimization without persistence constraints, shown in Table 7.2 and Figure 7.2.

Table 7.2 Re-Optimization with Limitations on the New Replicas per Task

4 applications	full optimization	2 rep per task	1 rep per task
Total energy and license cost	19.278	19.590	19.606
Execution costs	7.558	7.440	7.456
Fixed costs	11.72	12.15	12.15
Total number of replicas	36	36	34
number of new replicas (including newly arrived tasks)	34	29	25
Total # of hosts	9	9	9
# of new hosts used in the re-optimization	6	4	3

This comparison shows that the solution given by the full optimization has the smallest costs; the costs of the 2 rep per task and the 1 rep per task are similar, but 2 rep per task costs slightly less. The number of new replicas and the number of new active hosts used in the full optimization is larger than the other two. Because it has the most limiting constraints, the 1 rep per task has the smallest changes.

Figure 7.2 shows the number of new replicas per task required by each algorithm, when handling re-optimization to accommodate the newly arrived application APP 97.

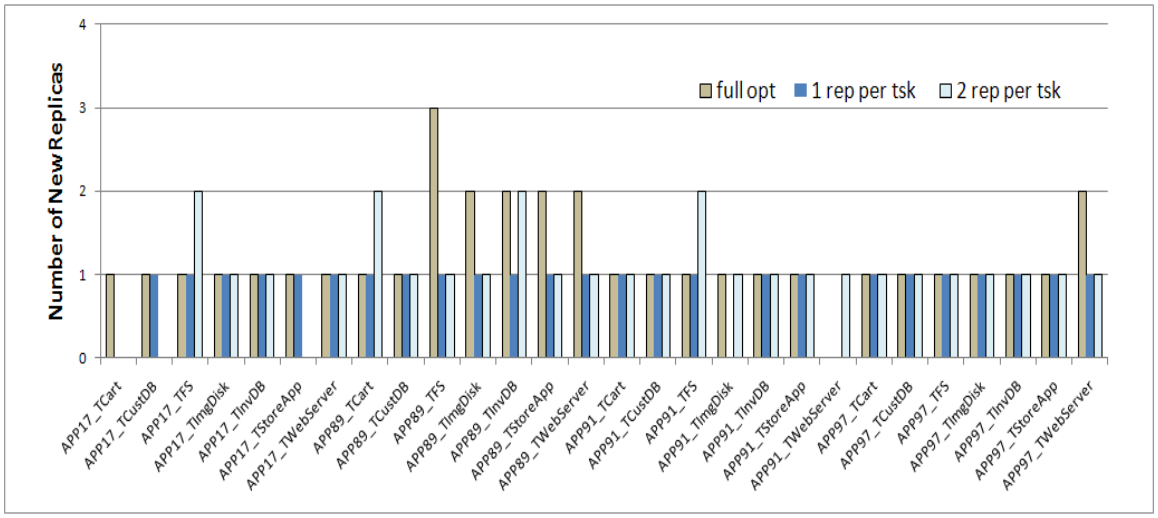


Figure 7.2 the number of new replicas of each task created in re-optimization (constraint on new replicas per task)

The results show that the optimization with persistence is effective in controlling the number of new replicas of each task in the re-optimization. In 1 rep per task and 2 rep per task, the number of new replicas per task can be effectively controlled below one and two respectively. Without this control, for some tasks, such as APP 89_TFS, the number of new replicas reaches three.

7.3.2 Case Study of Limiting New Replicas to a Small Value

This experiment is conducted on the same prototype as in the previous experiment which controls the replications per task. The constraint on the re-optimization to ensure persistence is changed. This experiment evaluates the effectiveness of limiting the number of running tasks that are changed to below 5% (indicated by ptg 5%), 10% (indicated by ptg 10%) and 20% (indicated by ptg 20%). Note that changes count the number of new replicas of running tasks, including the new replicas and migrated tasks; newly arrived tasks and loading changes are excluded from the count.

According to the results given in Table 7.1, limiting changes to 5% means that at most one new replica of a running task can be added, two for 10%, and five for 20%.

Table 7.3 Re-Optimization with Limitations on the Percentage of Changes

4 applications	**full optimization**	**Ptg 20%**	**Ptg 10%**	**Ptg 5%**
Total energy and license cost	19.278	19.607	20.132	20.358
Execution costs	7.558	7.457	7.172	7.398
Fixed costs	11.72	12.150	12.96	12.96
Total number of replicas	36	36	34	33
number of new replicas (including newly arrived tasks)	34	15	10	8
Total # of hosts	9	9	9	9
# of new hosts used in the re-optimization	6	2	2	2

The results in Table 7.3 show the effects of limiting the percentage of changes. Figure 7.3 shows the effects of controlling the percentage of changes. At 5%, the number of replicas of the running tasks is controlled at 1 (APP89_TWebServer), and in 10% the number of new replicas is successfully controlled within 2 (which are respectively APP89_TFS and APP89_InvDB). The number of new replicas at 20% is controlled at 5, meeting the constraint.

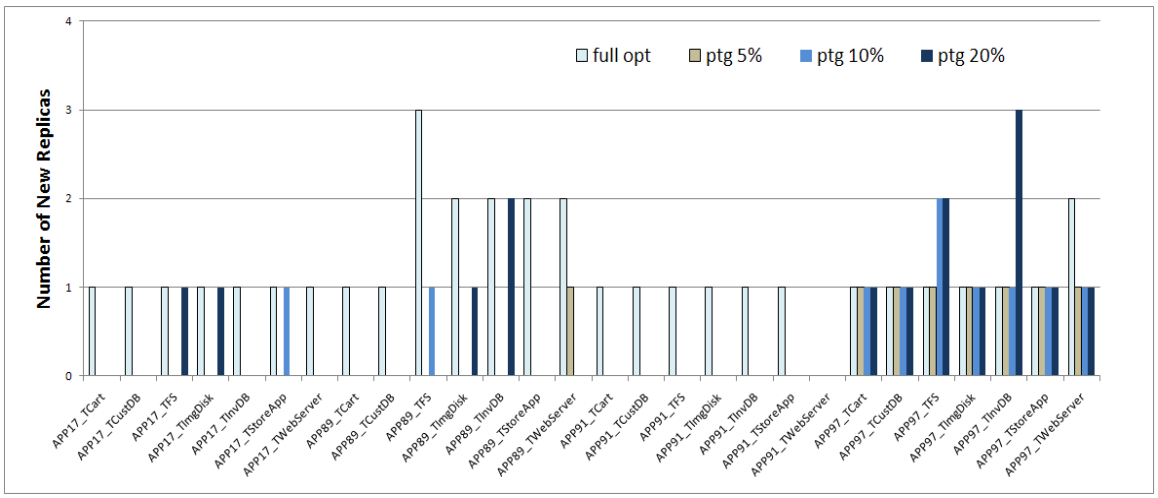


Figure 7.3 the number of new replicas of each task created in re-optimization (Ptg Control)

7.4 Summary of Optimization with Persistence

The experimental results demonstrate the effectiveness of imposing persistence in re-optimization, limiting changes to specific tasks and the scale of changes. These approaches can satisfy many dynamic requirements while accounting for the costs/risks of changes, and are effective in offering high-quality solutions with persistence.

These approaches can be combined to address complex problems with multiple simultaneous goals. For example, they could consider the scale of changes and the changes per task at the same time. In addition, constraints on the flow rates can be added to limit the loading of some specific tasks, while using penalty/rewards to guide the re-optimization.

Chapter 8 Case Study

A set of simulations was used to evaluate the scalability of the optimization approaches and the stability of management in responding to various dynamic changes.

The algorithms HP, Pure MIP and HMIP were evaluated on 12 deployment problems, the number of applications scaling from 1 to 10, and the stress rate ranging from 0.25 to 0.9. The algorithms in these tests account for contention in the response time calculation.

The case study of the stability of management includes control in several difficult dynamic environments, which include

- Varying Workloads
- Addition and removal of applications
- Failure and repair of host machines

8.1 Experimental Environment

The cloud environment in which the test applications are deployed is the host pool described in Table 6.4. Each application has the structure shown in Figure 6.2, each with different performance parameters and requirements (chosen randomly). The costs considered in the optimization include the execution cost, fixed cost and license cost.

All experiments expect to use at most 80% of the CPU capacity in order to avoid overloading. These experiments take into account:

- Average response time in SLA
- Resource availability and requirements (CPU and Memory)
- License availability and license cost
- Power Consumption (execution and fixed costs)
- Effects of contention

The experiments use LQNS V5.1 as the performance model solver to conduct performance and sensitivity analysis and CPLEX v12.1 as the MIP optimization solver. The implementation of the algorithm is coded with Java, running on JRE 1.6 on an Intel 2.4GHz Dual machine with 3GB of RAM. Since it is a MIP problem, the LP solver is not used here.

8.2 Evaluate the Scalability of the Three Algorithms

A cloud can host many applications with separate service contracts. This necessitates an approach that offers global management for large scale systems. This experiment demonstrates the scalability of the approaches developed here, and evaluates the performance of Pure MIP, HMIP and Heuristic Packing in a realistic deployment that considers the effects of contention.

The algorithms will be evaluated with practical deployment problems including consideration of contention issues. A system with 1 to 10 versions of the application template model in Figure 6.2 (each with different performance parameters and requirements) is used for deployment. The number of services increases from 10 to 110,

and the number of original tasks (without replicas) increases from 7 to 70. The results of the optimization are shown in Table 8.1.

Table 8.1 Evaluation of Pure MIP, HP and HMIP with Contention

Stress rate	**Very High (0.9±0.05)**			**High (0.7±0.05)**			**Medium (0.5±0.05)**			**Low (0.25±0.05)**		
1 app	HP	Pure MIP	HMIP	HP	Pure MIP	HMIP	HP	Pure MIP	HMIP	HP	Pure MIP	HMIP
# of iterations	1	2	1	2	1	1	1	1	1	2	2	1
Time on MIP**(sec)**	0.01	0.22	0.11	0.01	0.16	0.125	0.01	0.282	0.11	0.01	0.25	0.109
Time on LQNS**(sec)**	0.537	0.77	0.578	1.11	0.5	0.531	0.531	0.406	0.56	0.719	0.672	0.391
Time on LEndStep**(sec)**	7.438	5.94	6.766	11.84	6.3	7.922	7.469	4.344	7.08	4.234	4.39	4.578
Total Solution Time**(sec)**	7.969	6.92	7.545	12.95	6.95	8.578	8	5.032	7.83	4.953	5.328	5.078
Total Solution Time without LEndStep	0.547	0.99	0.688	1.12	0.66	0.656	0.541	0.688	0.67	0.729	0.922	0.5
Total Energy and License Cost	11.94	11.53	11.92	12.03	11.53	11.63	11.65	11.43	11.64	11.41	11.39	11.28
# of replicas (task)	13	12	13	14	12	13	13	12	13	13	12	12
# of services	20	19	21	22	19	21	20	19	20	20	19	19
Costs saved by sensitivity	0.001	5.00E-04	0.004	0.04	0.01	0.01	0.004	0.01	0.003	0.02	0.005	0.003
# of variables in MIP	-	127	97	-	157	97	-	217	97	-	382	82
# of hosts	6	6	6	6	6	6	6	6	6	6	6	6

Stress rate	**Very High (0.9±0.05)**			**High (0.7±0.05)**			**Medium (0.5±0.05)**			**Low (0.25±0.05)**		
5 app	HP	Pure MIP	HMIP	HP	Pure MIP	HMIP	HP	Pure MIP	HMIP	HP	Pure MIP	HMIP
# of iterations	3	8	2	6	6	4	8	3	1	7	1	4
Time on MIP**(sec)**	0.016	11.54	0.468	0.016	4.39	0.923	0.047	2	0.39	0.142	0.375	0.672
Time on LQNS**(sec)**	15.797	29.41	8.048	23.156	21.751	14.123	31.37	11.266	4.375	24.671	2.703	11.874
Time on LEndStep**(sec)**	131.53	124.8	151.03	171.76	99.794	147.7	180.7	104.95	119.61	139.7	56.7	109.41
Total Solution Time**(sec)**	157.15	165.9	159.56	194.94	125.86	162.78	212.1	118.25	124.38	164.62	59.7	122.02
Total Solution Time without LEndStep	15.813	40.95	8.516	23.172	26.141	15.046	31.417	13.266	4.765	24.813	3.078	12.546
Total Energy and License Cost	23.96	23.47	23.479	23.64	23.91	23.14	24.54	23.16	22.98	23.93	22.34	23.137
# of replicas (task)	42	45	45	47	44	45	48	44	44	45	43	45
# of services	67	71	73	76	71	72	77	69	71	71	69	72
Costs saved by sensitivity	0.06	0.08	0.022	0.1	0.04	0.03	0.11	0.018	0.006	0.04	0.008	0.0145
# of variables in MIP	-	887	887	-	1171	887	-	1597	816	-	2804	674
# of hosts	12	11	11	11	11	11	12	11	11	12	11	11

