QOS ASSURANCE AND CONTROL OF LARGE SCALE DISTRIBUTED COMPONENT BASED SYSTEMS

By

Nilabja Roy

Dissertation

Submitted to the Faculty of the

Graduate School of Vanderbilt University

in partial fulfillment of the requirements

for the degree of

DOCTOR OF PHILOSOPHY

in

Computer Science

December, 2010

Nashville, Tennessee

Approved:

Dr. Aniruddha Gokhale

Dr. Douglas C. Schmidt

Dr. Larry Dowdy

Dr. Gautam Biswas

Dr. Janos Sztipanovits

To Ma, Mashi, Baby and Anindita for always being there. And of course Oishi, such a bundle of joy

ACKNOWLEDGMENTS

I would like to take up this opportunity to thank anyone and everyone who helped me in coming to this point in my life. First and foremost my advisors Dr. Douglas Schmidt and Dr. Aniruddha Gokhale for believing in me and giving me an opportunity to work in the DOC group. Doug had been immensely influential in my development showing me a new perspective of academic and professional world. His ways of teaching are unique in that he wants his students to become independent and teaches more through questions rather than answers. Through his ways he has taught me to be mature technically as well as a person and has equipped me to be s stronger person to face the real world.

Dr Gokhale or Andy as we know him has been like both a friend and an advisor. I will always be grateful to him for believing in my thesis topic at a time when I was looking for funding sources. He spent countless hours discussing my problem area and ways in which to take it further to make it more relevant to current technologies. He has also been a great friend and his sense of humor being a great help in difficult times.

A great influence in my graduate student career has been that of Dr. Larry Dowdy. It was his course on Computer System Analysis which opened my mind to ideas and ways which helped me form my thesis topic. I have worked on mostly all my papers with him. Dr Dowdy helped me in focusing on the things that I want to do and teaching me immensely on techniques which I had no clue about. A whole new world of system analysis was unveiled to me through his teachings. The one to one sessions that I had with him helped me to become a better scholar and a person.

I also want to thank Dr. Gautam Biswas for advising me on a part of my thesis. I had spent a summer and another semester working with him. It was truly enjoyable and it taught me new ways of doing research.I would also like to thank Dr. Daniel Waddington with whom I worked in Lockheed Martin, Advanced Technology Lab, New Jersey for a year and half. He gave me new visions which helped me get my thesis direction.

All former and current DOC group members have been of great help. Nishanth and Kitty helped me initially. Jai has been a great friend. The frequent coffee sessions with Sumant, Akshay and Jai were always enjoyable. All of them greatly helped during good and bad times. ISIS has served as a great help always providing with new exposure through pizza lectures or advice from students in related areas.

I also want to mention about my wife Anindita who has provided tremendous support to me during all this difficult period. Many times she was the only audience to my frequent tales of frustration during the difficult days. She also has been very courageous in carving out a career of her own here in Nashville. Life in Nashville has been great and more so due to some great friends that I am lucky to have. Our bengali friends have been of great support and has been always there during dark days when entertainment and fun was badly required.

I would also like to thank my uncle Dr. Sumitra Deb and his wife Dr Swati Deb for always encouraging me to go on with my thesis. They are both professors and always urged me from my childhood days to work for higher education. Finally, my mother has been a great influence. I had heard stories of her PhD days when I was a kid and graduate studies was always something which I looked forward to. She had always been encouraging to my life as a grad student and was always there with advice whenever I was in trouble. My brother had been a great friend specially when things were not good and we used to spent long hours in conversation which helped me to get my perspectives right and carry on with my goal. My aunt, had been a great support always and has a great positive attitude. She always supported me with any decision that I have taken which has been of great help.

Thanks to you all for helping me to reach this point in my life.

ABSTRACT

Large scale distributed component based applications provide a number of different services to its clients. Such applications normally serve huge number of concurrent clients and need to provide a decent Quality of Service (QoS). A deployment domain composed of several machines are used to host these applications. The application components are distributed across the machines and communicate among themselves. An important objective of the owner of such a deployment will be to handle as much clients as possible during any given time which will obviously maximize the revenue earned. But this also needs to be done by keeping the costs down and also by providing every customer a minimum amount of QoS. The cost can be reduced by minimizing the number of machines used and using less power.

This thesis works towards a solution to the above and comes up with novel application component placement heuristics which makes sure that the overall resources of the domain is utilized in the best possible way. The intuition behind this work is that components are the smallest elements of an application from the perspective of resource usage. By distributing the components in a judicious way across the machines, it is possible to ensure that a minimum of resources is wasted. The work presented here uses a three phase strategy to come up with a solution. In the first phase, the component resource requirement is identified using profiling and workload modeling techniques. In the second phase detailed performance estimation of the application is carried out using analytical methods. In the third and final phase heuristics are proposed which uses the component resource requirement and the performance estimation methods to come up with placing the components across the machines. It ensures that such a placement will waste the least of resources.

In the final part of this work, it applies this work in the context of modern data center planning. The most important challenge in modern day data centers is to support large

customer bases with high expectation of performance. The incoming workload to the application is highly varying with periodic increase and decrease of workload. If resources are allocated for average workload then performance suffers during peak workload while planning for peak workload keeps resources idle during less workload. Cloud computing is an emerging trend which allows the elastic configuration of resources where machines can be acquired and released on the go. This work proposes a dynamic capacity planning framework for cost minimization based on a look ahead control algorithm which combines performance modeling, workload forecasting and cost optimization to plan for resource allocation in a dynamic environment. The results show how the resources can be allocated just-in-time with workload fluctuations. The dissertation also presents the various way resource is allocated as the various cost components change.

TABLE OF CONTENTS

Page

DEDICATION ii

ACKNOWLEDGMENTS iii

ABSTRACT v

LIST OF TABLES xi

LIST OF FIGURES xiii

Chapter

I. Introduction 1

I.1. Problem Area: SLA driven application management 1
I.2. Factors Affecting Performance 2
I.3. Solution Approach 4
I.4. Place Components To Balance Resource Usage 5
I.5. Component Resource Requirement 7
I.6. Performance Estimation 8
I.7. Requirements of Deployment Planning 9
I.8. Rice University Bidding System (RUBiS) - A Case Study 10

II. Related Research 13

II.1. What Is Missing ? 14

I Identifying Component Resource Requirement 16

III. Profiling Of Distributed Component Based Systems 18

III.1. Inserting Probes into OS Services 18
III.2. Microsoft Windows Performance Counters 21
III.3. Distributed System Monitoring 22
III.3.1. Monitoring of Component-Based Systems (MCBS) . . 23
III.3.2. OVATION 25
III.4. Virtual Machine Profiling 26
III.4.1. Virtual Machine Sampling 27
III.4.2. Profiling via VM Hooks 28
III.4.3. Bytecode Instrumentation 30
III.4.4. Aspect-Oriented Techniques used for Instrumentation . 32

III.5. Conclusion 36

IV. Framework For Monitoring & Profiling Distributed Component Based Applications 37

IV.1. The Design of the Bulls-Eye Target Manager 37
IV.1.1. Structure of Bulls-Eye 37
IV.1.2. Functionality of Bulls-Eye 39
IV.2. Resolving Bulls-Eye Design Challenges 41
IV.2.1. Challenge 1: Integrating Heterogeneous APIs of Multiple Platforms 41
IV.2.2. Challenge 2: Providing a Common Access Point to Provision Domain Resources 43
IV.2.3. Challenge 3: Presenting data to clients with bounded response time in uniform structure 44
IV.2.4. Challenge 4: Using Multiple Configurable Monitor Components to Extract Variety of Data 46
IV.3. Workload Modeling 47
IV.4. Conclusion 49

II Performance Estimation of Large Scale Distributed Component Based Systems 51

V. Analytical Models for Performance Estimation 53

V.0.1. Analytical Modeling of RUBiS Servlets 54
V.1. Challenges in Analytical Modeling of Multi-Tiered Applications 56
V.1.1. System Activity at Heavy Load 57
V.1.2. Multiprocessor effects 59
V.1.3. Dependent Transactions 59
V.1.4. Solution: Profile driven Regression based Extended Closed Queuing Network 65
V.2. Conclusion 75

VI. Modeling Software Contention Using Colored Petri Nets 78

VI.1. Model Driven Application Configuration 78
VI.2. Application Case Study: Target Tracking Simulator 80
VI.2.1. Overview of the Target Tracker 80
VI.2.2. Case Study Application Goals 82
VI.3. Experiments 84
VI.3.1. Application Profiling 84
VI.3.2. Colored Petri Net Model Construction 86
VI.3.3. Calibrating the Model 88
VI.3.4. Model Validation 88
VI.4. Application Configuration 91

VI.5. Related Work 94
VI.6. Concluding Remarks 95

VII. Modeling of Real Time Systems 104

VII.1. Real Time Systems 104
VII.2. Motivating Example 105
VII.3. Problem Formulation 106
VII.4. Modeling Approach 107
VII.4.1. Workload Modeling 107
VII.4.2. System Modeling 108
VII.5. A Complete Example 115
VII.6. Broader Methodology Benefits 116
VII.6.1. Sensitivity Analysis 117
VII.6.2. New Optimal Scheduling Algorithms 119
VII.7. Concluding Remarks 123

III Application Placement 125

VIII. Multi-Capacity Resource Allocation in Distributed Component Based Systems 127

VIII.1 Resource Allocation As A Bin Packing Problem 127
VIII.1.1 Point of Diminishing Returns for Running Resource Allocation Algorithms 132
VIII.2 Concluding Remarks 134

IX. Component Assignment Framework For QoS Assurance 137

IX.1. Problem Formulation and Requirements 137
IX.2. CAFe: A Component Assignment Framework for Multi-Tiered Web Portals 139
IX.2.1. Allocation Routine 139
IX.2.2. Algorithmic Framework to Co-ordinate Placement and Performance Estimation 141
IX.3. Experimental Evaluation 143
IX.3.1. Rice University Bidding System 143
IX.3.2. Computing Service Demand 144
IX.3.3. Customer Behavior Modeling Graph 146
IX.3.4. Analytical Modeling of RUBiS Servlets 147
IX.3.5. Application Component Placement 147
IX.3.6. Implementation of the CAFe Deployment Plan 150
IX.3.7. Algorithm for Component Replication and Placement . 152
IX.4. Evaluating The Replication And Placement Algorithm 157
IX.5. Conclusion 161

X. Modern Day Data Center 163
X.1. Introduction 163
X.2. Related Research 165
X.3. Solution Approach: On Demand Resource Provisioning Using Look-Ahead Optimization 169
X.4. Solution Details 173
X.4.1. Workload Prediction 173
X.4.2. Identifying resource requirement 173
X.4.3. Optimizing Resource Provisioning 174
X.5. Experimental Evaluation 176
X.5.1. Just in time resource allocation 177
X.5.2. Resource usage under various cost priorities 178
X.6. Conclusion 189

XI. Concluding Remarks 192

REFERENCES 195

LIST OF TABLES

Table		Page
1.	Optimization Problem To Maximize Clients	4
2.	The Dual Problem: Minimize Resources	4
3.	Transition Probabilities Between Various Services	48
4.	Machine Configurations Used	62
5.	Correction Factors for Various Services	73
6.	Profiled Data from the Application	85
7.	Model Predicted Data	89
8.	Target Hit Chances for Various Configurations	92
9.	Runtime Target Hit Occurrences	93
10.	A highly loaded system	105
11.	A lightly loaded system	106
12.	Erlang distribution parameters	108
13.	Task Parameters	110
14.	Task parameters	115
15.	Example performance metrics	116
16.	Task parameters	117
17.	Success Rate of Heuristics on Solvable Problems	132
18.	Component Names for Each Service	144
19.	CPU Service Demand for Each Component	146

20. Transition Probabilities Between Various Services 147
21. Component Placement and RUBiS Performance After Iteration 1 148
22. Iteration 2:Component Placement by Allocation Routine 149
23. Successive Iterations:Response Time of Each Service 149
24. Success Rate of Heuristics on Solvable Problems:Courtesy Chapter VIII . 155
25. Response Time and Utilization 159
26. Components of Cost Function 178
27. Summary Of Research Contributions 193
28. Summary of Publications 194

LIST OF FIGURES

Figure		Page
1.	Workload and Resource Affect Response Time	3
2.	Bottleneck Node:Typical Scenario	5
3.	Balanced Utilization: Accommodates Mode Clients	6
4.	Response Time Comparison	6
5.	RUBiS Architecture: Java Servlets Version	11
6.	Gap in current state of the art	15
7.	Process Systems calls "Intercepted" by Profiling Library	19
8.	MCBS Stubs and Skeletons are Instrumented with Probes	23
9.	Applications Running on Virtual Machine	26
10.	The Bulls-Eye Target Manager Architecture	38
11.	Using the Adapter Pattern in Bulls-Eye	42
12.	Providing a Common Access Point to Domain Resource Data	44
13.	Customer Behavior Modeling Graph of a Typical User	49
14.	Create Models of Application	54
15.	Closed Queuing Model for Rubis Java Servlets Version	55
16.	Validation of Analytical Model	56
17.	Comparison of Analytical vs Empirical Data	58
18.	Concurrent Overlapping Queries Have Similar Response Times	60
19.	Traditional Closed Queuing Model	61
20.	Additional Queue to Model Software Blocking	61

21. Model of Inter-Dependent Queries 63
22. Comparison of Analytical vs Empirical Data 64
23. Comparison of Analytical vs Empirical Data In Configuration 2 64
24. Overall Service Demand 68
25. Response Time in Single Processor Machine 76
26. Response Time in Multiple Processor Machine 76
27. Comparison of Empirical Correction Factor with Suri Proposed 77
28. Inverse of Correction Factor (CI) 77

29. Active Objects in Target Tracker 80
30. Application Logical Flows in the Target Tracking Simulator 97
31. CPN Model of Application Case Study 98
32. A CPN Model of Target's Active Object Thread 98
33. Contention Model for Software Lock 99
34. The CPN Model of CPU 99
35. The Formula for Cache Effects 99
36. Response Time of Target Thread with Locks 100
37. Throughput of Satellite Thread with Locks 100
38. Throughput of Location Thread with Locks 101
39. Throughput of Tracker Thread with Locks 101
40. Throughput of Location Thread without Locks 102
41. Response Time of Target Thread without Locks 102
42. Throughput of Satellite Thread without Locks 103

43. Task Representation 109
44. Task Arrivals 111

45. Task Execution/Deadline - EDF 113
46. A Simple Example using EDF scheduling 115
47. System with 83% Utilization 118
48. System with 67% Utilization 118
49. System with 47% Utilization 119
50. Utilization versus Variance 120
51. Optimal Algorithm (93% Util.) 120
52. Optimal Algorithm (93% Util.) 122
53. Optimal algorithm (83% Util.) 123
54. Distribution of Items (0-100 range) 130
55. Performance Comparison of Different Heuristics 131
56. The CAFe Component Assignment Framework Architecture 139
57. The Utilization of Memory and Disk for RUBiS Benchmark 145
58. Response Time with Increasing Clients 150
59. Deployment of CAFe Suggested Assignment 151
60. Performance of CAFe Installation 151
61. CPU Utilization 153
62. Queuing Model of RUBiS Scenario 158
63. Node Usage of Tiered and MAQ-PROWESS 159
64. Allocation of Components for 2,000 Client 160
65. Coefficent of Variation of Node Usage 161
66. Response Time for Tiered and MAQ-PROWESS 162
67. Client Population With Time 171

68. Just in time resource allocation with load 177

69. Resource Allocation for Low SLA Violation Cost and High Machine Cost 179

70. Resource Allocation for Medium SLA Violation Cost 180

71. Resource Allocation for High SLA Violation Cost 180

72. Resource Allocation for variety of systems 181

73. Resource Allocation for High SLA violation with low reconfiguration cost 184

74. Resource Allocation for High SLA violation with medium reconfiguration cost 185

75. Resource Allocation for High SLA violation with high reconfiguration cost 186

76. Resource Allocation for Medium SLA violation with low reconfiguration cost 187

77. Resource Allocation for Medium SLA violation with medium reconfiguration cost 187

78. Resource Allocation for Medium SLA violation with high reconfiguration cost 188

79. Low SLA violation, Medium Machine Cost and low reconfiguration cost 189

80. Low SLA violation, Medium Machine Cost and medium reconfiguration cost 190

81. Low SLA violation, Medium Machine Cost and high reconfiguration cost 190

82. Low SLA violation, Medium Machine Cost and very high reconfiguration cost 191

CHAPTER I

INTRODUCTION

I.1 Problem Area: SLA driven application management

Large scale component based distributed systems form the backbone of many applications in the modern computing world such as distributed data centers supporting large scale medical records processing or social networking applications; shipboard computing composed of numerous components distributed over nodes in a running ship and web portals that are multi-tier web applications supporting millions of users simultaneously.

In all of the above, the application is composed of a number of components that are distributed over numerous clusters of nodes. These clusters can be located in a single data center or distributed across several data centers located geographically apart. These applications also provide various types of services each of which are implemented with the use of separate set of components. In this chapter, an example of a web portal is used to motivate the challenges faced in deploying and maintaining such a large scale application.

Close scrutiny of the dynamics of such applications will reveal that at any instant in time a number of clients are logged onto the application, and each client is engaged in using one or more of the many diverse set of services provided. Individual clients expect these services to be *dependable* implying that clients must obtain acceptable response times and service availability in accordance with the service level agreement (SLA) despite fluctuations in system loads.

Service providers that own such applications will obviously want to maximize the amount of clients handled since that increases the utility of the application. For this, careful capacity planning must be carried out to handle the user base while meeting their SLAs. This challenge must be met without unduly increasing the cost of procuring and maintaining resources. Thus, minimizing resources and efficiently utilizing the resources is a key

objective. The above problem can be expressed in terms of an optimization problem as follows:

Maximize	Workload (W Clients)		
Subject To			
	Resources	<=	R Nodes
	Performance	>=	SLA Bound

The problem essentially states that a maximum number of clients need to be handled while ensuring that average performance characteristics is above some bound agreed in an SLA. All this needs to be done while using minimum hardware resources. This is a tricky problem since both clients and resources affect the performance of the system. It is thus important that the optimal operating point needs to be selected which enables a set of given resources to handle the maximum workload while maintaining the performance bounds. This is explained in more detail in the next section.

I.2 Factors Affecting Performance

Workload quantity and available resource both affect the performance of the application. Figure 1 shows how response time which is a performance characteristic varies with workload (clients) and with resources.

The SLA for the response time is 100 msec shown by the horizontal line in Figure 1. The objective of an administrator will be to handle as many clients as possible while maintaining the average response time within 100 msec. So the operating point of the application will be made as close as possible to the SLA value. This occurs with a little above 1000 clients when there are high resources and around 800 with medium resources and 575 with low resources. Thus its clear that number of clients is directly proportional to resources.

The job of an administrator will be to balance the cost of the resources with the revenue from clients so that the overall utility of the application is maximized.

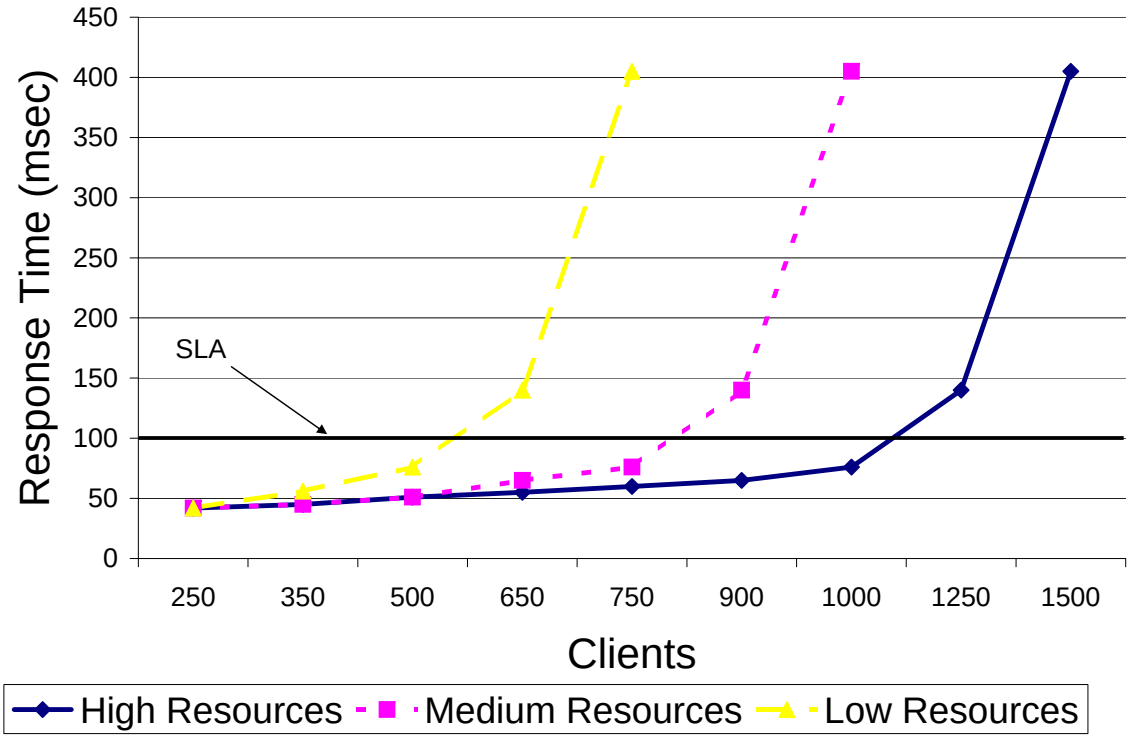


Figure 1: Workload and Resource Affect Response Time

Over and above workload and resources there are also other factors that affect the performance of an application. They are explained below

- *Workload Mix*: Each client request can be different from the other. Each type of service is composed of different set of components. A client making a call on a service will load those components that make up the service. So the overall performance of the application will depend on how many calls of each type of service is made.

- *Think Time or Period*: In case of an interactive system or for periodic calls, the think time or period of the jobs also will affect performance. The distribution of the think time may also matter.

- *Arrival Distribution*: In a system which has a regular flow of arrival requests, the distribution of arrival requests will also determine the performance of the application.

- *Faults*: There could also be both hardware and software faults. These will also affect the performance of an application.

It can be seen that there are a number of factors that affect the performance of an application. Its important to understand all of these in order to design a reliable, dependable application which assures Quality of service (QoS) parameters like response time.

I.3 Solution Approach

The above problem of maximizing the utility of an application by maximizing the number of clients handled and by minimizing the amount of resources required can be posed as a optimization problem(Table 1).

Maximize	Workload (W Clients)		
Subject To			
	Resources	<=	R Nodes
	Performance	>=	SLA Bound

Table 1: Optimization Problem To Maximize Clients

The problem can also be expressed in terms of the dual equivalent problem of the above which is given by Table 2.

Minimize	Resources (Nodes)		
Subject To			
	Workload	>=	W Clients
	Performance	>=	SLA Bound

Table 2: The Dual Problem: Minimize Resources

The next few sections detail strategies to come up with solutions for the above problem.

I.4 Place Components To Balance Resource Usage

In order to solve either of the above problems there is the need to quantify the main factors 1) workload, 2) resources and 3) performance. An example scenario in a distributed application is presented in Figure 2. Here the processor on the node named "App Server" is saturated with near 100% utilization. So if clients are increased then there will be a steep increase in the response time of the clients. Thus although there is some spare processing power on the other nodes the system cannot accommodate further clients. Figure 3 shows an alternate scenario where the utilizations are better balanced among the three machines and none of them are fully utilized this makes it possible to accommodate more clients. Thus using the same number of resources, its possible to increase the client size by balancing resource utilizations.