Stress rate	**Very High (0.9±0.05)**			**High (0.7±0.05)**			**Medium (0.5±0.05)**			**Low (0.25±0.05)**		
10 app	HP	Pure MIP	HMIP	HP	Pure MIP	HMIP	HP	Pure MIP	HMIP	HP	Pure MIP	HMIP
# of iterations	4	11	14	6	11	12	9	3	6	14	6	4
Time on MIP**(sec)**	0.077	4.468	6.921	0.046	4.531	5.175	0.076	4.063	3.204	0.234	7.5	1.609
Time on LQNS**(sec)**	56.86	154.5	200.25	94.718	129.91	156.05	139.89	43.09	81.859	198.69	73.048	42.406
Time on LEndStep**(sec)**	652.44	861.7	689.72	1084.4	494.67	482.27	933.52	540.6	494.39	946.2	525.31	401.02
Total Solution Time**(sec)**	709.52	1020	896.97	1179.2	629.16	643.58	1073.5	587.8	579.5	1145.2	605.9	445.09
Total Solution Time without LEndStep	56.937	158.968	207.171	94.764	134.441	161.225	139.966	47.153	85.063	198.924	80.548	44.015
Total Energy and License Cost	34.09	33.94	33.92	35.379	33.26	33.299	34.91	33.51	33.34	35.34	33.24	33.23
# of replicas (task)	67	85	83	86	83	83	86	81	81	84	84	80
# of services	105	137	133	137	131	131	135	129	130	135	134	128
Costs saved by sensitivity	0.036	0.07	0.027	0.008	0.019	0.026	0.03	0.04	0.019	0.003	0.032	0.012
# of variables in MIP	-	2467	2467	-	3313	2608	-	4582	2749	-	8107	2326
# of hosts	16	16	16	17	16	15	17	16	16	16	14	14

This evaluation shows that:

- The solution is very similar across low and high-stress problems, in the objective function value, the number of hosts used, and the number of replicas of tasks and services deployed. There does not seem to be any advantage in beginning with a lot of excess resources, in terms of being able to find a better solution.
- The LEndStep takes most of the time and gives only a small improvement in all the cases shown here, so it probably is not worthwhile in practice. In LEndStep, the calculation of LP is very fast, but the sensitivity analysis is time consuming because of a problem in LQNS, expected to be repaired soon. Nonetheless it is useful for comparing small differences between cases, since without it, there is a small effectively random jitter on the solution cost, of a few percent.
- There is no consistent trend in solution time, between low and high stress cases. If we adopt two minutes as a "maximum practical optimization time", then the approach without LEndStep is "practical" up 10 apps in most cases.
- Increasing the size of the host pool (to give a low stress rate) does not increase the total solution time. On the contrary, because a low stress rate allows for providing the required performance with fewer replicas, reducing the complexity of the performance model, the performance model can be solved with greater efficiency.

- For a small scale problem that has tens of tasks, the solution time is in a practical range for computing a deployment, as changes to deployment take on the order of minutes even for just a few machines. When a system scales up, the computation time is increased. In a whole optimization process, about 75~90% of the solution time is used on sensitivity analysis. And in the optimization loop over 70% of the solution time is used by the performance model for computing the contention delays at different layers and components. Over 85% of the total solution time is used by LQNS, which is the bottleneck of this approach.
- The efficiency of solving a MIP problem is not a critical issue in the optimization accounting for contentions. Though Bin-Packing can solve a MIP heuristically with the greatest efficiency, this had a limited effect on the overall solution efficiency, because the contention calculation in the performance model is much more time-consuming. Moreover, the costs of the solutions returned by the heuristic packing algorithm in most cases are the highest, since it cannot ensure the quality of optimization,
- Comparison of HMIP and Pure MIP: The objective costs given by both algorithms are close. MIP appears to be more effective for the deployment problem with 5 or fewer applications, while HMIP is faster for problems with 10 applications.

Based on these experimental results, our approach currently is able to handle the deployment problem for service systems with about 50 heterogeneous tasks. The complexity of the performance model has a significant impact on the optimization

efficiency. The LQNS developers expect to increase the speed of the software substantially in the near future.

8.3 Evaluation of the Stability of Management in Dynamic Environments

This section evaluates the effect of management in dynamic environments. The algorithms are used to handle several dynamic scenarios, including adaptive regulation of dynamic workloads, re-optimization for addition and removal of applications, and the failure and repair of host machines.

In the test, we define a "step" as a time interval for re-optimization (which could be global optimization, simple rules, or optimization with persistence) to update decisions, and define a "period" as a series of steps with one global optimization and possibly some non-global optimizations. A period may include one or several steps. For the algorithms of full optimization, optimization with persistence and simple rules please refer to Section 7.2.

In this test, the optimization with persistence sets $p_{arc\_ht} = -\alpha_{ht}$, rewarding existing deployments on the basis of the loads of replicas. A replica with larger execution demands gains a greater reward, driving the re-optimization to reserve the replica with a priority.

The experiments study the quality of full optimization, optimization with persistence, and optimization with simple rules by measuring the energy and license cost of the solutions, the degree of change and the number of replicas and hosts used, and study the effects of varying the period for full optimization.

In the figures below, we use this notation to represent the algorithms:

- *Full opt*: full optimization
- *Px*: take full optimization every *x* steps, for example p3 means full optimization will be conducted every 3 steps.
- *Pers*: use optimization with persistence in between full optimizations
- *Rules*: use simple rule in between optimizations, ensuring current replicas are unaffected by changes.

Let "new replicas" represent the replicas newly installed, and "new hosts" represent the hosts activated for use. The "percentage of new hosts" or the "percentage of new replicas" in the figures correspondingly mean the fraction of new hosts among all active hosts (including the new active hosts) and the fraction of the new replicas in all replicas in use (including new replicas just created), calculated as,

$$Percentage\ of\ New\ Hosts = \#h_{new\_t} / \#h_{total\_t} \times 100 \quad (44)$$

$$Percentage\ of\ New\ Replicas = \#R_{new\_t} / \#R_{total\_t} \times 100 \quad (45)$$

$\# h_{new\_t}$ is the number of the new hosts, i.e. that are used in period t but not used in period t-1,

$\#h_{total\_t}$ is the total number of hosts used in period t,

$\#R_{new\_t}$ is the number of new replicas created in period t,

$\#R_{total\_t}$ is the total number of replicas used in period t.

To evaluate the quality of the persistence in this experiment, two types of costs are measured. One is the energy (Section 6.1) and license cost, and the other is the *"start-up costs"* which measures the costs of using new hosts or new replicas. We define 1 unit

start-up cost for a new host (boot the computer) and 3 units start-up cost for a new replica (install VM and image) to quantify how much start-up costs are totally used.

In the tests arbitrary changes are permitted. Constraints as described Section 7.1.3 and Section 0 to limit the change with global optimization are not used here.

8.3.1 Dynamic Case I: Varying Workloads

With a change in the number of users, the resource requirement of each task is changed. This experiment applies optimization to find the minimum energy and license cost to ensure the maximum latency is not exceeded.

We assume that the request arrivals exhibit time-of-day variations typical of enterprise workloads, so the number of arrivals may change quite significantly during a one day period. The workload used in our experiments loosely resembles the behaviour found in the log files from the Soccer World Cup 1998 Web site [4] and is shown by the bars in the figures below. We use this workload pattern because it has significant workload variations with large dynamic spikes. In the performance model used in this experiment, the number of users (N_c) varies between 80~650 and the average think time (Z_c) is set at 1400ms. Optimization with persistence was applied at 20-minute intervals, with a response time specification (ignoring network delay) of RTSLA = 35ms.

1. *Response Time*

Figure 8.1 shows that throughout the test all control approaches are able to maintain the response time below the required value. Full optimization gives a stable response time, controlling the variation within 20%. In certain steps, some response times, especially some of those that are given by optimization with persistence (Pers) or with

simple rules (Rule), are much shorter than needed (with a cost penalty), since the persistence mechanism requires the optimizer to reuse existing configurations, resulting in resource over allocation. A short period between full optimizations can help to bring the configuration up to date, giving a response time closer to the SLA.

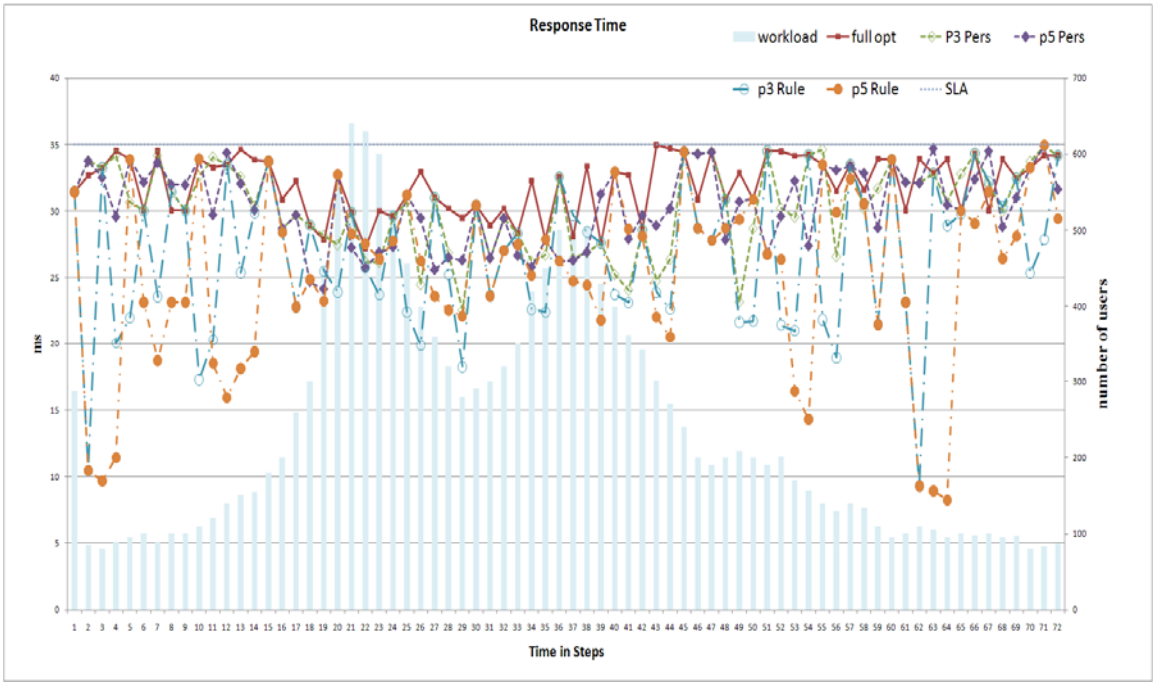


Figure 8.1 Response Time of the Application with Varying Workloads

Table 8.2 Average Response Time per Step (Varying Workloads)

	Full opt	P3 Pers	P5 Pers	P3 Rule	P5 Rule	SLA
Average RT	32.002	30.33	30.39	26.67	25.30	34.98

2. Capital and Start-up Costs

Figure 8.2 shows that the total energy and license cost returned by the full optimization are proportional to the change of workloads, and are the lowest. This cost returned by the simple rules is larger than other control approaches because sharing is not

allowed in the solution. Simple rules can isolate the environment, but they require more resources to handle the newly arrived workloads. Table 8.3 shows that the solution returned by the full optimization is with the largest start-up costs, about 2.5~3 times higher than the other approaches.

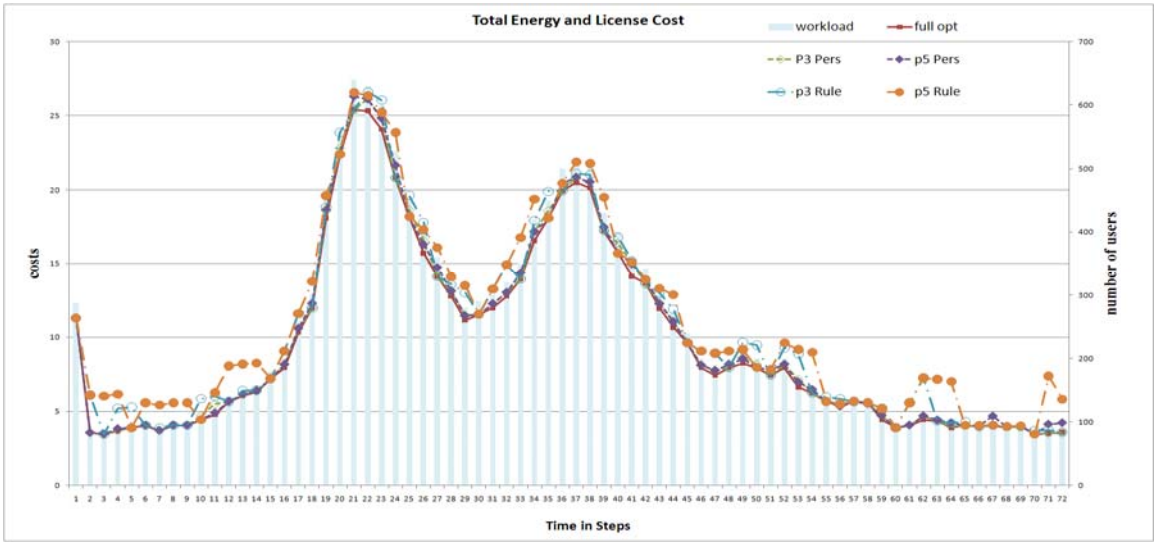


Figure 8.2 Total Energy and License Cost Subject to Varying Workloads

Table 8.3 Average Cost per Step (Varying Workloads)	Full opt	P3 Pers	P5 Pers	P3 Rule	P5 Rule
Total Energy and License Cost	9.70	9.88	9.93	10.41	10.95
Start-up costs on New Hosts	2.25	0.95	0.79	1.25	0.98
Start-up costs on new replicas	25.71	10.25	8.17	11.63	8.13
Start-up costs on new hosts and new replicas	27.96	11.19	8.96	12.88	9.11

3. Hosts in Use and the Percentage of New Hosts

Figure 8.3 and Figure 8.4 respectively illustrate the total number of hosts returned by the algorithms and the percentage of new hosts being activated to perform tasks. Figure 8.3 shows the resulting time-variation of hosts used, ranging from 2 to 13 subject to the Varying Workloads. It demonstrates that the algorithms are effective for providing

adaptive management for varying workloads with dynamic resource provisioning. The variation of the number of active hosts corresponds to the variation of costs shown above. A higher frequency of full optimization can give a more up-to-date adjustment.