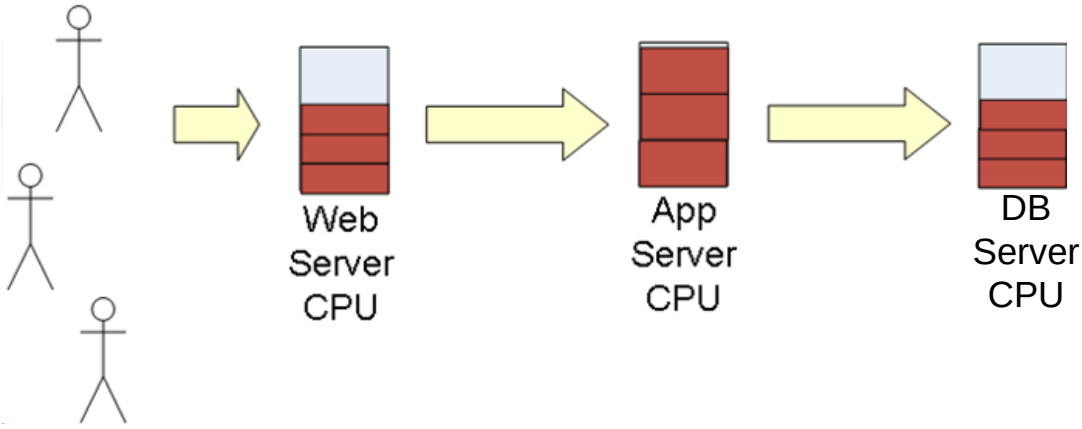


Figure 2: Bottleneck Node:Typical Scenario

Figure 4 compares the response times for the two systems. The response time is measured with increasing clients. The SLA bound for response time is assumed to be 100 msec. It can be seen that the system with the balanced resource reaches a response time of 100 msec much later than the imbalanced system. Thus it can be seen that the resources can be much better utilized if the utilization levels can be balanced across the nodes.

The applications that are considered in this work are composed of multiple components.

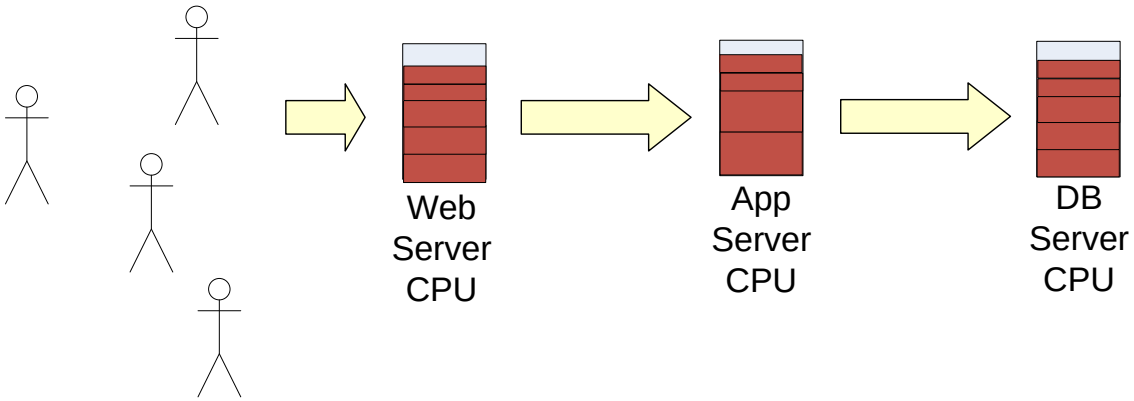


Figure 3: Balanced Utilization: Accommodates Mode Clients

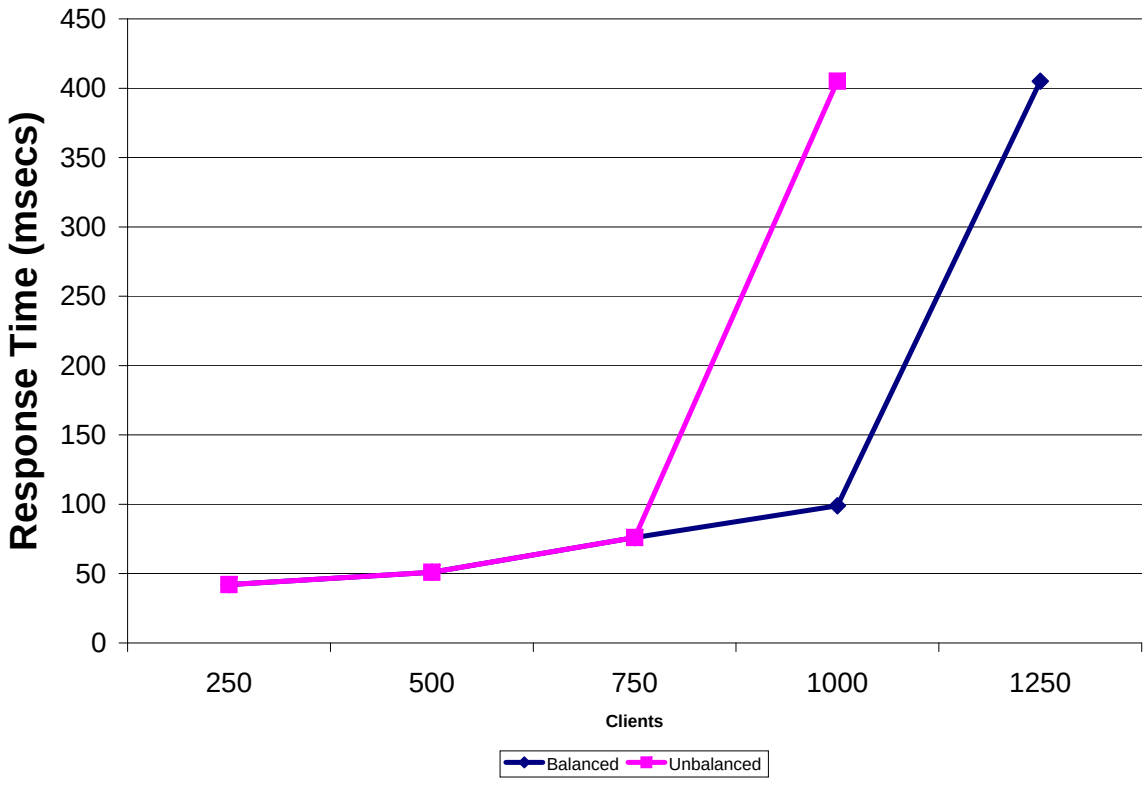


Figure 4: Response Time Comparison

Each basic functionality (such as a business logic or a database query) is wrapped within a component, such as a Java class. Several components are then composed together to implement a single functionality, such as placing bids in an auction site or booking air tickets in a travel site. Each component consumes an amount of resource. Since a component is the smallest part of the application, its easier to allocate the components to the nodes and achieve more balanced utilization across the nodes. Compare this to a tier-based development where each tier provides a number of functionalities and need to be collocated in the same machine. Obviously it will be difficult to balance load with larger tiers.

The above explanation will be clearer when thought in the context of a bin-packing problem. Lets assume that there are a number of items each having a finite size. These need to be packed onto bins which have a finite capacity. Its much easier to pack in smaller items onto the bins and ensure that the bins are balanced than having larger items. Thus the smaller granularity of components help in balancing the utilization of the nodes in the application. Using this insight as a central theme, the further challenges to coming up with the solution is pursued in the following sections.

I.5 Component Resource Requirement

As discussed in the previous section, the components help in balancing the resource usages on the different nodes. But before that can be done, it is required to find out the resource requirement of each component. Each component can have multiple resource requirements like cpu, memory, disk and network. The base resource requirement of the component is the amount of resource taken for one call and can be found by profiling. But the overall resource requirement of a component also depends on the workload of the application. For example if there are an average of 100 calls on the component, it will require more resource than when there are 50 calls. So it is important to find out the average number of incoming calls for each component. Once this can be done, the average resource requirement of each component can be computed.

In order to quantify the number of incoming calls on a component there is the need to model the workload to the application. In an web application, there could be different types of calls such as calls to the 'Home' page, to the 'Search Items' page, to the selling page or to the bidding page. Using historical data the number of hits to each type of page can be computed and an estimate for the number of calls to a particular component can be made. This can be used to compute the total resource requirement for each component. But note that this will be specific to a particular workload study. In case of a different/new installation the requirements will vary. Thus the resource requirement of each component can be found in two steps first by profiling the component and then by modeling the workload.

I.6 Performance Estimation

As mentioned in the earlier sections, the resource requirement of the components can be computed and can be used to place the components onto the available nodes. The system objective also requires that the performance of the application should be within a particular threshold. Thus for any kind of placement of the components, the performance of the application should also be verified to be within the given SLA bounds. This can be done by creating a performance model of the application. Such a model should be simple enough so that it can be run as part of an algorithm for placing the application components. On the other hand, it should also be aware of a few important things as given below:

- Model Resource Contention: Since multiple components are required to be placed in the same node. The components will contend for resources. Thus a performance model should take this contention into account and model the waiting time that a job may suffer due to contention.

- Component Aware: Since the whole solution approach is based on placing components onto different nodes, its important that the performance models are aware of components. Thus when a component is moved from one node to another, the models should be able to be modified with the least number of steps.

I.7 Requirements of Deployment Planning

The earlier sections discussed the problems of maximizing utility for a large scale distributed component based application. Utility is expressed as the difference between revenue from clients and cost of resources. The motivation of an owner/administrator of such an application will be to maximize this utility. This can be done by increasing the clients while keeping the resources constant or by minimizing the amount of resources required for a given set of clients. One way to ensure this is to make sure that resource wastage is minimized. By placing components across the nodes in an intelligent manner such that resource usage is balanced across machines, this can be achieved. The requirement of such a solution has been identified in the earlier sections and they are broadly in 3 areas. The individual areas along with their specific requirement is given below:

1. Component Resource Requirement
    - Component Profiling
    - Workload Modeling
2. Performance Estimation
    - Model Resource Contention
    - Component Aware
3. Component Placement Algorithm
    - Respect SLA bound
    - Balance Resource Utilization

The next section discusses a case study which will be used as a running example to highlight many of the problems and solutions.

I.8 Rice University Bidding System (RUBiS) - A Case Study

This section describes a case study application for large scale distributed component based application. This application is then used as a representative application in the rest of the chapter to showcase many aspects of the overall problem.

RUBiS [3] is a prototype of an auction site modeled after ebay that has the features of an online web portal. It implements the core functionality of an auction site: selling, browsing and bidding. It does not implement complementary services like instant messaging or newsgroups. Three kinds of user sessions: visitor, buyer, and seller are distinguished. For a visitor session, users need not register but are only allowed to browse. Buyer and seller sessions require registration. In addition to the functionality provided during visitor sessions, during a buyer session users can bid on items and consult a summary of their current bids, rating and comments left by other users. Seller sessions require a fee before a user is allowed to put up an item for sale. An auction starts immediately and lasts typically for no more than a week. The seller can specify a reserve (minimum) price for an item. RUBiS is a free, open source initiative. Several versions of RUBiS are implemented using three different technologies: PHP, Java servlets and EJB (Enterprise Java Bean).

RUBiS can be used from a web browser for testing purposes or with the provided benchmarking tool. A client is designed that emulates users behavior for various workload patterns and provides statistics.

The auction site defines 26 interactions that can be performed from the client's Web browser. Among the most important ones are browsing items by category or region, bidding, buying or selling items, leaving comments on other users and consulting one's own user page (known as myEbay on eBay). Browsing items also includes consulting the bid history and the sellerŠs information. The two workload mixes are: a browsing mix made up of only read-only interactions and a bidding mix that includes 15% read-write interactions. The bidding mix is the most representative of an auction site workload.

A client-browser emulator is designed. A session is a sequence of interactions for the

same customer. For each customer session, the client emulator opens a persistent HTTP connection to the Web server and closes it at the end of the session. Each emulated client waits for a certain think time before initiating the next interaction. The next interaction is determined by a state transition matrix that specifies the probability to go from one interaction to another one.

The think time and session time for all benchmarks are generated from a negative exponential distribution with a mean of 7 seconds and 15 minutes, respectively. The load on the site is varied by varying the number of clients.