However, the economical solution sought by full optimization is associated with great variation in the selection of hosts. In Figure 8.4 the percentage of new hosts used by full optimization ranges from 20% to 100%. For optimization with persistence (Pers), the number of new hosts can be controlled below 20% most of the time; and using simple rules, the percentage is further reduced and could be 0 most of the time.

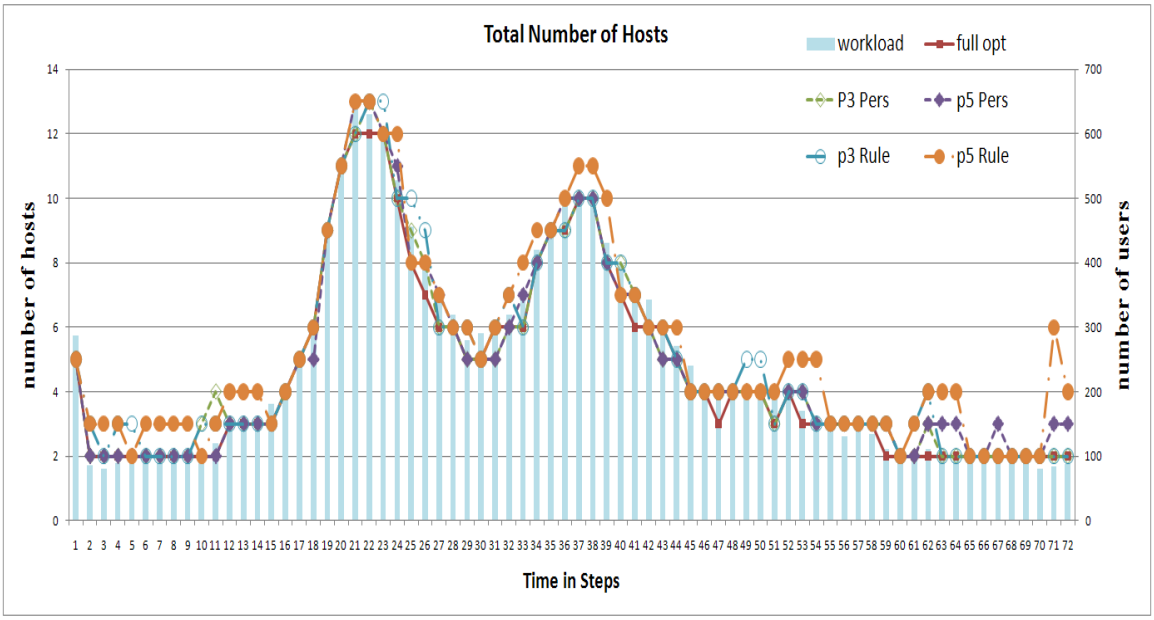


Figure 8.3 Total Number of Hosts subject to Varying Workloads

Table 8.4 Average Active Hosts per Step (Varying Workloads)

	Full opt	P3 Pers	P5 Pers	P3 Rule	P5 Rule
Average Active Hosts	4.63	4.76	4.83	4.94	5.32

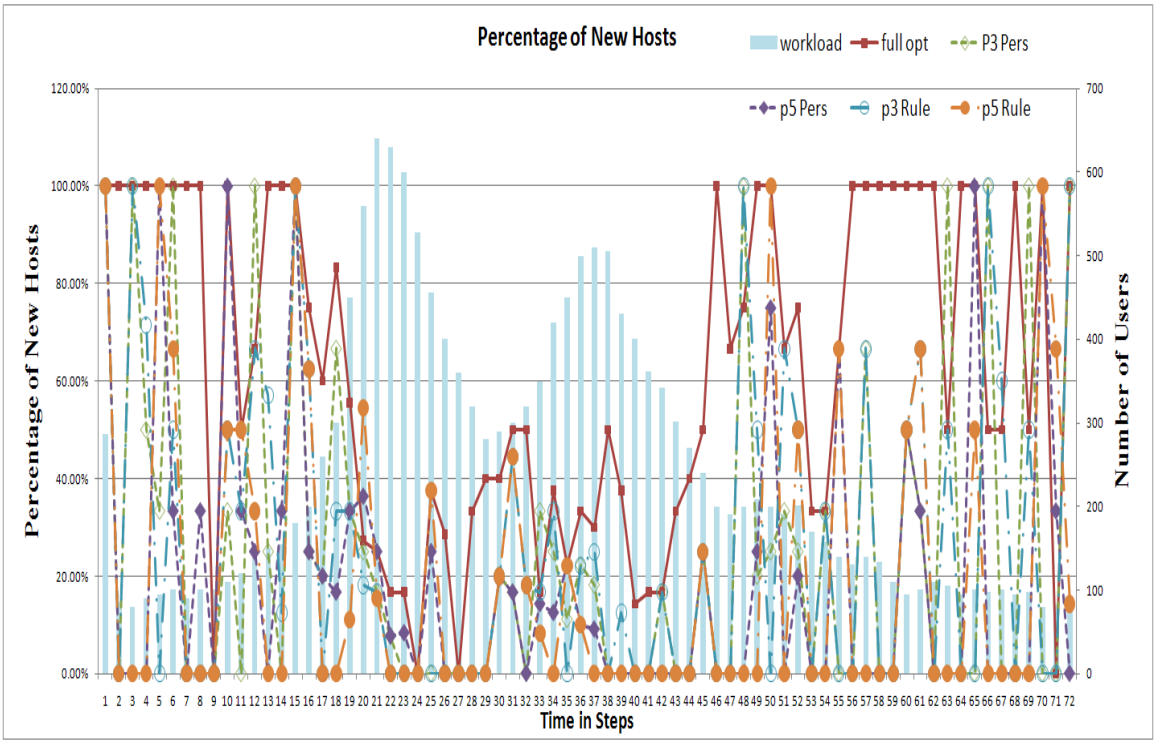


Figure 8.4 the Percentage of New Hosts in Use subject to Varying Workloads

Table 8.5 Average Percentage of New Hosts per Step (Varying Workloads)

	Full opt	P3 Pers	P5 Pers	P3 Rule	P5 Rule
Average Ptg of New Hosts	62.09%	25.15%	19.29%	26.14%	19.35%

4. Replicas in the System and the Percentage of New Replicas

Figure 8.5 and Figure 8.6 respectively illustrate the number of replicas in use and the degree of change at each step. Full optimization returns solutions using the smallest number of replicas; optimization with persistence (Pers) requires a few more, and the simple rules require the most. Simple rules use more replicas than other solutions because they are not allowed to create new replicas to offer workload aggregation, which combines the running and existing workloads, replacing the existing placements. In simple rules existing placements must be preserved and newly arrived workloads only

can be placed in new replicas on new machines. The number of replicas thus is increased, more than other approaches that are allowed to create large-size replicas to take the place of the existing placements.

To provide the most economical solution, the full optimization solution consists of 60% new replicas, and in some cases consists entirely of new replicas. A large number of changes of replicas increases the risk of instability and the operating costs for changes. In the same problem, optimization with persistence (Pers) and simple rules can keep the percentage of new replicas below 30 %.

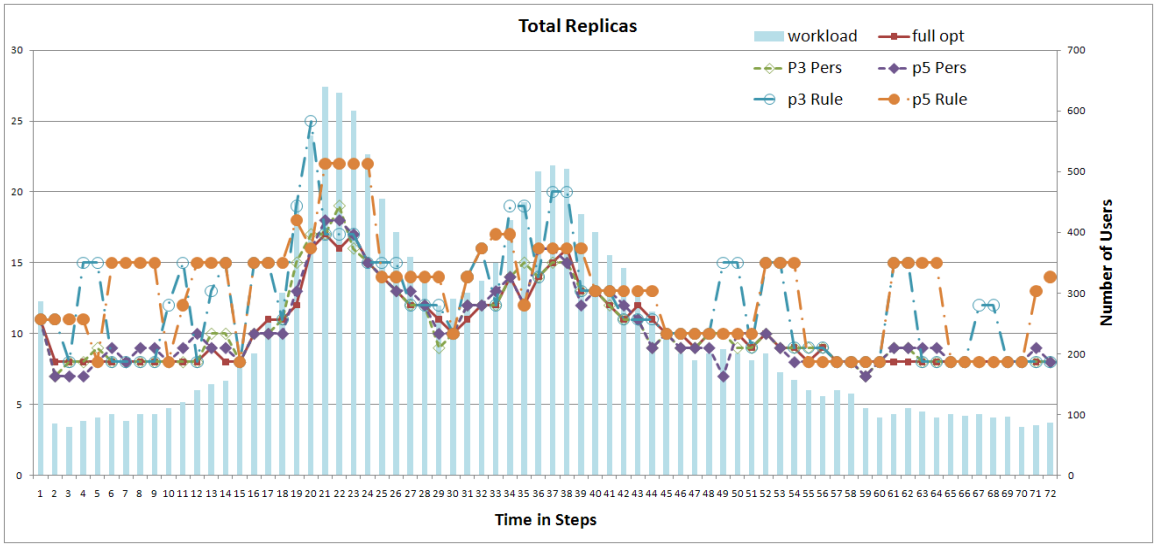


Figure 8.5 the Total Number of Replicas in Use subject to Varying Workloads

Table 8.6 Average Number of Replicas per Step (Varying Workloads)

	Full opt	P3 Pers	P5 Pers	P3 Rule	P5 Rule
Average Number of Replicas	10.28	10.36	10.36	12.25	12.99

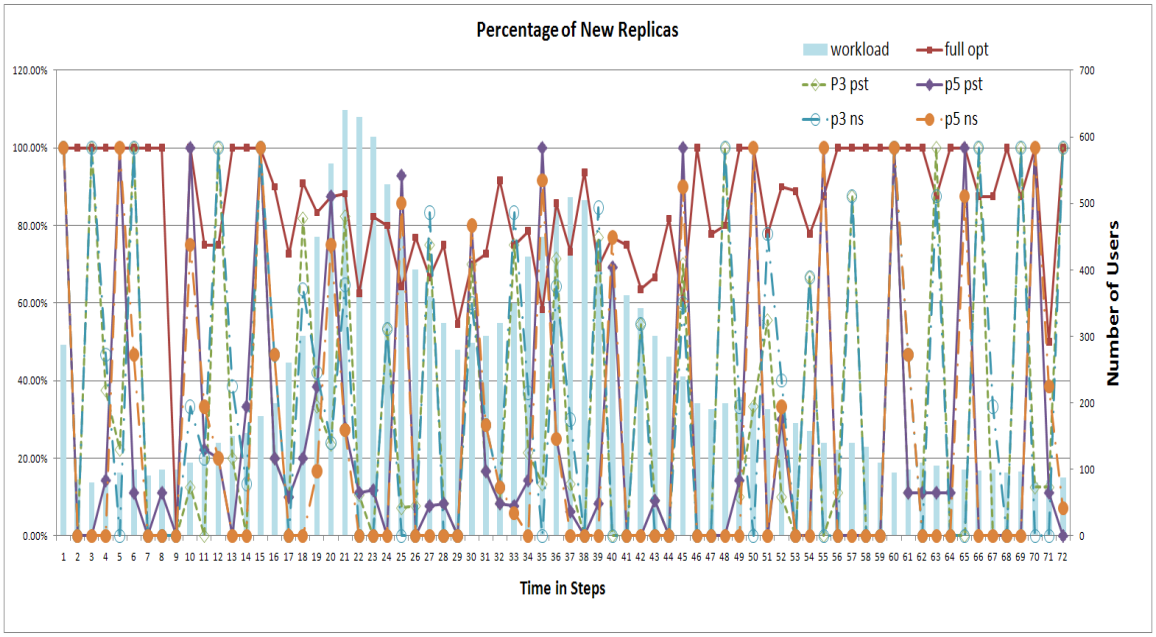


Figure 8.6 the Percentage of New Replicas subject to Varying Workloads

Table 8.7 Average Percentage of New Replicas per Step (Varying Workloads)

	Full opt	P3 Pers	P5 Pers	P3 Rule	P5 Rule
Average Ptg of New Replicas	84.68%	33.70%	26.28%	34.96%	24.31%

5. Summary : Management of Varying Workloads

As a summary for this evaluation, full optimization gives the most stable response time, and the solution is the most economical among these approaches, but in each step full optimization creates many new replicas, which increases the risks associated with changes, cost of operations and provisioning delays (which are not considered here). Simple rules give the fewest changes, but require the use of more hosts than other approaches, since simple rules do not allow new workloads to share hosts with the running tasks. Optimization with persistence (Pers) helps to constrain the number of changes, giving the required persistence of existing placements, and it allows some changes to the running tasks, allowing increased resource sharing.

This evaluation shows that a short period between full optimizations can help to quickly regulate the configurations, reducing costs due to over-allocated resources; however, a short period means an increase in the cost of changes.

8.3.2 Dynamic Case II: Host Failures and Repairs

This test evaluates the performance of the algorithms in handling host failures and repairs. In this test the initial stress rate is 0.25. Five applications with static workloads require guaranteed multi-class response time; 25% of running hosts fail and are removed in each step. This is a ridiculously high failure rate, but serves to underline the properties of the adaptive decision algorithm. New hosts are added into the host pool when the stress rate reaches 0.75. Figure 8.7 shows the variation of the size of the host pool. Because the hosts selected by each algorithm in a step might be different, the failed and remaining hosts may not be the same for each evaluation.

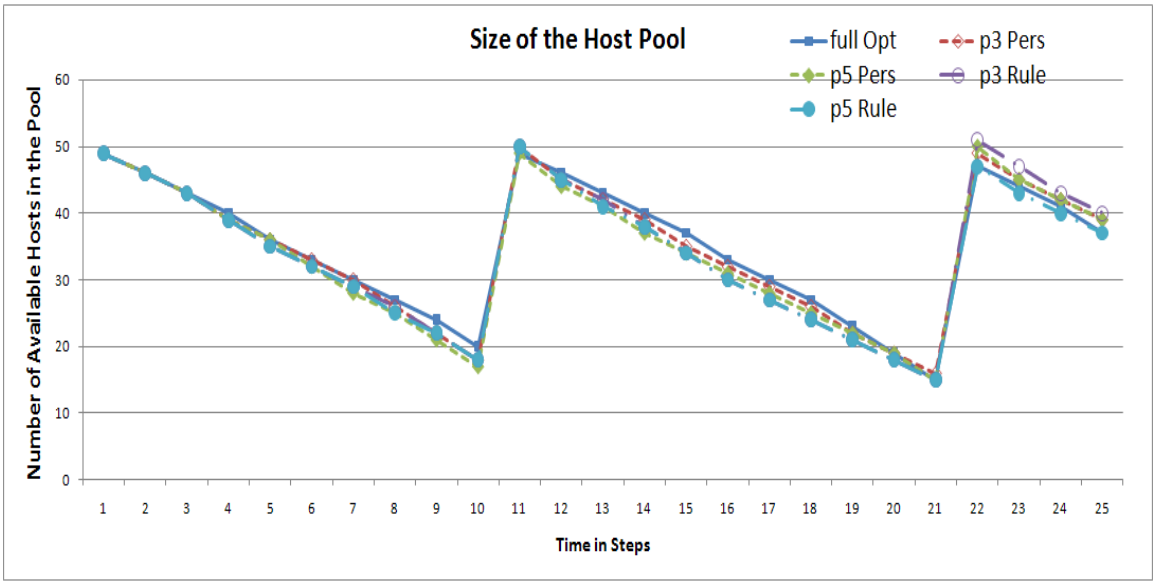


Figure 8.7 the Variation of the Size of the Host Pool in Host Failures and Repairs Environments

1. Costs Given By Different Control Approaches

Figure 8.8 illustrates the total energy and license cost given by different algorithms. It shows that full optimization gives the solution with the least cost, in most steps. For each algorithm, the periodic run of the full optimization reduces costs significantly, so full optimization is effective for reducing the costs subject to hosts failures and repairs and can save about 10~20% of the cost in comparison to other approaches. Because at each step the available hosts in the pool are not identical for each algorithm, the results given by the full optimization may not always be the best.

In most steps, optimization with persistence (Pers) performs better than simple rules when they are running with the same periods. This shows that allowing resource sharing helps to improve the quality of optimization. For either optimization with persistence (Pers) or simple rules, a short period returns lower costs than a long period most of the time.

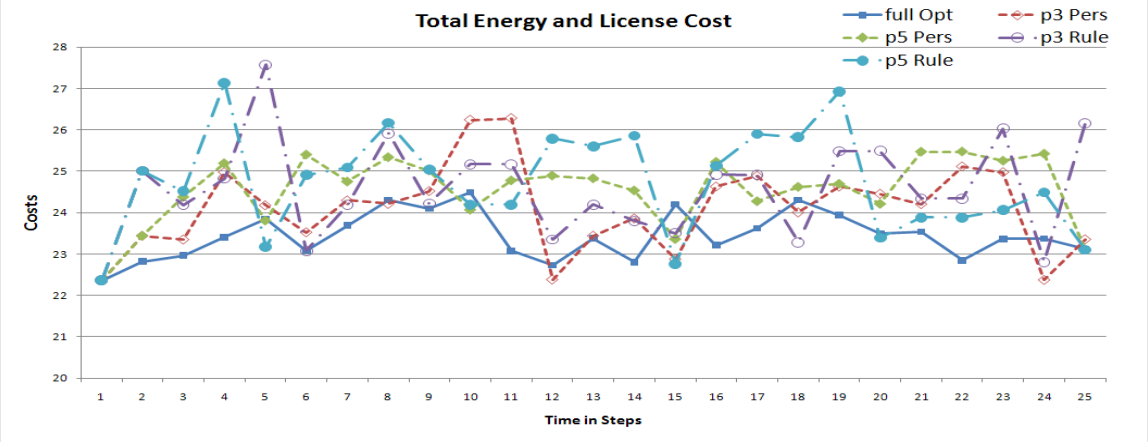


Figure 8.8 the Total Energy and License Cost Required by Each Approaches subject to Host Failures and Repairs

Table 8.8 Average Costs per Step (Host Failures and Repairs)

	Full opt	P3 Pers	P5 Pers	P3 Rule	P5 Rule
Total Energy and License Cost	23.44	24.10	24.56	24.57	24.74
Start-up costs on new hosts	4.76	4.16	3.84	4.32	3.68
Start-up costs on new replicas	118.56	71.4	55.56	71.28	58.92
Start-up costs on new hosts and new replicas	123.32	75.56	59.4	75.6	62.6

2. Replicas Required and Persistence

Figure 8.9 evaluates solution persistence by measuring the change in the number of replicas. It shows that most of the time the number of replicas returned by simple rules is slightly larger than returned by full optimization and optimization with persistence (Pers). This is again because of the constraints in simple rules, which preserve the remaining replicas and create new replicas to accommodate the affected workloads in the failed hosts; in the other solutions unaffected replicas can be aggregated with the affected tasks as a new replica is deployed in another machine.

In the comparison between the percentages of new replicas (Figure 8.10), it can be seen that in every step around 90% of the replicas are newly created by full optimization. Because of the effects of the persistence mechanisms, in each step the percentages of new replicas created are in the range of 20~40% for the simple rules or for the optimization with persistence (Pers). According to the average percentage of new replicas shown in Table 8.10, it can be seen that a long period gives fewer changes than a short period (about 8~10%). This is the same for both optimization with persistence (Pers) and simple rules. Though a short period helps to keep the configurations up-to-date, large scale changes may destroy the system stability. The effects of Pers in offering persistence is

similar to simple rules, with only 1~2% difference. The period has more impact on persistence than the algorithms.

This study implies that optimization with mechanisms to offer persistence are effective in providing robust management subject to hosts failing and being repaired. In particular the optimization with persistence (Pers) has the ability to limit changes to existing replicas, stabilizing the system structure. However the percentage of new replicas is quite high in all cases. Full optimization can reduce costs for execution, but it is associated with increased changes, increasing the associated risks and costs.

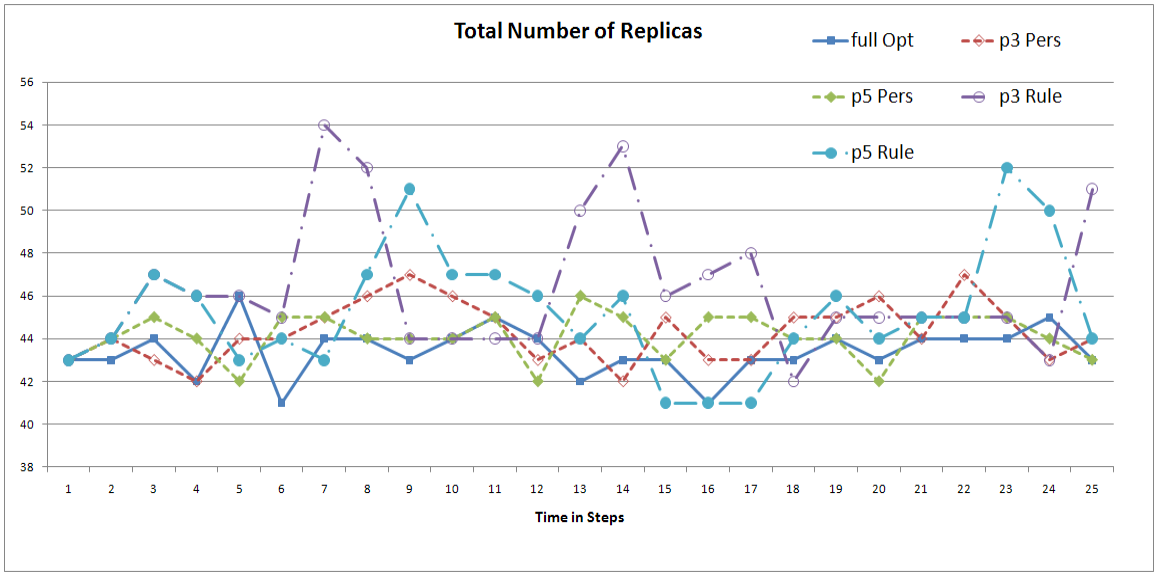


Figure 8.9 the Total Number of Replicas in the Host Failures and Repairs Environment

Table 8.9 Average Number of Replicas per Step (Host Failures and Repairs)

	Full opt	P3 Pers	P5 Pers	P3 Rule	P5 Rule
Average Number of Replicas	43.4	44.32	44.12	46.32	45.24

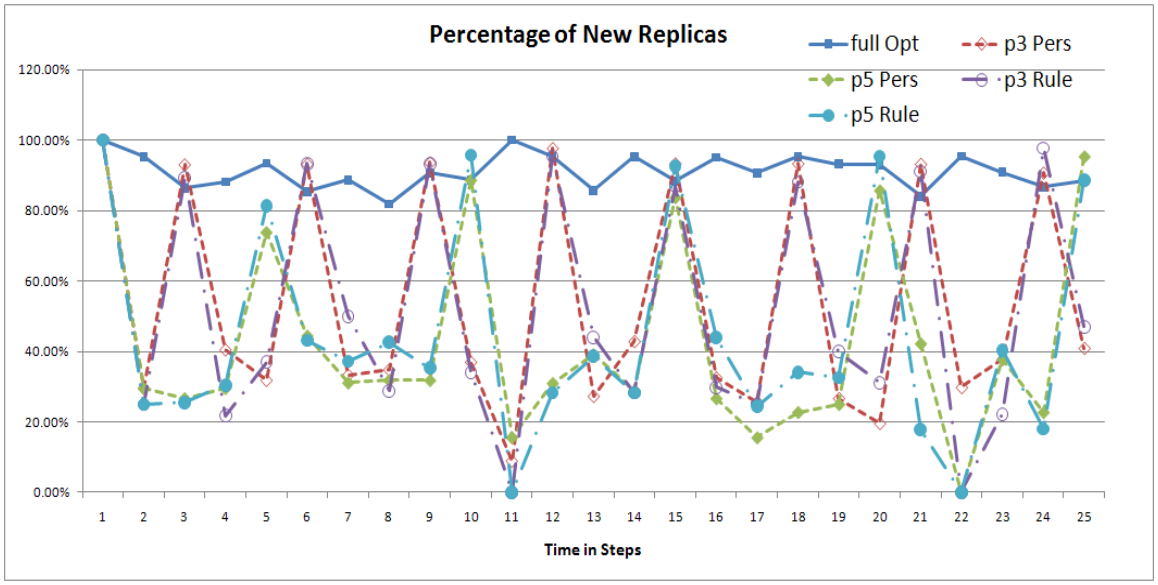


Figure 8.10 the Percentage of New Replicas in the Host Failures and Repairs Environments

Table 8.10 Average Percentage of New Replicas per Step (Host Failures and Repairs)

	Full opt	P3 Pers	P5 Pers	P3 Rule	P5 Rule
Average Percentage of New Replicas	91.05%	53.87%	42.38%	51.97%	43.98%

3. Active Hosts and Persistence

Figure 8.11 gives the number of hosts in use subject to the hosts failing or being repaired. Full optimization uses the smallest number of hosts to provide the required performance. Optimization with persistence (Pers) is more effective than simple rules in reducing the active hosts most of the time, but not always, since in the same control period the available hosts remaining in the pool are not the same for each algorithm.

Figure 8.12 gives the percentage of new hosts in use. In most steps, the percentage of new hosts in full optimization is larger than the others. When the stress rate reaches 0.7 and new hosts are added into the pool, full optimization introduces more changes because of the increase of optimization options.