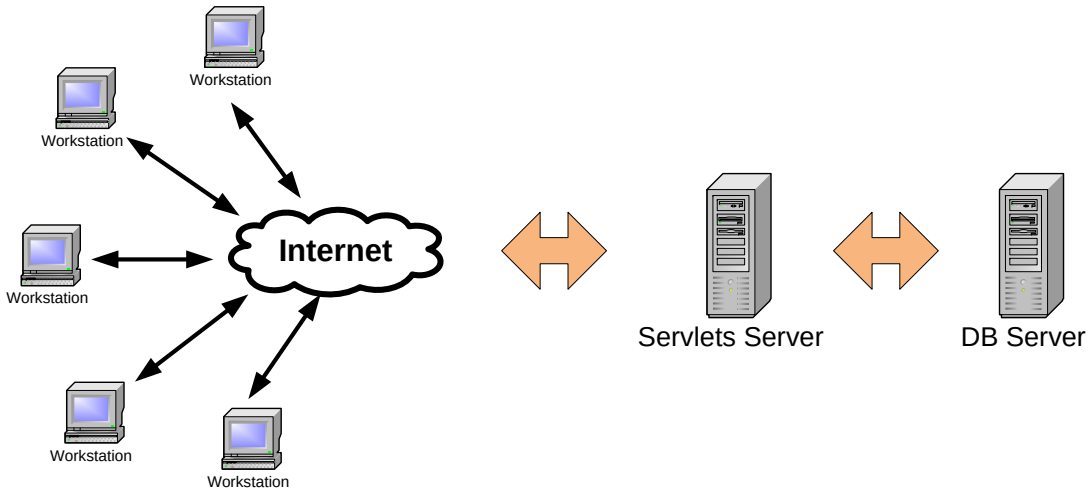


Figure 5: RUBiS Architecture: Java Servlets Version

A MySQL database is used that contains 7 tables: users, items, categories, regions, bids, buy_now and comments. The users table records contain the user's name, nickname, password, region, rating and balance. Besides the category and the seller's nickname, the items table contains the name that briefly describes the item and a more extensive description, usually an HTML file. Every bid is stored in the bids table, which includes the seller, the bid, and a max_bid value used by the proxy bidder (a tool that bids automatically on behalf of a user). Items that are directly bought without any auction are stored in the buy_now

table. The comments table records comments from one user about another. As an optimization, the number of bids and the amount of the current maximum bid are stored with each item to prevent many expensive lookups of the bids table. This redundant information is necessary to keep an acceptable response time for browsing requests. As users only browse and bid on items that are currently for sale, the item table is splitted into a new and an old item table. The very vast majority of the requests access the new items table, thus considerably reducing the working set used by the database.

The system sizing is done according to some observations found on the eBay Web site. There is always about 33,000 items for sale, distributed among eBayŠs 20 categories and 62 regions. A history of 500,000 auctions in the old-items table is there. There is an average of 10 bids per item, or 330,000 entries in the bids table. The buy_now table is small, because less than 10% of the items are sold without auction. The users table has 1 million entries. It is assumed that users give feedback (comments) for 95% of the transactions. The comments table contains about 506,500 comments refering either to items or old items. The total size of the database, including indices, is 1.4GB.

The next chapter will discuss related work within the context of the requirements identified in Section II.1 for deployment planning of large scale distributed applications for maximizing utility.

CHAPTER II

RELATED RESEARCH

This chapter discusses the related work in the research literature in the context of the main areas of focus that were identified in section I.7. Various work in the different areas such as 1) analytical techniques for performance estimation, 2) profile driven techniques for system management and 3) application placement techniques using optimization are presented here. These are compared against the requirements that have been described in section I.7.

Analytical Techniques: A large body of work on analytical techniques to model and estimate the performance of multi-tiered internet applications exists. For example, [52, 75, 77, 84, 85] use closed queuing networks to model multi-tiered internet applications. These efforts typically model an entire tier as a queue. Such models are also usually service-aware. This allows system management decisions involving components and services to be implemented. These efforts also typically model an entire tier as a queue.

Profile-based Techniques: Stewart et. al. [70] propose a profile-driven performance model for cluster based multi-component online services. They use this model to perform system management and implement component placement across nodes in the cluster. Profiling is nicely used to find the resource required by each class of service. It is found that the resource requirement increases linearly with client population. An M/G/1 queue is used to represent each server and all the different classes of services are approximated by one class of service. This overall service has a service time equal to the sum of the service times of the different services. A problem with this kind approach is that performance characteristics of individual services like throughput and device utilzation cannot be computed and thus cannot be used for system management.

Application Placement Techniques: Karve et al. [30] and Kimbrel et. al. [34] present a

framework for dynamic placement of clustered web applications. Their approach considers multiple resources, some being load dependent while others are load independent. An optimization problem is solved which attempts to alter the component placement at runtime when some external event occurs. Components are migrated to respond to external demands. A simple model calculates service demands of different requests to characterize resource requirements of components. Carrera et al. [14] design a similar system but they also provide utility functions of applications mapping CPU resource allocation to the performance of an application relative to its objective. Urgaonkar et. al. [76] identify resource needs of application capsules (components) by profiling them and using this information to characterise the application's Quality of Service (QoS) requirements. They also propose an algorithm for mapping the application capsules onto the platforms (nodes).

All of the work above uses a performance estimation mechanism that do not take into account resource contention among different classes of requests. Resource contention and the waiting time accrued due to that accounts for a large part of the response time of a request. This is more so important at high utilization. Thus there is the need to use some kind of technique which will quantify the wait time due to resource contention. None of the prior works above (except [77]) enforces explicit performance bounds.

II.1 What Is Missing ?

This section outlines the missing link in the related research that is discussed above in the context of the research focus areas given in section I.7.

Figure 6 shows the gap in the current state of the art. There is the need for component placement techniques that use performance estimation methods which are component aware as also should take into account resource contention among various jobs. From the above analysis three basic research challenges are identified and are enumerated as below,

1. *Component Resource Requirement* needs to find out the resource requriement of each

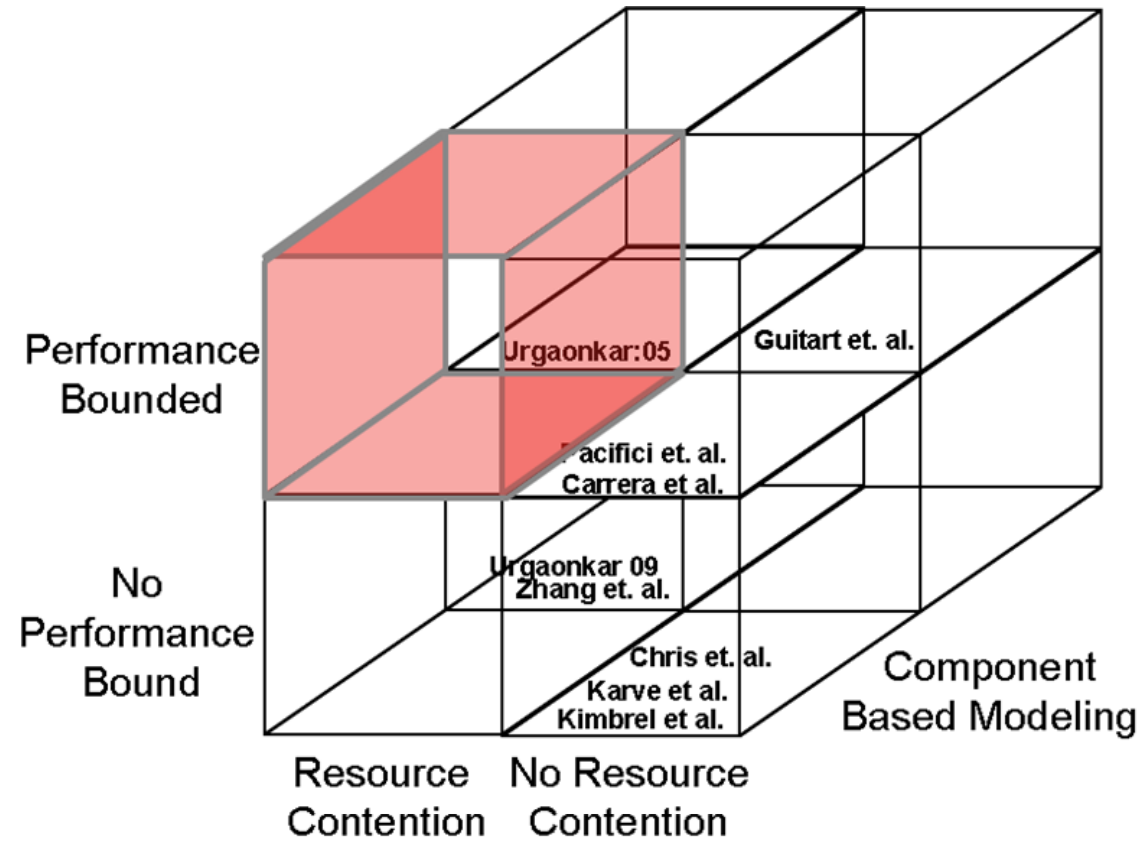


Figure 6: Gap in current state of the art

component. It also needs to model the workload so that overall component needs can be identified.

2. *Performance Estimation* needs to predict the performance characteristic of the application under various environment and workload. It has to take into account resource contention among the various jobs and also should be component aware.

3. *Component Placement* involves the assigning of the individual components onto the nodes such that the overall performance of the application is within given bounds. It will also ensure that the utility of the application is maximized by reducing cost and increasing workload.

Part I

Identifying Component Resource Requirement

This part of the dissertation deals with the issues in finding out the resource requirement of application components. Chapter III discusses various methods of profiling that are used and the pros and cons of each. Chapter IV discusses a profiling and monitoring framework for distributed component based systems. This chapter also discusses workload modeling concepts which helps in identifying the average number of hits to each component which in turns helps in estimating the overall resource requirement of a component.

CHAPTER III

PROFILING OF DISTRIBUTED COMPONENT BASED SYSTEMS

All applications rely upon services provided by the underlying Operating System(OS). These services are primarily used to coordinate access to shared resources within the system. To measure service "requests" probes can be placed directly within the OS code that can record individual application access to provided services. Many COTS operating systems also provide a number of performance counters that explicitly track usage of shared resources. Data generated from these counters along with data from embedded probes can be usefully combined to form a more complete picture of application behavior. Another common form of shared processing infrastructure is distributed computing middleware, such as OMG's CORBA and Microsoft's .NET, which provide common services, such as location transparency and concurrency management. Distributed computing middleware often provides a number of "hook points" that are accessible to users. These hooks provide placeholders for adding probe functionality that can be used to measure events typically hidden deeper within the middleware. This chapter first discusses techniques that can be used to place probes into OS services and how one can combine this information with data generated from OS-performance counters. The general practices of distributed-application monitoring using the services available in distributed computing middleware are discussed after that.

III.1 Inserting Probes into OS Services

A typical OS process contains one or more threads and a shared memory space. Application code that is executed by threads within a process is free to access various OS resources and services, such as virtual memory, files, and network devices. Access to these resources and services is facilitated through APIs that are provided by system libraries.

Each thread in the system executes in either user space or kernel space, depending upon the work that it is doing at that given time. Whenever a thread makes a system call, it transitions (e.g., via a trap) into kernel mode (Soloman, 1998; Beck, et al., 1999). Calls to system calls for thread management (e.g., thread creation, suspension, or termination) and synchronization (e.g., mutex or semaphore acquisition) will often incur a transition to kernel mode. System-call transitioning code therefore provides a useful interception point at which process activity can be monitored and a profile of system resource use can be extracted on a per-thread basis. Figure 7 shows the use of this so-called "inter-positioning" technique where libraries are built that mimic the system API. These libraries contain code that records a call event and then forward calls to the underlying system library.