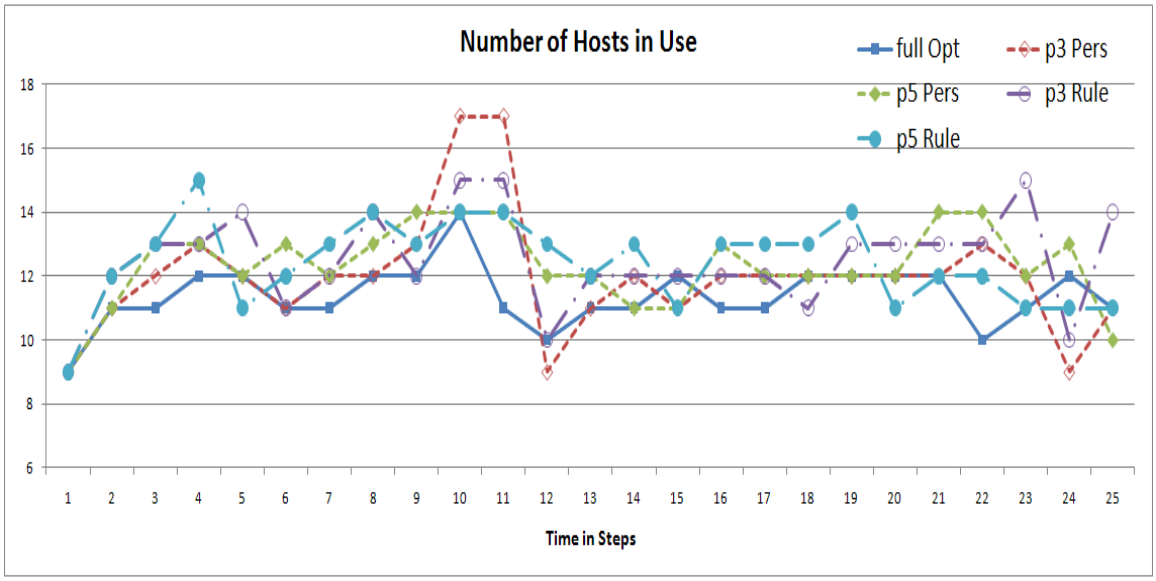


Figure 8.11 Active Hosts in the Host Failures and Repairs Environment

Table 8.11 Average Number of Hosts per Step (Host Failures and Repairs)

	Full opt	P3 Pers	P5 Pers	P3 Rule	P5 Rule
Average Number of Hosts	11.36	11.96	12.32	12.48	12.4

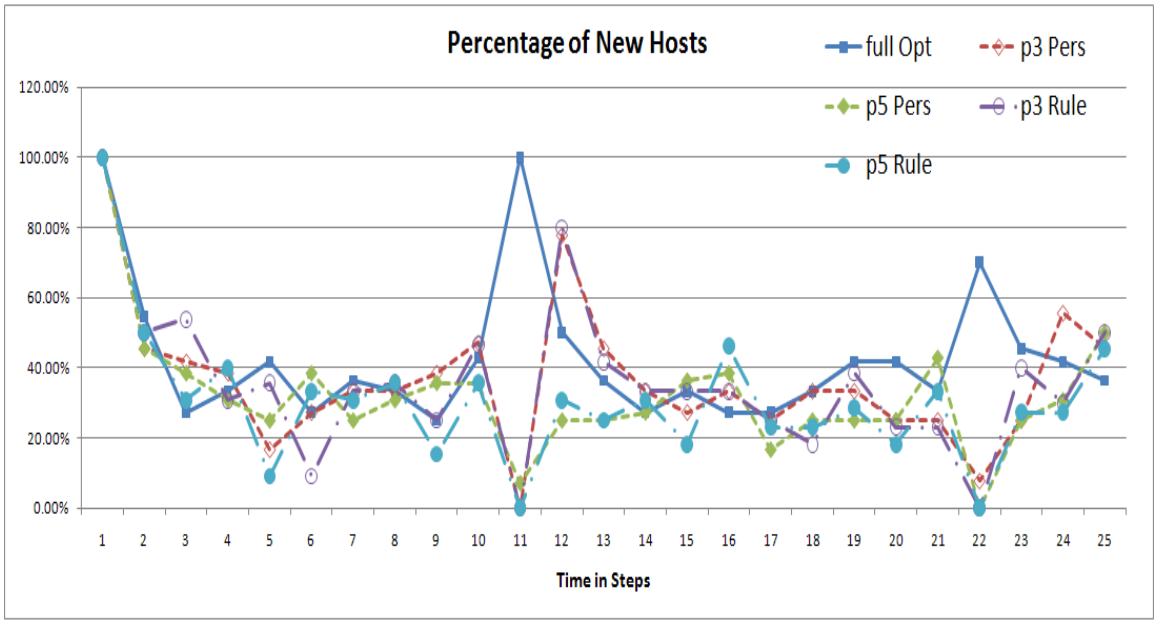


Figure 8.12 Active Hosts in the Host Failures and Repairs Environment

Table 8.12 Average Percentage of New Hosts per Step (Host Failures and Repairs)

	Full opt	P3 Pers	P5 Pers	P3 Rule	P5 Rule
Average Percentage of New Hosts	42.67%	36.57%	32.20%	35.58%	30.32%

4. Summary: Management for the Host Failures and Repairs Environment

The above study evaluated the effectiveness of algorithms in host failed/repaired environments. Full optimization gives the most economical solution (saving 10~20% energy and license cost), but it is associated with many changes in each step. In comparison to simple rules, optimization with persistence (Pers) can reduce the energy costs, reduce the number of replicas used, and give solution persistence, with results that are nearly as good as simple rules. A short period is more effective than a long period in keeping the configurations up-to-date; however, it correspondingly requires many changes, which are associated with increased costs and risks.

8.3.3 Case III: Management for Applications Addition/Removal

In this test, applications are deployed on a private cloud consisting of 29 hosts. Applications are increased from 2 to 11 and then removed until 3 applications remain. The process is repeated. It is a highly dynamic environment. There are applications to be deployed or removed at each step.

This test studies the effectiveness of the algorithms in handling application addition/removal. The management system is required to adaptively adjust resource provisioning in response to the change of the applications and must ensure that each application does not exceed the maximum response time.

1. Costs associated with the Application Addition/removal

This first measurement of the capital and start-up costs (Figure 8.13, Table 8.13) shows that the algorithms in this test perform similarly as to when handling varying workloads or when hosts fail and are repaired. Full optimization gives the most economical solution; control with persistence is better than simple rules. A short period between full optimizations is more helpful to reduce the energy and license cost than a long period. Simple rules have higher energy costs than the other methods because more hosts are used. However, full optimization is associated with high start-up cost, which is over double of other approaches. And a shorter period between successive full optimizations increases the start-up cost of both hosts and new replicas.

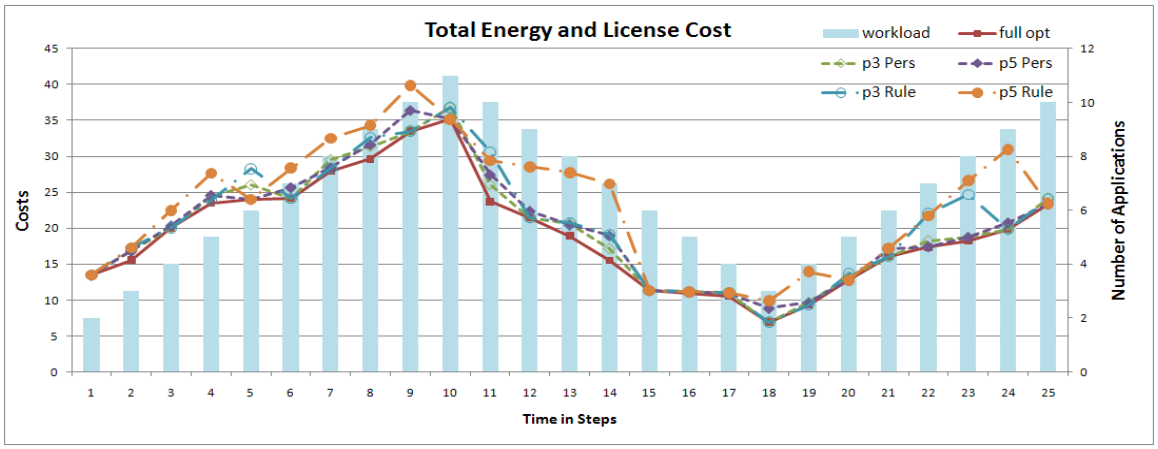


Figure 8.13 the Total Energy and License Cost subject to Application Addition/removal

Table 8.13 Average Costs per Step (Application Addition/removal)

	Full opt	P3 Pers	P5 Pers	P3 Rule	P5 Rule
Total Energy and License Cost	19.34	20.07	20.33	20.81	23.08
Start-up costs on new hosts	2.68	1.84	1.28	2.04	1.76
Start-up costs on new replicas	143.88	75.72	61.56	59.16	46.2
Start-up costs on new hosts and new replicas	146.56	77.56	62.84	61.2	47.96

2. Active Hosts subject to the Application Addition/removal

Figure 8.14 shows the variation of active hosts in response to the change of applications. It shows that simple rules need an extra 20~30% additional hosts vs. other solutions, since running applications cannot be changed. When newly arrived applications are to be deployed, simple rules seek unused hosts to accommodate the new applications; when some applications are to be removed, the remaining unrelated tasks keep running without changes, so there is no significant reduction in the number of the active hosts. Therefore, the number of hosts in use is more than in the other approaches in either adding newly arrived applications or removing terminated applications.

Figure 8.15 shows that without the requirement of persistence the percentage of new hosts used by full optimization is around 20~50% most of the time, and sometimes reaches 100%. In the other approaches, the percentages of new hosts are controlled below 30%, and in some steps, the percentage of changes required by simple rules is 0.

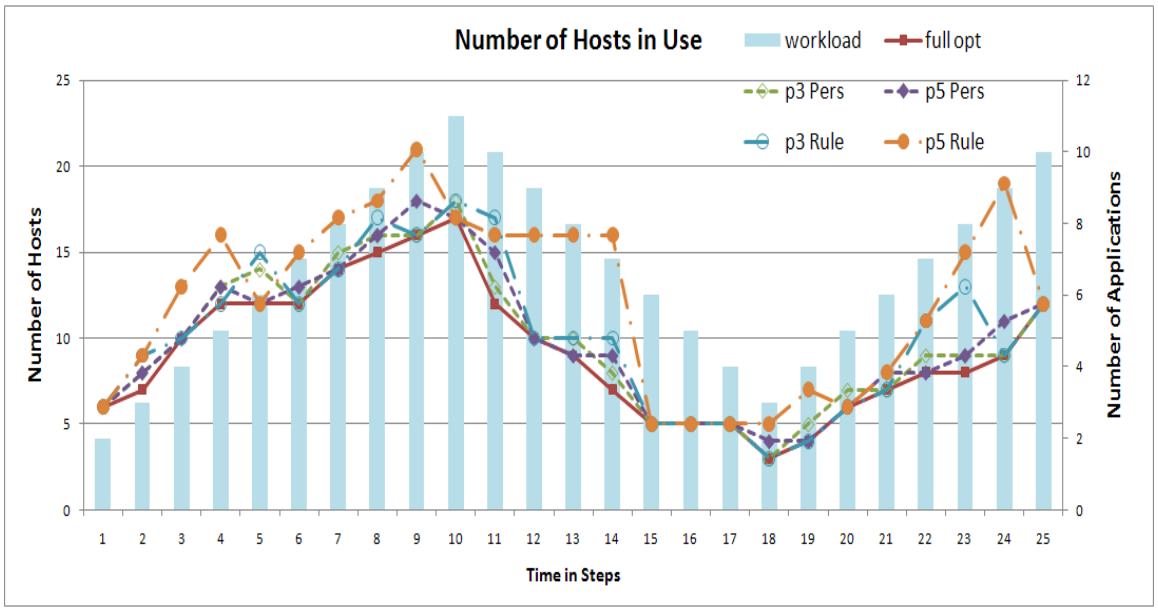


Figure 8.14 the Total Number of Hosts in Use subject to Application Addition/removal

Table 8.14 Average Number of Active Hosts per Step (Application Addition/removal)

	Full opt	P3 Pers	P5 Pers	P3 Rule	P5 Rule
Average Number of Hosts in Use	9.24	9.8	9.88	10.24	12.24

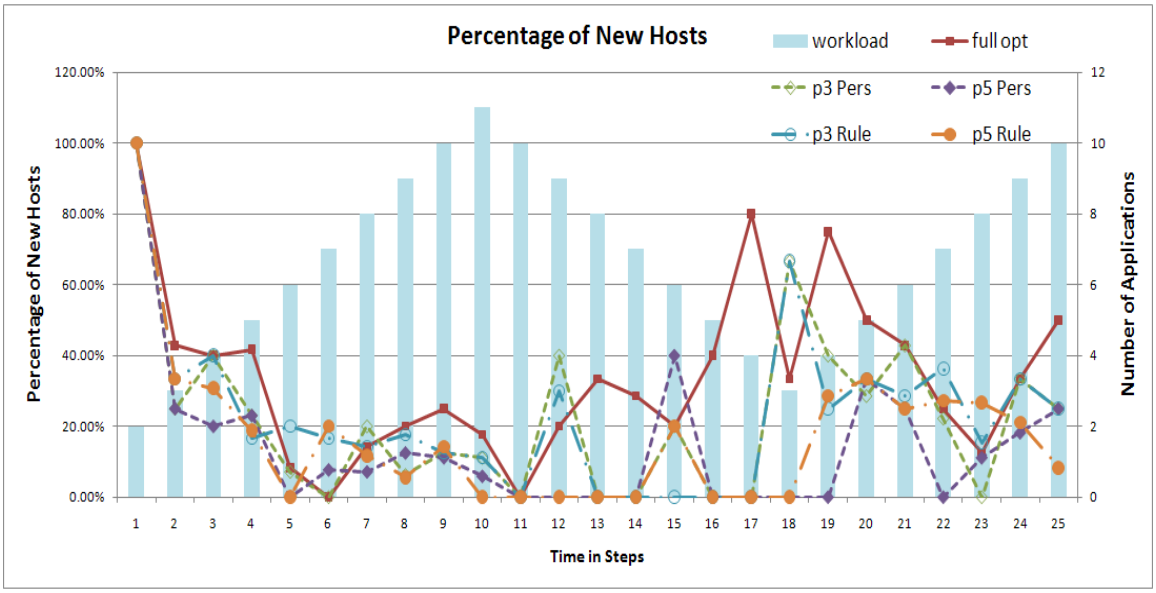


Figure 8.15 the Percentage of New Hosts subject to Application Addition/removal

Table 8.15 Average Percentage of New Hosts per Step (Application Addition/removal)

	Full opt	P3 Pers	P5 Pers	P3 Rule	P5 Rule
Average Percentage of New Hosts	34.15%	22.55%	14.60%	23.03%	16.99%

3. Replicas in Use and Persistence

The numbers of replicas required by each approach are very similar except for the P5 Rule that takes slightly fewer replicas. This is different than the results in the previous experiments on varying workloads and host failures and repairs. The number of replicas is affected by two operations: workload aggregation, which can reduce the number of replicas, and workload distribution, which may increase the number of replicas. In

workload variation or host failures and repairs, changes have direct impacts on the running applications. Such operation as workload aggregation, placing the existing and newly arrived (or affected) loadings into a new replica, can be conducted by the full optimization or the optimization with persistence (Pers). However this is not allowed in simple rules because of changes to the running tasks. Other approaches thus can use fewer replicas to accommodate the load than simple rules (which need to use extra replicas to accommodate the new loadings) in these cases.

In the current case of application addition/removal, newly arrived applications are independent of the running applications. This means that the new workloads to be deployed cannot be aggregated with the existing workloads. So no workload aggregation is used. And since the full optimization or the optimization with persistence (Pers) allows splitting arrival loading into several small-size replicas in order to reduce costs by sharing resources with running tasks, the number of replicas could be increased in comparison to the simple rules, which only place new loadings onto new hosts.

Figure 8.17 shows that the percentage of new replicas in full optimization is over 80%; optimization with persistence (Pers) is around 20%~40% and the simple rules limit changes in the range of 0~20%. This shows that simple rules need to install the fewest replicas, and optimization with persistence (Pers) needs a bit more. To guarantee stability for management, the simple rules are good candidates. Optimization with persistence (Pers) is a sound algorithm, as long as the costs associated with resource utilization are taken into account. If it is only required to reduce the execution costs, full optimization performs the best.

We see again that a short period can deliver a solution with lower costs in response to shifts in the environments, but is associated with great changes to the configurations. Optimization with persistence (Pers) in a dynamic process is not only capable of achieving the expected performance at low cost, but it can effectively reduce the number of changes, thereby significantly increasing management satiability.

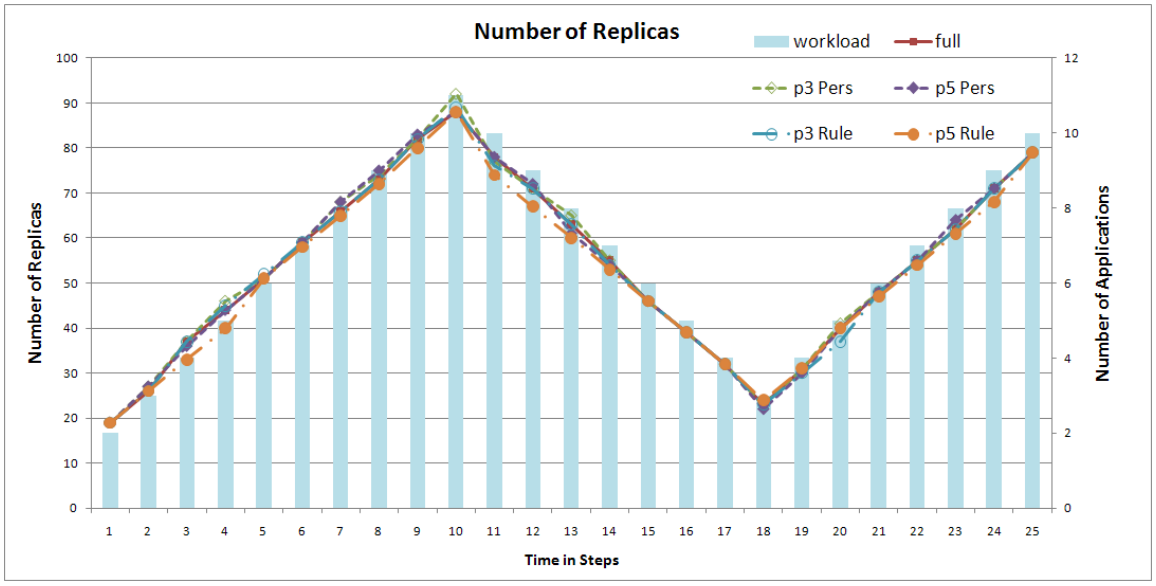


Figure 8.16 the Total Number of Replicas subject to App Addition/removal

Table 8.16 Average Number of Replicas per Step (Application Addition/removal)

	Full opt	P3 Pers	P5 Pers	P3 Rule	P5 Rule
Average Number of Replicas	53.52	54	53.64	53.36	52.28

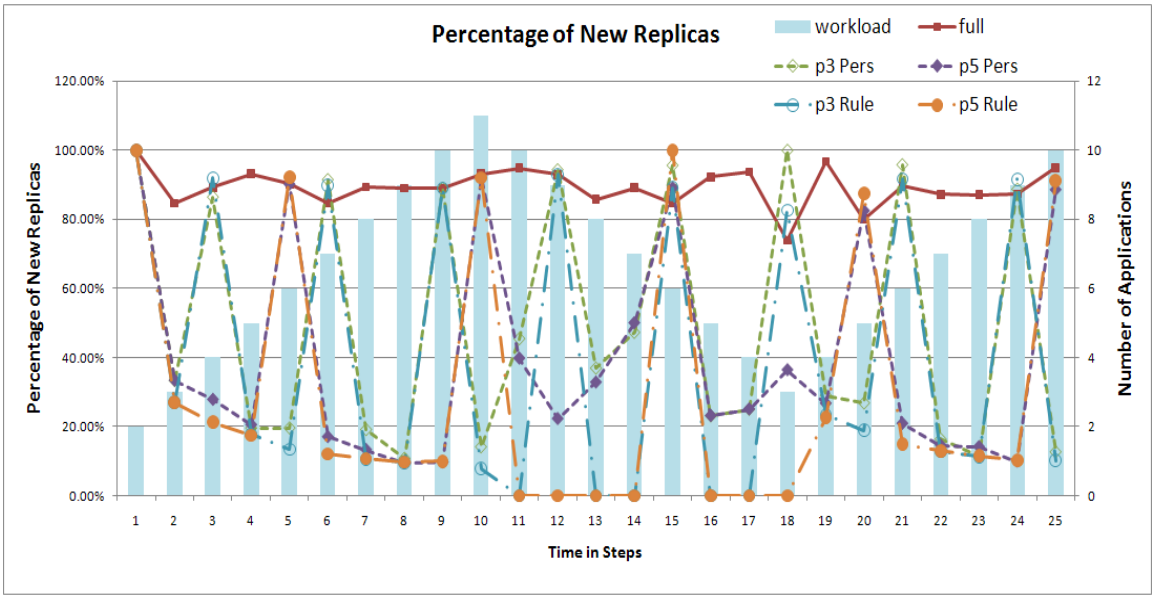


Figure 8.17 the Percentage of New Replicas subject to the App Addition/removal

Table 8.17 Average Percentage of New Replicas per Step (Application Addition/removal)

	Full opt	P3 Pers	P5 Pers	P3 Rule	P5 Rule
Average Percentage of New Replicas	89.32%	49.28%	39.53%	39.25%	29.73%

4. Summary: Management of Application Addition/removal

The evaluation demonstrates that full optimization gives the smallest energy and license cost in response to changing applications, but is associated with a large number of changes to replicas, increasing the potential risks to the system stability and high costs due to operations for changes.

Simple rules give good persistence; however, they cannot keep the configurations up-to-date, resulting in many resources being over allocated. These wasted resources increase the energy costs.

Optimization with persistence (Pers) achieves a level of persistence similar to that of simple rules (about 10% difference), with costs that are within 25% of those for full optimization.

A short period helps keep the configurations up-to-date, reducing the energy and license cost due to wasted resources, but it also causes more frequent changes in the assignments. Therefore, if it is important to maintain a stable performance, a long period performs better than short period.

8.3.4 Summary of the Effectiveness of the Algorithms in Dynamic Environments

The above evaluations under conditions of varying workloads, application addition/removal and host failure/repair show the effectiveness of the algorithms in response to changes, and the effects of period variation. The experimental environment was highly dynamic, but the algorithms provide effective responses to the changing conditions.

The experimental results show that all approaches are capable of managing application performance subject to changes. Full optimization is able to achieve the most economical solutions among the algorithms. Optimization with persistence (Pers) gives a solution with lower energy and license cost than using simple rules. This is because optimization with persistence (Pers) has greater flexibility to regulate the resource provisioning in adapting to the changing resource requirements. Allowing resource sharing with running replicas helps to reduce the resource cost.

Optimization with persistence (Pers) mechanisms control changes subject to dynamic variations. In the above experiments, both optimization with persistence (Pers) and

simple rules control the number of new replicas below 40%, while the percentage of new replicas required by the full optimization is around 90%.

In these experiments, full optimization saves about 5~10% costs in comparison with optimization with persistence (Pers), but an update is associated with many changes. Simple rules give stable management; however its solutions are associated with high costs. Optimization with persistence (Pers) returns solutions with similar cost to full optimization, and is able to limit the changes on a system while maintaining the quality of the solution. These experimental results show that optimization with persistence (Pers) is more suitable for dynamic environments than either full optimization or simple rules. A short period between full optimizations helps to achieve an economical solution, but is associated with big changes.

Chapter 9 Conclusions

New algorithms have been described for deployment management of large scale service systems deployed in clouds. Test cases demonstrated the ability to handle numerous applications simultaneously, on many hosts, to accommodate multiple constraints, and to provide stable solutions over time in a dynamically changing environment.

9.1 Achievements

This thesis presents several new approaches that are effective for deployment optimization.

Algorithm I (Chapter 5) is a creative approach to seek near-optimal solutions, giving sound allocations of host reservations to tasks, and optimizing request traffic between multiple task replicas, where applicable. A key contribution of the combination of NFM with an analytical model (LQN) is its effectiveness in solving a non-linear constrained optimization problem by a series of LP solutions. The experiment in Section 5.8 shows that this approach can address very difficult problems, such as minimizing multi-class response time with low cost in very large systems. An early version has been demonstrated on a small real system comprised of the Tivoli Intelligent Orchestrator, Websphere and DB2.

The LEndStep algorithm (Section 5.7) gives a new tool to optimize loadings across allocated replicas. Experimental results in Section 6.4 and Section 8.2 demonstrate that linear programming based on sensitivity analysis can save execution power and reduce resource consumption.

An effective MIP model extends the NFM to address integer constraints. This model (model II, Section 6.1) accounts for the QoS in SLAs, execution power and available capacity, allocation of memory to tasks, license availability and costs, and power costs associated with host activity at the same time. A new algorithm, named HMIP (Algorithm III in Section 6.2), provides a scalable heuristic solution for the MIP models. Experimental results demonstrate that HMIP is an effective tool that provides high quality decisions, nearly as good as using an exact MIP solver. HMIP can handle extremely large scale problems consisting of over 80,000 variables, corresponding to over 100 hosts and 500 tasks, which is beyond the capability of such MIP solvers as CPLEX v12.1.

Providing high-quality adaptive resource provisioning is another key contribution of this thesis. Several new approaches have been developed in this research, which are effective in providing ensured QoS subject to dynamic changes. The approaches for re-optimization with persistence (in Chapter 7) perform very well in the face of dynamic changes in demand. Hard constraints on the flow bounds, soft constraints to guide selection, and precise control on the scale of changes improve the stability of the management system. These approaches can be combined to satisfy many difficult goals. The experiments in Section 7.3 show the effectiveness of the methods in controlling the

number of new installations in order to limit the associated costs/risks. The experimental results (in Section 8.3) demonstrate that the persistence mechanism is able to deliver high quality solutions with limited changes on running applications.

Experiments demonstrate that this deployment approach is able to scale up to hundreds of applications across hundreds of machines. With the improvement of the speed of the analytical performance model calculations and the optimization algorithms for MIPs in the future, this approach has great potential to handle much larger numbers of applications for real clouds.

9.2 Limitations and Assumptions

A prerequisite of applying the algorithms in management is that the performance model must be available for the service system to be controlled. In dynamic environments, model parameter predictions are needed to update the performance model in advance, to allow control without delays. Many performance researchers have proposed effective solutions to construct the performance model (from scenarios defined in UML or from operational data), estimate the system parameters (using tracking filters or statistical data analysis), and update the performance with prediction algorithms [51]. So this is not a critical limitation. In practice, there may be some delay in delivering the data for the performance model update, which may result in some errors of prediction. This issue can be addressed by giving the estimated parameters some margin, such as increasing the estimated value by 10%.

As the experimental results demonstrate, over 85% of the solution time is used on the performance model calculations. In comparison with the scalability and efficiency of

solving LP or MIP, the scalability of the performance model currently is the bottleneck of this approach. If the efficiency of the performance model can be improved in the future, this approach has great potential to handle much larger scale applications. Improving the calculation capability of the performance model can help to increase the scalability of LEndStep and extend its applicable areas.

9.3 Future Work

This thesis addresses the optimization of deployment decisions for performance on the basis of software architecture. Now that the effectiveness of the optimization algorithms has been demonstrated, more comprehensive extensions might be possible. Multi-tier caching is commonly use nowadays and has a significant impact on performance. But the performance of caching is related to disk/memory operations and data structure. How to model the cache performance with a performance model and how to describe these issues with effective optimization models are not covered in this thesis, but these might be worthwhile future research topics.

Co-allocation is another problem not addressed so far. A good co-allocation can reduce the overhead for communications. Some constraints on the allocations may help to guide the optimization. However, the calculation of costs is nonlinear, since communication costs are associated with the allocations of tasks (determined by A_t) and the workloads (determined by α_{ht}) in transmission. Though an optimization model to minimize the communication costs can be constructed, new algorithms will be needed to efficiently address these problems with good scalability.