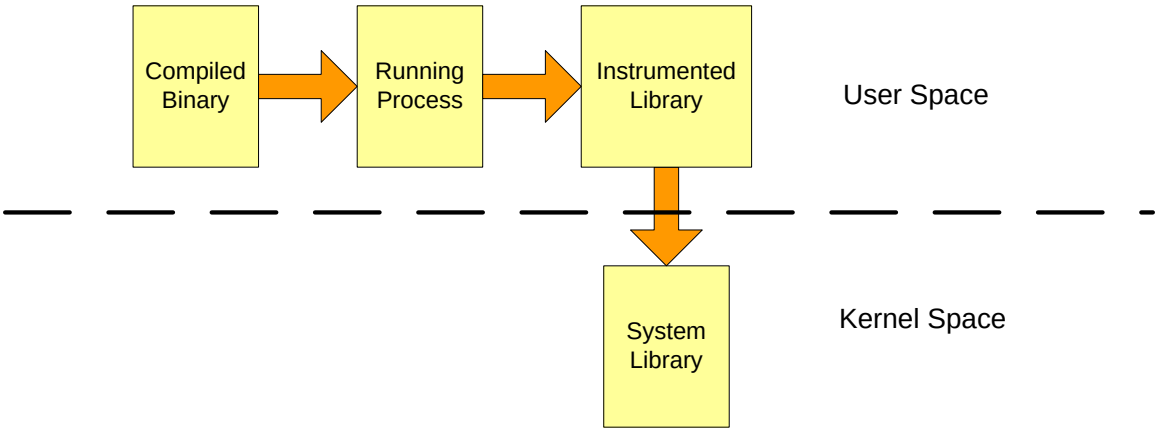


Figure 7: Process Systems calls "Intercepted" by Profiling Library

The threadmon [12] tool uses inter-positioning to insert trace code between the user-level threads library and the application by redefining many of the functions that the library uses internally to change thread state. This technique is also used by VPPB [9] to gather user-level thread information. In both approaches, data obtained by user library inter-positioning is integrated with data collected from other OS services, such as the UNIX /proc file system or kstat utility. The threadmon and VPPB tools both target the Solaris

OS and therefore rely upon Solaris-specific system utilities, such as memory mapping of /dev/kmem to access the kernel.

Cantrill et al. [12] and Broberg et. al. [9] have also used another tool known as Trace Normal Form (TNF) [50]. This tool generates execution event traces from the Solaris kernel and user processes. Solaris provides an API for inserting TNF probes into the source code of any C/C++ program. A TNF probe is a parameterized macro that records argument values. The code excerpt below shows how C macro probes can be inserted at the beginning and end of critical code to record the absolute (wall-clock) time required for the code to execute.

```
#include <tnf/probe.h>
.
.
extern mutex_t list_mutex;
.
.
TNF_PROBE_1(critical_start, "critical section start",
"mutex acquire", tnf_opaque, list_lock, &list_mutex)

mutex_lock(&list_mutex);
.
.
/* critical section code */
.
.
mutex_unlock(&list_mutex);
```

```
TNF_PROBE_1(critical_end, "critical section end", "mutex release",
tnf_opaque, list_lock, &list_mutex)
```

These probes can be selectively activated dynamically at run time. Events are recorded each time a probe is executed. Each probe automatically records thread-specific information, such as the thread identifier, but it may also record other data related to the state of the application at the time the event was triggered. Event records are written to a binary file that is subsequently parsed and analyzed by an offline process. The Solaris kernel also contains a number of TNF probes that can record kernel activity, such as system calls, I/O operations, and thread state change. These probes can be enabled/disabled using a command line utility known as prex [50]. Data records from the probes are accumulated within a contiguous portion of the kernel's virtual- address space and cannot be viewed directly. Another utility that runs with administrator privileges can be used to extract the data and write it to a user file. This data can then be correlated with other user-level data to provide a clear understanding of the behavior of the application run. The probe-based technique described above provides a detailed view of the running state of the application. Behavioral data details call counts, timing information, and resource use (thread and system state). There are some drawbacks to this type of approach, however, including: " The solution is not portable because it depends on Solaris features that are not available on other operating systems. " It requires a considerable amount of development effort because thread libraries must be modified. " Applications must be separately built and linked for profiling. " Tools that are used to collect the data like TNF or kstat may require lengthy setup and configuration.

III.2 Microsoft Windows Performance Counters

Other operating systems have comparable features that can be used to get comparable data-defining application behavior. For example, Microsoft Windows provides performance counters that contain data associated to the running system. Windows provides a

console that can be used to select certain specific counters related to specific processes. Once selected, the values of these counters will be displayed on the console at regular intervals. Table 5 shows example counters that are available.

Table 5. Performance Counters Provided by the Windows Operating System Windows stores the collected values in the registry, which is refreshed periodically. Developers typically retrieve the data from the registry directly or use an API known as Performance Data Helper. Alternatively, the Microsoft .NET framework provides the System.Diagnostics namespace that facilitates access to all the counters from within a .NET application. Windows performance counters can be used to acquire data related to the running system, which can be correlated with a particular application run. These counters give an external view of the application, however, and there is no straightforward method of mapping counter values to logical application events. To more closely inspect a running application, therefore, instrumentation is needed within the application itself to record logical events and combine them with data generated through performance counters.

III.3 Distributed System Monitoring

A distributed application consists of components spread over a network of hosts that work together to provide the overarching functionality. The complexity of distributed applications is often considerably greater than a stand-alone application. In particular, distributed applications must address a number of inherent complexities such as latency, causal ordering, reliability, load balancing, and optimal component placement, that are either absent from (or less complicated in) stand-alone applications. The analysis and profiling of distributed applications involves monitoring key interactions and their characteristics along with localized functionality occurring within each component.

III.3.1 Monitoring of Component-Based Systems (MCBS)

MCBS [40] is a middleware-based monitoring framework that can be used to capture application semantics, timing latency, and shared resource usage. The MCBS approach recreates call sequences across remote interfaces. Probes are instrumented automatically through the Interface Description Language (IDL) compiler, which directly modifies the generated stubs and skeletons that record function entry and return, as shown in Figure 8.

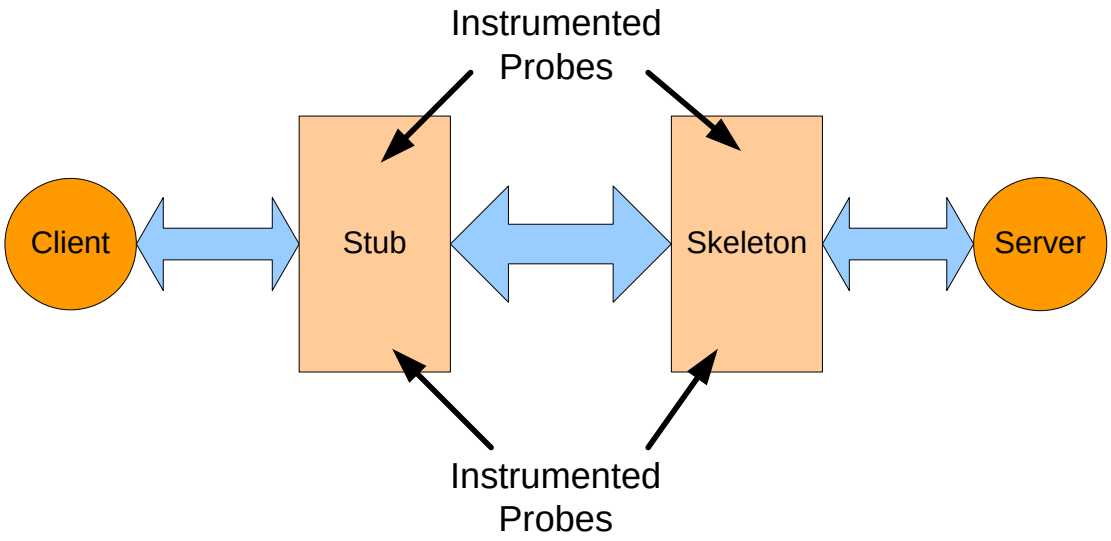


Figure 8: MCBS Stubs and Skeletons are Instrumented with Probes

Along with tracking interactions, the MCBS-modified stubs and skeletons also record all transactions (as a call sequence), parameters (e.g., in, in/out), and return values. Event data is recorded to a log and a unique identifier assigned so that the scenario/call chain can be identified later. This identifier is generated at the start probe and is propagated through the calling sequence via thread-specific storage [23]. When each new interface is invoked, the stub receives the identifier from the thread-specific storage, creates a record with it, and stores a number identifying its position in the call chain. After control returns to the caller stub, the last record is generated. Hence, the call chain is recorded and mapped across threads. Whenever a new thread is created by the user application code the parent thread

identifier is stored along with the new thread identifier, which helps identify the actual call chain in cases where threads are spawned by user-application code (i.e., by tracking parental hierarchy). Event data is stored in a memory buffer during application execution and is dumped to a file regularly as the buffer becomes full. An offline data collector picks up the different files for the different processes and loads it in a database. The analyzer component processes the data in the database and forms the entire call graph. The end-to-end timing latency of call scenarios is measured by noting the time at the client stub when the call is made and again when the call returns from the server. The latency is measured from the difference of these two timestamps. The overhead due to the interference of these probes is measured and tabulated against normal non-instrumented operation [40]. Table 6 shows performance data for a sample application. The sample scenarios are known to have deterministic functionality, i.e., they perform the same set of actions every time they run, so multiple system runs can be compared together. To minimize measurement overhead, only specific components of the application are monitored. This selection process can be done in two ways: " Statically prior to executing, where monitored components are selected and the application is then run. The application must be stopped and restarted if the selected set of components changes. " Dynamically while the application is running, where the monitored components can be selected at runtime. Dynamic selection helps developers focus on problem area and analyze it without incurring overhead due to measurement of other components. Table 6. Overhead of Instrumentation Due to Probes Inserted in Stubs and Skeletons [40] has implemented both approach and shown that static selection is less complex than dynamic selection because dynamic selection needs to take care of data inconsistency, which can occur if a component process receives an off event (whereby monitoring is stopped) while it runs. In this case, selection must be deferred until the system reaches a steady state. Steady state will vary from application to application and is thus hard to identify in generically.

III.3.2 OVATION

OVATION is a distributed monitoring framework that uses similar concepts as the MCBS framework, but it is targeted for CORBA middleware, such as TAO [64] and JacORB [10]. It therefore uses CORBA Portable Interceptors to insert probes. Portable Interceptors are based on the Interceptor pattern [23], which allows transparent addition of services to a framework and automatic triggering of these services when certain events occur. Whenever a CORBA client calls a server component, therefore, client stub and server skeleton interceptors are invoked, which record the event. These interception points also occur in the reverse order when the call returns from server to client. OVATION thus provides a more transparent and standard than the non-standard way of inserting probes into the stub and skeleton code used by MCBS. OVATION provides a number of pre-defined probes, such as: " Snooper Probe, which captures the CORBA end-to-end invocation path. Provides such things as request name, arguments, request start time, end time and the threads and the processes to which it belongs. " Milestone Probe, which permits the manual demarcation of specific events in the application code. " Trace Probe, which is used to capture information about the other non-CORBA, C++ object method calls. It also allows users to add their own probes to the monitoring framework, thereby allowing application developers to monitor certain application-specific characteristic without changing their source code. OVATION also provides mechanisms that allow applications to enable or disable these probes transparently. OVATION re-creates the dynamic-system-call graph among components in a given scenario, along with latency measurement. OVATION generates log files during program execution that contain information detailing processes, threads and objects involved in the interaction. The OVATION visualizer transforms the log file into a graphical representation of the recorded remote object interactions. The distributed-system monitoring tools described above help monitor distributed application behavior by re-creating call graphs along with latency information for application traces. This capability is helpful because it simplifies the generation and collection of distributed application

data so it can analyzed in detail. For example, [40] describes how shared-resource-use data, such as processor consumption or heap-memory use in a particular application run, can help measure performance, devise capacity plans, and guide component-placement algorithms. One disadvantage of these tools is that they cannot follow local procedure calls, so a developer cannot track local application events. It may therefore be necessary to combine distributed monitoring tools with other application profiling tools, such as threadmon described in Section III.1. As a result, the entire profiling process may become unduly complicated because both sets of tools must be configured in a single run and there may be subtle interdependencies, such as conflicts in thread-library instrumentation used by local techniques, such as threadmon [12] and MCBS [40].

III.4 Virtual Machine Profiling

With the rebirth of virtual machines (VMs), such as the Java Virtual Machine and the Microsoft Common Runtime Language (CLR), the paradigm of application development and deployment has changed from the traditional architecture of applications interacting directly with the underlying OS. Figure 9 illustrates a typical VM-based application architecture where each user application is layered above the VM.

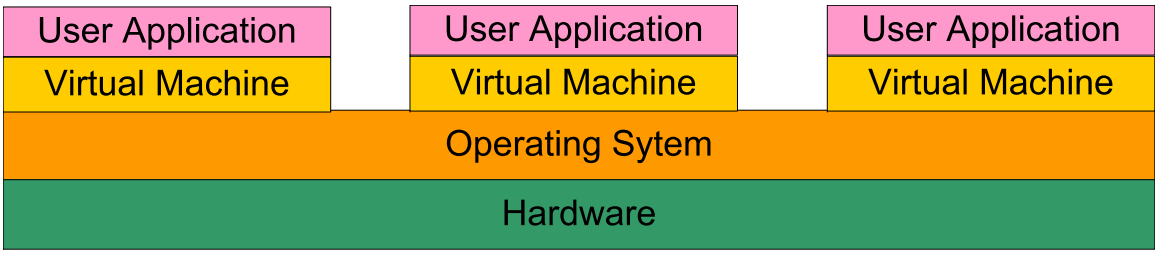


Figure 9: Applications Running on Virtual Machine

The use of VMs to run managed programs helps make profiling more portable. For example, techniques like dynamic instrumentation of complete binaries (discussed in Section 3) for VM platforms is more straightforward because the application runs within the

context of the VM and is thus easier to access and profile, and generated bytecode [25] is standardized, portable, and more straightforward to instrument. In general, profiling techniques, such as sampling and instrumentation, remain the same in a VM. There are a number of ways, however, in which these techniques can be used. The main factors that affect a particular method include (1) implementation difficulty, (2) overhead incurred, and (3) level of detail in the output. This section describes different methods used for VM profiling and compares and contrasts the pros and cons of each.

III.4.1 Virtual Machine Sampling

Sampling techniques typically result in less overhead than instrumentation techniques because sampling involves recording the program-counter value at regular (usually fixed) intervals, rather than using embedded code snippets into the application program at various points to record events. The corresponding program counter values (i.e., memory addresses) associated with each method are identified a priori and stored in a search structure. When the application runs on the VM, the program counter for each thread is recorded at regular intervals and stored with a processor timestamp. Once profiling is complete, the recorded data is analyzed, the number of invocations on each method counted, and the time spent executing each call calculated. Despite being relatively lightweight, however, sampling-based profiling methods are susceptible to certain problems [71], including: " Straddling effect of counters - the initial analysis to segregate the bytecode for different methods will be approximate, causing inconsistent results. " Short sub-methods - short-lived calls that take less time than the sampling frequency may not be recorded at all. " Resonance effects - the time to complete a single iteration of a loop can coincide with the sampling period, which may sample the same point each time, while other sections are never measured. These problems can be avoided by using techniques described in [71]. To obtain a consistent picture of application behavior, however, a significant number of runs must be performed. This number will again vary from application to application, so

the sampling period also may need to be configured for a particular application. Another method of sampling is described by [7]. This approach does not check the program counter at regular intervals. Instead, a snapshot of the call stack is recorded by each thread after a certain number of bytecodes are executed. The motivation for this approach is that bytecode counting is a platform-independent method of resource accounting [5, 6]. Bytecode counting can also be done without relying on more low-level, platform-dependent, utilities to acquire resource usage data, which make it more portable and easier to maintain. The work in [7] is an example of bytecode counting implemented by statically instrumenting bytecode.

III.4.2 Profiling via VM Hooks

A VM hook is a previously defined event, such as method entry/exit or thread start/stop, that can occur within the context of a running application. The profiling agent implements callback methods on the profiling interface and registers them with VM hooks. The VM then detects the events and invokes the corresponding callback method when these events occur in the application. It is straightforward to develop profilers based on VM hooks because profiler developers need only implement an interface provided by the VM and need not worry about the complications that can arise due to interfering with the running application. Although the VM and profiling agent provide the monitoring infrastructure, profiler developers are responsible for certain tasks, such as synchronization. For example, multithreaded applications can fire multiple instances of the same event simultaneously, which will invoke the same callback method on the same instance of the profiling agent. Callbacks must therefore be made reentrant via synchronization mechanisms, such as mutexes, so the profiler internal state is not compromised. The Microsoft Common Language Runtime (CLR) profiler and the Java Virtual Machine Tool Interface (JVMTI) are examples of VM profilers that that support VM hooks, as discussed below.

III.4.2.1 CLR Profiler

The CLR Profiler interface allows the integration of custom profiling functionality provided in the form of a pluggable dynamic link library, written in a native language like C or C++. The plug-in module, termed the agent, accesses profiling services of the CLR via the ICorProfilerInfo2 interface. The agent must also provide an implementation of ICorProfilerCallback2 so that the CLR can call back the agent to indicate the occurrence of events in the context of the profiled application. At startup, the CLR initializes on the agent, which configures the CLR and establishes which events are of interest to the agent. When an event occurs, the CLR calls the corresponding method on the ICorProfilerCallback2 interface. The agent then collects the running state of the application by calling methods on ICorProfilerInfo2. In between processing function enter/exit call-backs, the profiling agent requests a stack snapshot so that it can identify the fully qualified method name and also the parent (i.e., the method from which the method being traced was call) of the call. Inspecting the stack to determine parental methods (and ultimately the call-chain) is a useful technique for disambiguating system calls. For example, this approach can be used to disambiguate different lock calls so that per-lock information (e.g., hold and wait times) can be correlated with different call sites in the source code.

III.4.2.2 JVMTI Profiler

The JVMTI is similar to the CLR Profiler Interface in that it requires a plug-in, which is implemented as a dynamic link library using a native language that supports C. The JVM interacts with the agent through JVMTI functions, such as Agent_OnLoad(JavaVM *vm, char *options, void *reserved) and Agent_OnUnload(JavaVM *vm), which are exported by the agent. The JVM supplies a pointer, via the Agent_Onload() call, that the agent can use to get an instance of the JVMTI environment. The agent can use this pointer to access JVMTI features, such as reading the state of a thread, stopping/interrupting threads, obtaining a stack trace of a thread, or reading local variable information. The agent uses the

SetEventCallbacks() method to pass a set of function pointers for different events it is interested. When events occur, the corresponding function is called by the JVM, which allows the agent to record the state of the application. The CLR and JVMTI profilers share many common features, such as events related to methods or threads and stack tracing ability. There are differences, however, e.g., the JVMTI provides application-specific details, such as the method name, object name, class name, and parameters, from the calls, whereas the CLR interface provides them in a metadata format and details can only be extracted using the metadata API, which is tedious. The JVMTI also provides additional features compared to the CLR, including monitor wait and monitor waited, which provide information related to thread blocking on critical sections of code. Research [57, 58] has shown that the JVMTI interface incurs significant runtime overhead This overhead stems from the fact that profiling agent is written in a native language, so whenever there is a call to this agent there is the need to make JNI calls. JNI calls can incur significant overhead because they perform actions like saving registers, marshaling arguments, and wrapping objects in JNI handles [19]. This overhead may not be acceptable for some applications, so explicit bytecode instrumentation may be a solution that has less overhead because it does not require the use of JNI.

III.4.3 Bytecode Instrumentation

Although sampling and hook-based instrumentation can be performed with relatively little overhead, the extent of the information collected is limited and often insufficient to build application-level detail. Bytecode instrumentation inserts bytecode that performs application tracing within compiled code. In this approach, profiler developers redefine classes they need to profile by replacing the original bytecode with instrumented bytecode that contains logging actions at the occurrence of specified events. This approach enables the capture of application-specific events, such as transaction completion or data regarding critical sections of the application that may not be possible using only the standard events

provided by the profiler interface discussed in Section 5.3. Bytecode instrumentation therefore has less overhead and greater flexibility the profiler interface, though it can also be more complex. There are several types of bytecode instrumentation, including: " Static instrumentation, which involves changing the compiled code offline before execution i.e., creating a copy of the instrumented intermediate code. Many commercial profilers, such as OptimizeIt (Borland 2006), work this way. Static instrumentation has also been implemented by [57] and later extended in [58]. " Load-time instrumentation, which calls the agent before loading each class, and passes it the bytecode for the class that can be changed by the agent and returned. The JVMTI/CLR profiler interfaces are examples of load-time instrumentation. " Dynamic instrumentation, which works when the application is already running and also uses a profiler interface (Dmitriev, 2002). The agent makes a call to the VM passing it the new definitions of the classes that are installed by the VM at runtime. As discussed in Section 3, dynamic instrumentation supports "fix and continue" debugging instead of exiting, recompiling, and restarting. It also helps to reduce application overhead by enabling developers to (1) pinpoint specific regions of code that are experiencing performance problems at runtime and (2) instrument the classes' involved, rather than instrumenting the entire application. Instrumented classes can be replaced with the original ones after sufficient data is collected. The gathered data can be analyzed offline, the problem fixed, and the classes can be replaced at runtime. Dynamic instrumentation of bytecode is more straightforward than dynamic instrumentation of low-level machine instructions (as described in Section 3). It can also be more portable across operating systems because it uses bytecode. There is a method call provided by the JVMTI known as RedefineClasses() that is called by a profiler agent to insert the "new" bytecode of the class. When this method is called, the JVM performs all the steps needed to load a class, parse the class code, create objects of the class, and initializes them. After these steps are complete, the JVM performs hot-swapping by suspending all threads and replacing the class, while ensuring that all pointers are updated to point to the new object [20]. These dynamic

instrumentation activities can incur significant overhead in production environments and thus must be accounted for during dynamic instrumentation. Current research is addressing this problem by optimizing the swapping method so that bytecode replacement can be done at the finer-grained method level, rather than at the coarser-grained class level [19]. Similar techniques are also being explored on the .NET platform by (Vaswani, 2003). A number of tools have been developed to help instrument bytecode, much like the API for Pin described in Section 3. Examples of these tools include BIT (Lee, 1997) and IBM's Jikes Bytecode Toolkit (IBM Corporation, 2000). These tools shield application developers from the complexity of bytecode by providing an API that can be used to parse the bytecode and change it. The three bytecode instrument techniques described above incur similar overhead, due to the execution of instrumented code. Although dynamic bytecode instrumentation is the most flexible approach, it also has several drawbacks, including: " It is more complex and error-prone, than static and load time instrumentation, especially because it allows bytecode modification at runtime. " Dynamic instrumentation requires creating 'new' objects of the 'new' classes corresponding to all 'old' objects in the application, initializing their state to the state of the old object, suspend the running threads, and switching all pointers to the 'old' objects to the 'new' objects. This replacement process is complicated, e.g., application state may be inconsistent after the operation, which can cause incorrect behavior. " Static and load-time instrumentation are generally easier to implement than dynamic instrumentation because they need not worry about the consistency of a running applicationĚ Dynamic instrumentation has a broader range of applicability, however, if done efficiently. Current research [19, 21] is focusing on how to make dynamic instrumentation more efficient and less complicated.