This thesis demonstrates the feasibility of combining an optimization model and LQN. The current version of LQN is the bottleneck of this approach, limiting the scalability and efficiency. Optimizing the LQN system architecture to give efficient analysis could be a research topic in the future. The real cloud system might have thousands of hosts and applications. The scalability and efficiency of the approach should be further improved in the future work.

Solving a MIP problem is NP-hard. Reducing the optimization options is an effective way to improve the solution speed in most cases, but does not guarantee optimality. Developing model specific search strategies (such as decomposition methods) for these large-scale MIPs might improve the solution efficiency in conjunction with the current HMIP heuristic. How to further improve the efficiency is a problem to be answered in the future.

Reference

[1] Abdelzaher, T. F. and Bhatti, N., "Web Server Qos Management By Adaptive Content Delivery", in *Proceedings of International Workshop on Quality of Service, London*, UK, Jun 1999.

[2] Abdelzaher, T., Shin, K. G., and Bhatti, N., "Performance Guarantees for Web Server End-Systems: A Control-Theoretical Approach", *IEEE Transactions on Parallel and Distributed Systems*, Vol. 13, No. 1, pp. 80-96, Jan 2002.

[3] Amazon Corp. "Auto Scaling Developer Guide", http://aws.amazon.com/documentation/autoscaling/ accessed Nov. 23, 2010

[4] Arlitt, M. and Jin, T., "Workload Characterization Of The 1998 World Cup Web Site", Hewlett-Packard Labs, Technical Report HPL-99-35R1, Sept. 1999.

[5] Bennani, M.N., and Menascé, D. A., "Resource Allocation for Autonomic Data Centers Using Analytic Performance Models", in *Proceedings of IEEE International Conference on Autonomic Computing*, Seattle, WA, June 13-16, 2005.

[6] Bennani, M.N., and Menascé, D. A., "Assessing the Robustness of Self-Managing Computer Systems under Highly Variable Workloads", in *Proceedings of International Conference on Autonomic Computing,*New York, NY, May 17-18, 2004.

[7] Bilgin, S. and Azizoğlu, M.,"Operation Assignment And Capacity Allocation Problem In Automated Manufacturing Systems", *Journal of Computers and Industrial Engineering*, Vol. 56, Issue. 2, pp. 662-676, Mar 2009.

[8] Bazaraa, M.S., Jarvis, J.J., Sherali, H.D., "Linear Programming and Network Flows", *John Wiley & Sons, Inc*., Hoboken, New Jersey, 2005.

[9] Bokhari, S. H. "Partitioning Problems in Parallel, Pipeline, and Distributed Computing", *IEEE Transactions on Computers*. Vol.37, Issue. 1, pp. 48-57, Jan 1988.

[10] Bobroff, N. , Kochut, A., and Beatty, K. "Dynamic Placement Of Virtual Machines For Managing SLA Violations", in *Proceedings of Integrated Management*, pp 119-128, Munich, Germany, May 2007.

[11] Brown, G., Dell, R., Wood, K., "Optimization and Persistence", *Interfaces* 1997.

[12] Calinescu, R. and Kwiatkowska, M. 2009. "Using Quantitative Analysis To Implement Autonomic IT Systems", in *Proceedings of the IEEE 31st international Conference on Software Engineering* , Vancouver, Canada, May 16 - 24, 2009.

[13] Carrera, D., "Adaptive Execution Environments for Application Servers", *PhD dissertation*, *Technical University of Catalonia,* 2008.

[14] CERAS (Centre of Excellence for Research in Adaptive Systems) https://www.cs.uwaterloo.ca/twiki/*view/CERAS* accessed April 2011.

[15] Chaisiri, S.; Bu-Sung Lee; Niyato, D., "Optimal Virtual Machine Placement Across Multiple Cloud Providers", in *Proceedings of IEEE Asia-Pacific Services Computing Conference,* pp. 103 – 110, Singapore, December 2009.

[16] Chao, S., Chinneck, J.W., Goubran, R.A., "Assigning Service Requests in Voice-over-Internet Gateway Multiprocessors", *Computers and Operations Research*, Vol. 31, pp. 2419-2437, 2004.

[17] Chen, J., Soundararajan, G., and Amza, C. 2006. "Autonomic Provisioning of Backend Databases in Dynamic Content Web Servers", in *Proceedings of the 3rd IEEE International Conference on Autonomic Computing* , Dublin, Ireland, June 12 - 16, 2006.

[18] Chen, Y., Iyer, S., Liu, X., Milojicic, D., and Sahai, A. 2007. "SLA Decomposition: Translating Service Level Objectives to System Level Thresholds", in *Proceedings of the 4th IEEE International Conference on Autonomic Computing*, Florida, USA, June 11 - 15, 2007.

[19] Cherkasova, L., and Phaal, P., "Session Based Admission Control: A Mechanism For Improving The Performance Of An Overloaded Web Server", *Technical Report* HPL-98-119, HP Labs, June 1998.

[20] Chinneck, J.W., "Processing Network Models of Energy/Environment Systems", *Computers and Industrial Engineering*, vol. 28, no. 1, pp. 179-189, 1995.

[21] Coffman, E.G, Garey, M.R., Johnson, D.S., "An Application Of Bin-Packing To Multiprocessor Scheduling", *SIAM Journal on Computing*, vol. 7, pp. 1-17, Feb. 1978.

[22] Coffman, E.G, Garey, M.R., Johnson, D.S., "Approximation Algorithms for Bin Packing: a Survey", in *Approximation Algorithms For NP-Hard Problems*, D. S. Hochbaum, Ed. PWS Publishing Co., Boston, MA, pp 46-93, 1997.

[23] CPLEX,http://www-01.ibm.com/software/integration/optimization/cplex-optimizer/ accessed Dec. 2010.

[24] Kusic, D., Kephart, J., Hanson, J., Kandasamy, N., and Jiang, G., "Power And Performance Management Of Virtualized Computing Environments Via Lookahead Control". *Transaction of Cluster Computing* , Vol.12, Issue.1, pp.1-15, March 2009.

[25] Diao, Y., Hellerstein J. L., Parekh S., Bigus, J. P., "Managing Web server performance with AutoTune agents", *IBM Systems Journal*, Vol.42, No.1, pp.136-149, January 2003.

[26] Diao, Y., Hellerstein, J. L., and Parekh, S. 2002. "Optimizing Quality of Service Using Fuzzy Control", in *Proceedings of IFIP/IEEE international Workshop on Distributed Systems: Operations and Management: Management Technologies For E-Commerce and E-Business Applications* , October 21 - 23, 2002.

[27] Diao, Y , Gandhi N., et al.: "Using MIMO Feedback Control To Enforce Policies For Interrelated Metrics With Application To The Apache Web Server", in *Procceding of the Network Operations and Management Symposium*, Florence, Italy. 2002.

[28] Diao, Y., Hu, X., Tantawi, A., and Wu, H. 2009. "An Adaptive Feedback Controller For SIP Server Memory Overload Protection", in *Proceedings of the 6th international Conference on Autonomic Computing*, Barcelona, Spain, June 15 - 19, 2009.

[29] Franks, G., Petriu, D., Woodside, M., Xu, J., and Tregunno, P. "Layered Bottlenecks and Their Mitigation", in *Proceedings of the 3rd international Conference on the Quantitative Evaluation of Systems* ,Riverside, California, September 11 - 14, 2006.

[30] Franks, G., Al-Omari, T., Woodside, M., Das, O., and Derisavi, S. 2009. "Enhanced Modeling and Solution of Layered Queueing Networks", *IEEE Transaction of Software Engineering,* Vol.35, Issue. 2, pp.148-161, Mar. 2009.

[31] Franks, G., "Performance Analysis of Distributed Server Systems", *Report OCIEE-00-01, PhD. thesis*, *Carleton University*, Jan. 2000.

[32] Gelenbe, E., Bagchi, K., Zobrist, G, "Network Systems Design", *CRC 1 edition* April 23 1999.

[33] Garey, M.R. , Johnson. D.S.. "Computers And Intractability: A Guide To The Theory Of NP-Completeness". W.H. Freeman and Company; 1979.

[34] Gartner "Cloud Computing Will Be As Influential As E-business" , *Special Report Examines the Realities and Risks of Cloud Computing, STAMFORD, Conn.,* June 26, 2008

[35] Gmach, D., Krompass, S., Scholz, A., Wimmer, M., and Kemper, A., "Adaptive Quality Of Service Management For Enterprise Services". *ACM Transaction of Web*, Vol.2, No.1, pp.1-46, Feb. 2008.

[36] Hellerstein, J. L., Diao, Y., Parekh, S., and Tilbury, D. M. "Feedback Control of Computing Systems". *John Wiley & Sons*., 2004

[37] Hellerstein, J. L,"Engineering Autonomic Systems ". *Keynote Talk, ICAC* 2009.

[38] IBM Corp., "From Cloud Computing to the New Enterprise Data Center", 2011.

[39] IBM Corp., "Cloud Computing", http://www.ibm.com/ibm/cloud/ibm_cloud/, accessed Dec 2009.

[40] IBM Corp., "Dynamic Infrastructure", http://www03.ibm.com/systems/dynamicinfrastructure/, accssed Dec. 2009

[41] IBM Corp., "Seeding the Clouds: Key Infrastructure Elements for Cloud Computing" , http://www.ibm.com/grid/, accssed Dec.2009

[42] IBM Corp., "IBM EnergyScale for POWER7 Processor-Based Systems", http://www-03.ibm.com/systems/power/hardware/whitepapers/energyscale7.html accessed April 2010

[43] IBM Corp., "The New Enterprise Data Center Technical White Paper", http://www-935.ibm.com/services/in/gts/pdf/ibm_nedc_whitepaper_07_15.pdf, accessed April, 2011

[44] IBM Corp., "An Architectural Blueprint For Autonomic Computing", http://www01.ibm.com/software/tivoli/autonomic/pdfs/AC_Blueprint_White_Paper_4th.pdf accessed April 2010.

[45] IBM Corp., "Understanding Quality Of Service For Web Services", http://www.ibm.com/developerworks/library/ws-quality.html, accessed Jun 2010.

[46] Kansal, A., Zhao, F., Liu, J., Kothari, N., and Bhattacharya, A. ,"Virtual Machine Power Metering And Provisioning", in *Proceedings of the 1st ACM symposium on Cloud computing (SoCC '10)*. ACM, New York, NY, USA, 2010.

[47] Kephart,J.O., and Das, R.,"Achieving Self-Management via Utility Functions", *IEEE Internet Computing*, Vol. 11, No. 1, pp. 40-48, Jan./Feb. 2007.

[48] Karve, A., Kimbrel, T., Pacifici, G., Spreitzer, M., Steinder, M., Sviridenko, M., and Tantawi, A., "Dynamic Placement For Clustered Web Applications", in *Proceedings of the 15th international Conference on World Wide Web*, Edinburgh, Scotland, May 23 - 26, 2006.

[49] Kounev, S. and Buchmann, A., "Simqpn: A Tool And Methodology For Analyzing Queueing Petri Net Models By Means Of Simulation", *Performance. Evauationl,* Vol.63, Issue. 4, pp.364-394, May 2006.

[50] Krishna B. Misra, (Ed.) "Handbook of Performability Engineering", *Springer*, August 27, 2008.

[51] Kumar. D, Tantawi. D, and Zhang. L., "Real-Time Performance Modeling For Adaptive Software Systems", in *Proceedings of the Fourth International ICST Conference on Performance Evaluation Methodologies and Tools*, Brussels, Belgium, 2009.

[52] Lazowska, E. D., Zahorjan, J., Graham, G. S., and Sevcik, K. C., "Quantitative System Performance: Computer System Analysis Using Queueing Network Models", *Prentice-Hall, Inc*. Feb, 1984.

[53] Li, J, " CLoudOpt: Multi-Goal Optimization of Application Deployments Across a Cloud ", *Techniqual Report*, Setp -15 2010.

[54] Li, J, "Optimization with Persistence for Deployment Management in Cloud Computing", *Techniqual Report*. Nov 2010

[55] Li, J., Chinneck, J., Woodside, M., Litoiu, M., and Iszlai, G, "Performance Model Driven QoS Guarantees and Optimization in Clouds", in *Proceedings of Workshop on Software Engineering Challenges in Cloud Computing @ ICSE 2009*, Vancouver, May 2009.

[56] Li, J., Chinneck, J., Woodside, M., Litoiu, M, "Fast Scalable Optimization to Configure Service Systems having Cost and Quality of Service Constraints", in *Proceedings of 6th Interational Conference on Autonomic Computing,* Barcelona, Spain, June 2009.

[57] Li, J., Chinneck, J., Woodside, M., Litoiu, M, "Deployment of Services in a Cloud Subject to Memory and License Constraints", in *Proceedings of 2nd IEEE International Conference on Cloud Computing* , Bangalore, India, September 21-25, 2009.

[58] Li, J., Levy, D., Chen, S., and Zic, J., "Auto-Tune Design And Evaluation On Staged Event-Driven Architecture", in *Proceedings of the 1st Workshop on Model Driven*

Development For Middleware (MODDM '06) @ Middleware 2006, Melbourne, Australia, November 27 - December 01, 2006.

[59] Li, J., Levy, D., Chen, S., and Zic, J., "Explicitly Controlling the Fair Service for Busy Web Servers", in *Proceedings of the 2007 Australian Software Engineering Conference* , Melbourne, Australia, April 10 - 13, 2007.

[60] Lightstone, S. S., Surendra, M., Diao, Y., Parekh, S., Hellerstein, J. L., Rose, K., Storm, A. J., and Garcia-Arellano, C., "Control Theory: a Foundational Technique for Self Managing Databases", in *Proceedings of the 2007 IEEE 23rd international Conference on Data Engineering Workshop*, Istanbul, Turkey, April 17 - 20, 2007.