III.4.4 Aspect-Oriented Techniques used for Instrumentation

Although explicit bytecode instrumentation is more flexible and incurs less overhead than VM hooks, the implementation complexity is higher because developers must be

highly skilled in bytecode syntax to instrument it effectively without corrupting application code. Aspect Oriented Programming (AOP) helps remove this complexity and enables bytecode instrumenting at a higher level of abstraction. Developer can therefore focus on the logic of the code snippets and the appropriate insertion points, rather than wrestling with low-level implementation details [17]. Relevant AOP concepts include (1) join-points, which define placeholders for instrumentation within the application code, (2) point-cuts, which identify a selection of join-points to instrument, and (3) advice, which specifies the code to be inserted at the corresponding join-point. AspectWerkz [8] is a framework that uses AOP to support static, load-time, and dynamic (runtime) instrumentation of bytecode. The pros and cons of the various techniques are largely similar to that discussed in Section 5.4. There are also other pros and cons affecting the use of AOP, which is discussed below. The AOP paradigm makes it easier for developers to insert profiling to an existing application by defining a profiler aspect consisting of point-cuts and advice. The following excerpt illustrates the use of AspectWerkz to define join-points before, after, and around the execution of the method HelloWorld.greet(). The annotations in the comments section of the Aspect class express the semantics e.g.,

"@Before execution (* <package_name>.<class_name>.<method_name>)"

means the method will be called before the execution of the <method_name> mentioned.

```
//////////////////////////////////////////////////////////
//
package testAOP;

import org.codehaus.aspectwerkz.joinpoint.JoinPoint;

public class HelloWorldAspect {

    /**
```

```
     * @Before execution(* testAOP.HelloWorld.greet(..))
     */
     public void beforeGreeting(JoinPoint joinPoint) {
         System.out.println("before greeting...");
     }

    /**
     * @After execution(* testAOP.HelloWorld.greet(..))
     */
     public void afterGreeting(JoinPoint joinPoint) {
         System.out.println("after greeting...");
     }

   /**
     * @Around execution(* testAOP.HelloWorld2.greet(..))
     */
     public Object around_greet (JoinPoint joinPoint) {
         Object greeting = joinPoint.proceed();
         return "<yell>" + greeting + "</yell>";
     }
}
```

Advice code can be written in the managed language, so there is no need to learn the low-level syntax of bytecode because the AOP framework can handle these details. The bulk of the effort therefore shifts to learning the framework rather than bytecode/IL syntax, which is advantageous because these frameworks are similar even if the target application language changes, e.g., from Java to C#. Another advantage is the increased reliability and stability provided by a proven framework with dedicated support, e.g., developers need

not worry about problems arising with hot-swap or multiple threads being profiled because these are handled by the framework. Some problems encountered by AOP approaches are the design and deployment overhead of using the framework. AOP frameworks are generally extensive and contain a gamut of configuration and deployment options, which may take time to master. Moreover, developers must also master another framework on top of the actual application, which may make it hard to use profiling extensively. Another potential drawback is that profiling can only occur at the join-points provided by the framework, which is often restricted to the methods of each class, i.e., before a method is called or after a method returns. Application-specific events occurring within a method call therefore cannot be profiled, which means that non-deterministic events cannot be captured by AOP profilers. For a specific case, therefore, the decision to choose a particular profiling technique depends upon application requirements. The following criteria are useful to decide which approach is appropriate for a given application: " Sampling is most effective when there is a need to minimize runtime overhead and use profiling in production deployments, though application-specific logical events may not be tracked properly. " The simplest way to implement profiling is by using the JVMTI/CLR profiling interface, which has the shortest development time and is easy to master. Detailed logical events may not be captured, however, and the overhead incurred may be heavier than bytecode/IL instrumentation. " Bytecode/IL instrumentation is harder to implement, but gives unlimited freedom to the profiler to record any event in the application. Implementing a profiler is harder than using the JVMTI/CLR profiling interface, however, and a detailed knowledge of bytecode/IL is required. Among the different bytecode/IL instrumentation ways, complexity of implementation increases from static-time instrumentation to load-time to dynamic instrumentation. Dynamic instrumentation provides powerful features, such as "fix and continue" and runtime problem tracking. " The use of an AOP framework can reduce the development complexity and increase reliability because bytecode/IL need not be manipulated directly.

Conversely, AOP can increase design and deployment overhead, which may make it unsuitable for profiling. Moreover, application-level events may be hard to capture using AOP if the join-points locations are limited.

III.5 Conclusion

This chapter reviewed the approaches to profiling distributed component based systems. The advantages and disadvantages of each approach with respect to measuring the performance of systems was highlighted. It was also demonstrated how these approaches can be applied in practice. The main contention is between two conflicting factors, the richness of data collected to the level of intrusion allowed. A nice proper balance between the two needs to be sought and that also depends on a particular application and its requirement. The material provided in the above chapter will help anyone wanting to develop such a tool for system management.

CHAPTER IV

FRAMEWORK FOR MONITORING & PROFILING DISTRIBUTED COMPONENT BASED APPLICATIONS

This chapter describes a resource monitoring framework based upon the Object Management Group (OMG), specification for Deployment and Configurations of Component based applications. This framework is known as Bulls-Eye Target Manager or Bulls-Eye in short.

IV.1 The Design of the Bulls-Eye Target Manager

Bulls-Eye is a resource provisioning service designed to enable software developers and applications in enterprise systems to (1) retrieve a list of the initial available resources in a target domain, thereby enabling the preparation of a deployment plan that meets the allocation and connection requirements of each component, (2) allocate resources for a particular deployment plan and release resources when the components or the entire deployment is removed, (3) obtain runtime resource available in the system, and (4) dynamically update the resource consumption data. This section describes the structure and functionality of Bulls-Eye.

IV.1.1 Structure of Bulls-Eye

Figure 10 shows the architecture of Bulls-Eye, which implements the CORBA interface in the Target Manager specification. Bulls-Eye's architecture has two parts: (1) a logically centralized service, known as the Target Manager Core (TM-Core), which is used by applications and system services to allocate/release resources and (2) multiple monitors (TM-Monitors) distributed in the domain to perform resource moni-toring and update the TM-Core's model of the amount of resources available.

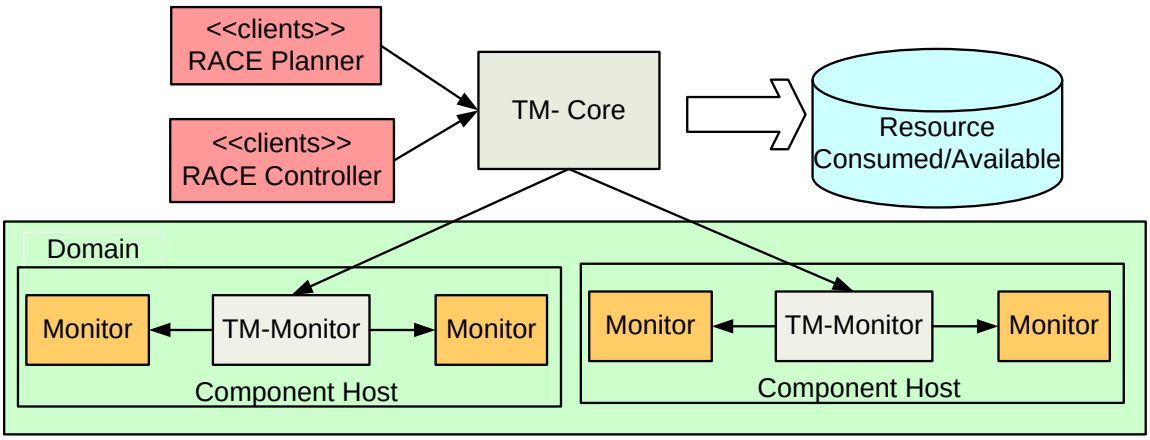


Figure 10: The Bulls-Eye Target Manager Architecture

The Domain contains all the elements of a target environment, including the nodes, interconnects between nodes, bridges connecting interconnects, and the set of resources belonging to these elements. A Domain is a logical concept, i.e., a single resource or node element can be part of more than one target domain. Domains can therefore be structured hierarchically, which a top-level domain containing other Domains. Each Domain has a TM-Core that accumulates the resource information for elements in the target domain. The TM-Core provides a standard set of operations that applications and system services can use to provision available resources statically (i.e., prior to system launch) as well as dynamically (i.e., during system runtime) in the form of a generic structure known as the DomainStruct [51]. This structure describes the contents of the entire target environment by composing data related to available nodes in the network, the connections between nodes, connection between networks, the shared resources among them, and the resources for each element. A TM-Monitor is placed on each node in the target domain to monitor the resource usage in that node. The TM-Monitor sends updates periodically to the TM-Core, with the current resource utilization/availability on that node. Upon receiving the updates, the TM-Core aggregates the data received with previous data and updates its content. Bulls-Eye uses standard CORBA request/response calls for this communication. Bulls-Eye maintains a top-level Domain element that contains all the elements of a target

domain and is identified by a universally unique identifier (UUID). This Domain element is designed so that all possible elements in the target domain can be incorporated, thereby alleviating the need to create separate structures for different types of resources, such as processors, memory, storage, and/or network bandwidth. This design also makes client code flexible by alleviating the need for any specific type of resource in the target domain since it can handle all the varieties of resource elements present. TM-Monitors collect data pertaining to their sub-domain and update the TM-Cores with fresh data. Clients are interested in data across sub-domains, so the data from different TM-Monitors are aggregated and presented uniformly. To avoid latency issues, the distributed monitors exchange only the data that changed from the previous update. This data is aggregated with the remaining target domain data that are already present.

IV.1.2 Functionality of Bulls-Eye

Bulls-Eye provides the following standard Target Manager operations that can be invoked by clients to provision system resources:

- Querying static resources. Developers or planner applications can use getAllResources() to obtain the initial static resource availability in the target domain. This operation returns the Domain structure that describes the entire target domain resources hierarchically.

- Querying dynamic resources. Dynamic resource availability can be returned by getAvailableResource(). This operation returns the same Domain structure as above, except that the resources reflect their remaining capacity.

- Committing resources. A planning application can call createResourceCommitment() to commit (i.e., allocate) resources for a deployment plan. This operation creates a ResourceCommitmentManager that can commit and release resources for a specific

plan. A pool of resources can be specified when a call to createResourceCommitment() is made or can be allocated after it is created. An exception is raised if a requested resource cannot be committed.

- Releasing resources. When an application or component is deleted all its resources must be released so they can be reallocated to subsequent applications. Applications can release resources by calling releaseResources() on the associated ResourceCommitmentManager. When a ResourceCommitmentManager is itself deleted via destroyResourceCommitment(), all remaining committed resources it holds are released automatically.

- Updating dynamic resource data. The target domain data in the TM-Core can be updated via updateDomain(). The updated information is passed using the Domain structure, which is a subset of the higher level domain structure. An enumeration called DomainUpdateKind can be used to tell Bulls-Eye whether the subset should be added, deleted, or updated.

The Bulls-Eye Target Manager functionality plays a key role in the deployment and configuration of enterprise DRE systems. On startup, it reads a standard XML configuration script that describes the resources present in the target domain. The script is prepared by a human or automated domain administrator who understands the initial target domain contents, such as nodes, the interconnects that link them, and the resources contained in them (such as processor capacity, memory capacity, and disk capacity) that are available for application usage. This script is structured according to a standard DomainStruct described in Section 3.1. The TM-Monitor used to monitor component resources on a node is collocated and started together with its associated Node Manager, which is an entity defined by the OMG D&C specification and implemented by CIAO as a daemon process running on each node. The TM-Core finds the object reference addresses of the various Node Managers from a Naming Service and establishes connection with them. At startup,

the TM-Core is passed the subset of the entire Domain, which it then uses to instructs the TM-Monitors on each node which resources to monitor. Each TM-Monitor then checks the Domain information and reports any discrepancies (such as the hard disk capacity being smaller than the initial domain description or the node is single processor instead of a multiprocessor) to the TM-Core. After Bulls-Eye starts running, clients can use it to query information about the domain, e.g., RACE components can extract domain related information for preparing a deployment plan. Any entity, such as an Execution Manager, that deploys plans in the target domain needs to provision resources via Bulls-Eye to run applications successfully.

IV.2 Resolving Bulls-Eye Design Challenges

Although the CCM specification defines the interface and the functionality of the Target Manager service it does not prescribe any design details. Thus there are a number of unresolved design challenges in implementing Bulls-Eye. This section describes key challenges encountered, presents the solutions, and outlines how these are applied to the shipboard computing applications supported by the MLRM subsystem described in Section 2.