[61] Litoiu, M., Rolia, J., and Serazzi, G, "Designing Process Replication and Activation: A Quantitative Approach", *IEEE Transactions on Software Engineering,* Vol. 26, No. 12, pp. 1168-1178, Dec. 2000.

[62] Litoiu M., Woodside M., Zheng T., "Hierarchical Model Based Autonomic Control Of Software Systems", in *Proceedings of Design and Evolution of Autonomic Software (DEAS'05) Workshop*, St. Louis, USA, May 2005.

[63] Little, J. D. C. "A Proof of *the Queueing Formula L = λ W," Operations Research*, 9, 383-387 ,1961.

[64] Liu Z., Wynter L., Xia C. H., Zhang F., "Parameter Inference Of Queueing Models For IT Systems Using End-To-End Measurements", *Performance Evaluation*, Vol. 63, Issue 1 , pp36-60, Jan. 2006.

[65] Liu, Y., Fekete, A., Gorton, I., "Design Level Performance Prediction of *Component-Based* Applications", *IEEE Transactions on Software Engineering*, Vol. 31, No.11, pp. 928-941, November, 2005.

[66] Luciano Bertini, Julius C. B. Leite, and Daniel Moss, "Power Optimization For Dynamic Configuration In Heterogeneous Web Server Clusters", *Journal of System and Software.* Vol. 83, Issue. 4, pp. 585-598, April 2010.

[67] Martens, A., Brosch, F., and Reussner, R., "Optimising Multiple Quality Criteria Of Service-Oriented Software Architectures", in *Proceedings of the 1st international*

Workshop on Quality of Service-Oriented Software Systems, Amsterdam, Netherlands, August 25 - 25, 2009.

[68] Menasce, D. A. and Almeida, V., "Capacity Planning for Web Services: Metrics, Models, and Methods," *Prentice Hall; Rev Sub edition*, 2001.

[69] Menascé, D. A., "Automatic QoS Control". *IEEE Internet Computing,* Vol.7, No.1 ,pp.92-95, Jan 2003.

[70] Menascé, D. A., Ruan, H., and Gomaa, H., "A Framework For Qos-Aware Software Components ", in *Proceedings of 3rd ACM International Workshop on Software and Performance*, Redwood Shores, California, Jan 2004.

[71] Menascé, D. A., Casalicchio, E., and Dubey, V., "A Heuristic Approach To Optimal Service Selection In Service Oriented Architectures", in *Proceedings of 7th International Workshop on Software and Performance,* Princeton, USA, Jun 2008.

[72] Menascé, D. A., and Bennani, M.N., "Autonomic Virtualized Environments", in *Proceedings of IEEE International Conference on Autonomic and Autonomous Systems*, Silicon Valley, CA, USA, July 19-21, 2006.

[73] Menascé, D. A., and Bennani, M.N., "Dynamic Server Allocation for Autonomic Service Centers in the Presence of Failures". *in the book Autonomic Computing: Concepts, Infrastructure, and Applications, eds. S. Hariri and M. Parashar*, ISBN 0-8493-9367-1, CRC Press, 2006.

[74] Menascé, D. A., Bennani, M.N. and Ruan, H., "On the Use of Online Analytic Performance Models in Self-Managing and Self-Organizing Computer Systems". *in the book Self-Star Properties in Complex Information Systems, O. Babaoglu, M. Jelasity, A. Montresor, C. Fetzer , S. Leonardi, A. van Moorsel, and M. van Steen, eds., Lecture Notes in Computer Science,* Vol. 3460, Springer Verlag, 2005.

[75] Menascé, D. A., and Bennani, M.N., "On the Use of Performance Models to Design Self-Managing Computer Systems", in *Proceedings of Computer Measurement Group Conference*, Dallas, TX, Dec. 7-12, 2003.

[76] Menasce , D.A., Barbara, D., and Dodge, R.,"Preserving QoS of E-commerce Sites Through Self-Tuning: A Performance Model Approach", in *Proceedings of 2001 ACM Conference on E-commerce*, Tampa, FL, October 14-17, 2001.

[77] Pacifici, G., Segmuller, W., Spreitzer, M., Tantawi. A., "CPU Demand For Web Serving: Measurement Analysis And Dynamic Estimation", *Performance Evaluation*, Vol.65, Issues 6-7, December 2007.

[78] Petri, Carl A. "Kommunikation Mit Automaten". Ph. D. Thesis (1962).*University of Bonn.*

[79] Ramirez, A. J., Knoester, D. B., Cheng, B. H., and McKinley, P. K, "Applying Genetic Algorithms To Decision Making In Autonomic Computing Systems", in *Proceedings of the 6th international Conference on Autonomic Computing*, Barcelona, Spain, June 15 - 19, 2009.

[80] Rao, J., Bu, X., Xu, C., Wang, L., and Yin, G., "VCONF: A Reinforcement Learning Approach To Virtual Machines Auto-Configuration", in *Proceedings of the 6th international Conference on Autonomic Computing*, Barcelona, Spain, June 15 - 19, 2009.

[81] Rayward-Smith, V. J., Osman, I. H., Reeves, C. R., "Modern Heuristic Search Methods", *John Wiley*. December 1996.

[82] Rolia, J. A. and Sevcik, K. C., "The Method of Layers", *IEEE Transaction of Software Engineering*, Vol. 21, No.8,pp. 689-700, Aug 1995.

[83] Salesforce.com, Quality of Services, http://trust.salesforce.com/trust/status/, accessed May 31, 2009.

[84] Salesforce.com, SAAS, http://www.salesforce.com/saas/ accessed May 2009.

[85] Salehie. M., and Tahvildari. L., "Self-Adaptive Software: Landscape And Research Challenges", *ACM Transaction on Autonomous and Adaptive Systems (TAAS)*, Vol. 4, No. 2, pp. 1-42, 2009.

[86] Smith, CU., Williams,LG. "Performance Solutions", Addison-Wesley, 2002.

[87] Soror, A. A., Minhas, U. F., Aboulnaga, A., Salem, K., Kokosielis, P., and Kamath, S, "Automatic Virtual Machine Configuration For Database Workloads", in *Proceedings of the 2008 ACM SIGMOD international Conference on Management of Data* , Vancouver, Canada, June 09 - 12, 2008.

[88] Steinder, M., Whalley, I., Carrera, D., and Chess, D, "Server Virtualization In Autonomic Management Of Heterogeneous Workloads", *Proc.* in *Proceedings of Integrated Management ,* Munich, May 2007.

[89] Storm, A. J., Garcia-Arellano, C., Lightstone, S. S., Diao, Y., and Surendra, M., "Adaptive Self-Tuning Memory In DB2", in *Proceedings of the 32nd international Conference on Very Large Data Bases*, Seoul, Korea, September 12 - 15, 2006.

[90] Sutton, R.S, Barto, A.G. "Reinforcement Learning: an Introduction, Cambridge", *MIT Press*, 1998.

[91] Tang, C., Steinder, M., Spreitzer, M., and Pacifici, G, "A Scalable Application Placement Controller For Enterprise Data Centers", in *Proceedings of 16th International Conference on the World Wide Web*, Banff, Alberta, Canada, 2007.

[92] Toktay and Uzsoy, "A Capacity Allocation Problem With Integer Side Constraints", *European Journal of Operational Research*, Vol.109, Issue.1, pp.170-182, 1998.

[93] Trivedi, K. S. and Sigmon, T. M. 1981. "Optimal Design of Linear Storage Hierarchies", *Journal of the ACM* , Vol. 28, Issue. 2, pp. 270-288, April 1981.

[94] Tsai C., Shin K. G. and Reumann J. and Singhal S., "Online Web Cluster Capacity Estimation and Its Application to Energy Conservation", *IEEE Transaction of Parallel Distributed Syst*ems, Vol.18, Issue.7, pp.932-945, July 2007.

[95] Urgaonkar, B. and Chandra, A, "Dynamic Provisioning of Multi-tier Internet Applications", in *Proceedings of the Second international Conference on Automatic Computing* , June 13 - 16, 2005.

[96] Verma, A., and Neogi, A., “Pmapper: Power And Migration Cost Aware Application Placement In Virtualized Systems”, *in Proceedings of the 9th*

*ACM/IFIP/USENIX International Conference on Middleware,*Leuven, Belgium, 2008.

[97] Vengerov, D., "A Reinforcement Learning Approach To Dynamic Resource Allocation", *Journal of Engineering Applications of Artificial Intelligence*, Vol.20, No.3, pp.383-390, April 2007.

[98] Petrucci, V., Loques, O., and Moss, D., "A Dynamic Optimization Model For Power And Performance Management Of Virtualized Clusters", in *Proceedings of the 1st International Conference on Energy-Efficient Computing and Networking (e-Energy '10)*, University of Passau, Germany, April 13-15,2010.

[99] Xi, B., Liu, Z., Raghavachari, M., Xia, C. H., and Zhang, L., "A Smart Hill-Climbing Algorithm For Application Server Configuration", in *Proceedings of the 13th international Conference on World Wide Web*, New York, NY, USA, May 17 - 20, 2004.

[100] Xu, J., "Rule-based Automatic Software Performance Diagnosis and Improvement", *in Proceedings of ACM WOSP 08, Princeton*, June 2008.

[101] Walsh, W.E., Tesauro, G., Kephart, J.O., and Das, R., "Utility Functions In Autonomic Systems", in *Proceedings of Internation Conference on Autonomic Computing*, New York , USA, May, 2004.

[102] Wei Xu, Xiaoyun Zhu, Sharad Singhal, Zhikui Wang: "Predictive Control for Dynamic Resource Allocation in Enterprise Data Centers", in *Proceedings of 10th IEEE/IFIP Network Operations and Management Symposium,* 2006.

[103] Weise, T., "Global Optimization Algorithms– Theory and Application, 2nd edition", Version: 2009-06-26, http://www.it-weise.de/projects/book.pdf, accessed April 2011

[104] Welsh M. and Culler D.: "Adaptive Overload Control for Busy Internet Servers", in *Proceedings of the 5th USENIX Symposium on Internet Technologies and Systems*, 2003

[105] Woodside, C.M., "Sensitivity Analysis with LQNX/LQX", http://www.sce.carleton.ca/rads/lqns/lqn-documentation/sensitivity-howto.pdf, accessed Mar 10, 2010.

[106] Woodside, C.M., "The Relationship of Performance Models to Data", Keynote talk in *SPEC Internation Workshop on Performance Evaluation (SIPEW)*, Darmstadt, Lecture Notes In Computer Science, Vol. 5119, pp 9 - 28, June 2008.

[107] Woodside, C. M., "A Composable Performance Model for Service/Resource Systems", in *Proceedings of the 7th Workshop on Performability Modelling of Computer and Communications Systems (PMCCS7),* Torino, Italy, pp. 89-92, Sept 2005.

[108] Woodside, C.M., Neilson, J. E., Petriu, D. C., and Majumdar, S., "The Stochastic Rendezvous Network Model for Performance of Synchronous Client-Server-like Distributed Software", in *IEEE Transation of Computers.* 44, 1, pp.20-34, Jan. 1995.

[109] Woodside, C.M., Monforton, G.G., "Fast Allocation of Processes in Distributed and Parallel Systems", *IEEE Transaction. on Parallel and Distributed Systems*, Vol. 4, No. 2, pp. 164-174, 1993.

[110] Woodside, M., Petriu, D. C., Petriu, D. B., Shen, H., Israr, T., and Merseguer, J., "Performance By Unified Model Analysis (PUMA)", in *Proceedings of the 5th International Workshop on Software and Performance*, Palma, Illes Balears, Spain, July 12 - 14, 2005.

[111] Zhang A., Santos A., Beyer D., Tang H.K., "Optimal Server Resource Allocation Using An Open Queueing Network Model Of Response Time", *HP Technical Report HPL-2002-301,* 2002.

[112] Zhang, Q., Cherkasova, L., Mi, N., and Smirni, E., "A Regression-Based Analytic Model For Capacity Planning Of Multi-Tier Applications", *Jouncal of Cluster Computing*, Vol. 11,No.3, Sep. 2008.

[113] Zhang, M., Martin, P., Powley, W., and Bird, P., "Using Economic Models To Allocate Resources In Database Management Systems", in *Proceedings of the 2008*

Conference of the Center For Advanced Studies on Collaborative Research: Meeting of Minds , Toronto, Canada, October 27 - 30, 2008.

[114] Zhang, Z., Cheng, R., Papadias, D., and Tung, A. K., "Minimizing The Communication Cost For Continuous Skyline Maintenance", *in Proceedings of the 35th SIGMOD International Conference on Management of Data*, Providence, Rhode Island, USA, June 29 - July 02, 2009.

[115] Zheng, T., "Model-based Dynamic Resource Management for Multi Tier Information Systems", *PhD thesis*, *Carleton Universtiy*, August 2007.

[116] Zheng, T., "Optimization of Distributed Real-Time Systems with Scenario Deadlines", *M. Eng thesis*, *Carleton Universtiy*, Aug 2002.

[117] Zheng, T., Yang, J., Woodside, M., Litoiu, M., and Iszlai, G., "Tracking Time-Varying Parameters In Software Systems With Extended Kalman Filters", in *Proceedings of the 2005 Conference of the Centre For Advanced Studies on Collaborative Research* , Toranto, Ontario, Canada, October 17 - 20, 2005.

[118] Zheng, T.,Woodside, C. M. and Litoiu, M., "Performance Model Estimation And Tracking Using Optimal Filters", *IEEE Transaction of Software Engineering,* Vol. 34 , No. 3, pp. 391-406, 2008.