IV.2.1 Challenge 1: Integrating Heterogeneous APIs of Multiple Platforms

Context. The resource utilization information provided to clients of Bulls-Eye should be consistent, i.e., use similar units/structures. Otherwise, the users of the data will need to convert them manually, which is tedious, error-prone, and can yield redundancies and/or inconsistencies in conversion logic.

Problem Encapsulating diverse resource utilization APIs on different OS platforms. The target domain of DRE systems typically consists of multiple operating systems, each with its own platform-specific APIs that provide information on resource usage. For example, the Unix/Linux /proc file system tracks process resources usage, such as processor,

memory, and byes sent/received from network devices. Windows, in contrast, has multiple ways of procuring this information, including (1) using a DLL (PDH.dll) that provides an API for querying resource consumption data or (2) extracting the raw data from the system registry using the HKEY_LOCAL_MACHINE key. The data structures and units returned by platform-specific OS APIs are sufficiently diverse that it is hard to write portable resource management algorithms. Diversity even extends to different versions of the same OS, e.g., performance data in Windows NT 4.0 is contained in a counter named '% Total processor time', which is named '% processor time' in Windows XP. Ideally, the middleware should convert the diverse OS-specific APIs and data into a uniform and consistent format that can be used portably by clients.

Solution. Use the Adapter pattern to adapt diverse API. To mitigate the problem of diverse resource usage APIs and data in Bulls-Eye, the Adapter pattern was used [24]. This pattern converts non-standard APIs that extract resource data into the standard interface defined by the Target Manager specification, as shown in Figure 11. The implementation of this interface converts the platform-specific data into a uniform type for storing and disseminating resource usage information to Bulls-Eye clients.

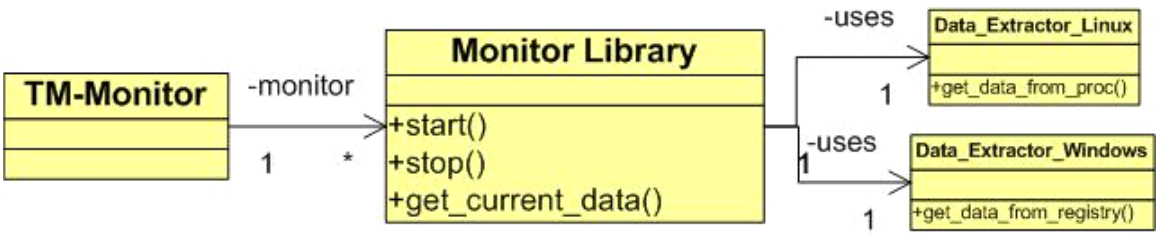


Figure 11: Using the Adapter Pattern in Bulls-Eye

The extraction of resource consumption data is tricky and obtaining accurate values depends on certain optimizations [67], such as keeping /proc open between reads and reading data in a block rather than individual characters. Likewise, extracting raw value from the registry is faster on Windows, though it is more complicated to program since it involves

parsing a text stream and extracting different structures containing performance counters. Each structure is variable length and contains headers that must be parsed to get details about the data. Although the PDH.dll mentioned above does this parsing automatically it is somewhat slower since it incurs more overhead.

IV.2.2 Challenge 2: Providing a Common Access Point to Provision Domain Resources

Context. Enterprise DRE systems are often distributed across many entities. The entire application environment is arranged hierarchically, with a top-level domain containing sub-domains, which in turn contain computing nodes connected via bridges and interconnects. Any planner for a target domain will require information of resources contained in the entire domain.

Problem. Monitoring resource utilization of multiple physically distributed resources. Information regarding resource utilization and/or availability of system resources is tracked by multiple monitors that are physically distributed across the domain. If clients were responsible for determining the resource utilization and/or availability of system resources they would have to (1) obtain the location of the monitors, (2) contact the monitors to obtain the resource utilization data, and (3) process the data acquired from multiple monitors to obtain a global view of utilization of system resource, which is tedious, error-prone, and non-scalable for each client to perform individually.

Solution. Use distributed monitors to collect data across the domain and consolidate the information via a common access point. To provide a common access point for data in the target domain, Bulls-Eye uses the TM-Monitors to send data updates at a periodic interval (which can be configured) to a centralized location, the TM-Core. TM-Monitors are located on each node in the target domain infrastructure shown in Figure 12 and use the platform-independent adapters described in Challenge 1 to extract the resource information. The

TM-Core serves as a common access point where the clients can obtain the information regarding global resource availability/utilization in a target domain.

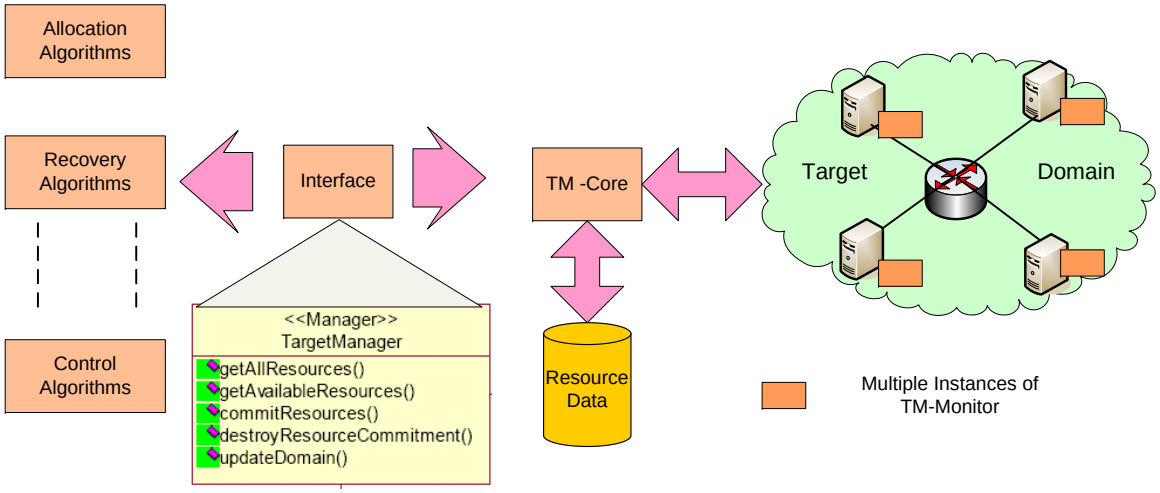


Figure 12: Providing a Common Access Point to Domain Resource Data

IV.2.3 Challenge 3: Presenting data to clients with bounded response time in uniform structure

Context. An enterprise DRE system typically have many resources present in various forms of composition. For example, a target domain may contain multiple nodes, and each node may in-turn contain multiple resources, e.g., a node may contain multiple network cards and thus be connected to many other nodes in the domain. The data from different parts of a domain should be presented in a uniform and aggregated form for the clients to manage resources effectively.

Problem Providing aggregated data of entire domain within bounded delay quickly. In a typical enterprise DRE system scenario there can be many domain elements, the data exchanged by these elements may be large, and there may be significant latency transferring such information. Data updates from distributed monitors will reach the TM-Core separately and will only pertain to a subset of the entire domain. When these updates are

merged along with many other updates to the main data store, parsing and locating the data store for a particular resource can considerably slow down response time. As a result, there is a need for an efficient and scalable algorithm to merge the data. Moreover, changes in target domain resource usage should be disseminated to clients within bounded delay so the clients can utilize the data for time-sensitive planning and resource management.

Solution. Combination of heap-sort and timer based aggregation algorithm. To solve the problem of aggregating domain data quickly, Bulls-Eye uses a combination of the following approaches:

- It optimizes communication to minimize unnecessary CPU and network processing by maintaining a cache of the last update sent to the TM-Core. When Bulls-Eye gets fresh data from the underlying component it compares the data received with the cached data and only sends an update if it detects differences. For example, no update is sent to the TM-Core if a particular reading informs the TM-Monitor that memory usage has not changed from the last update.

- Bulls-Eye also uses a combination of heap-sort and a timer-based aggregation mechanism [4] that provides O (log n) time complexity in the worst case. Each resource entity is labeled with a unique identity and is stored in a heap together with the label a pointer to the actual data so that getting the actual data from the resource label can be done in constant time. Each resource entity is labeled with a unique identity that is placed along with pointers to the actual data structure in a heap. Once updated data is received from the monitors, it is stored in a cache. An externally configured timer then fires at regular intervals and the cache is examined for any outstanding data update. The resource entity id of the update is located in the heap in O (log n) time and its corresponding data structure is updated in constant time.

IV.2.4 Challenge 4: Using Multiple Configurable Monitor Components to Extract Variety of Data

Context. There are many types of elements in a target domain for an enterprise DRE system. Each element can have its own TM-Monitor tracking its resource usage. There can also be multiple monitors for the same resource to record data from various perspectives. For example, system utilization can be monitored via the load average value on a CPU, the percentage of CPU usage, or the average amount of slack time in each cycle.

Problem. Configuring and using different resource data extraction mechanisms. There may be a variety of monitor elements that record resource data using different mechanisms, e.g., in some platforms (e.g., Windows) vendor-supplied software may track processor consumption, whereas in other platforms (e.g., Linux) developers may need to write code to access the data. In yet other platforms (e.g., VxWorks), developers may want to use specialized third-party hardware monitoring utilities. Since each mechanism may use different APIs Bulls-Eye should be well-equipped to integrate and configure different types of monitors with minimal developer effort.

Solution. Monitor algorithm wrapped within a shared library and configured using initial domain data. Bulls-Eye uses the Strategy pattern [24] to encapsulate each monitor algorithm in a DLL whose common interface defines lifecycle activities and data extraction methods. When Bulls-Eye starts running the TM-Core sends each TM-Monitor the initial Domain data containing the DLL name along with the specific configuration details for each resource element it monitors. The TM-Monitor then uses the Component Configurator pattern [35] to link the DLL and update a map that associates each resource element to its library name. The TM-Monitor calls each library to start/stop monitoring and to periodically get their current data. It also combines the data from each library into one Domain structure before uploading it to the TM-Core, thereby simplifying extensions to Bull-Eye's monitoring capabilities.

IV.3 Workload Modeling

This section describes the various factors affecting the workload of a web application portal which is used as a representative example of a component-based distributed application. The workload modeling will include demand for various services, sequence of service invocations, and roles played by customers. It also describes the methods and strategies that have been proposed in recent research to characterize those factors and produce workload models. These workload models can then be used to evaluate web application portal performance. The workload modeling process starts from live traces of the system that contains logs of incoming user requests to the system. The traces represent actual workload and may potentially contain a substantial amount of data. Since processing such a large amount of data for performance evaluation is often unrealistic it may be ne-cessary to find some inherent patterns in the data. This pattern can then be represented through the use of probabilistic models and statistical distributions. The models should be generic enough so they can be used for a wide set of performance evaluations. For example a web application portal, such as an auction site like ebay, can have a number of different types of users, such as casual browsers, sellers, buyers, bidders, and reviewers. Users will likely invoke different services on the portal in different sequences, depending upon their objectives. By studying the observed log of user behaviors, it is possible to characterize the sequence of activities of a particular type of user.

Table 20 shows a possible set of transitions of a user doing simple browsing. It also contains the probability of a browsing user invoking a particular service after another. The row and the column headings consist of the various available services. Element x_i, j is the entry in the i^{th} row and the j^{th} column and represents the probability of invoking the ith row service after invoking the jth column service. For example, consider the entry $X\_5,7$ which is equal to 0.99, which conveys that a typical user invokes "Search_it_reg", 99% of the times after invoking ""Browse_Cat_Reg". Such a set of transition can be estimated from the observed logs. In this manner, the behavioral patterns of different categories of

	Home	Browse	Browse Cat	Browse Region	Browse Cat Reg	Srch Items Cat	Srch Items Reg	View Items	View User Info	View Bid Hst	View Items Reg	View User Info Reg	View Bid Hst Reg	Probabilities
Home	0	0.01	0.0025	0.0025	0.0025	0.0025	0.0025	0.0025	0.0025	0.0025	0.0025	0.0025	0.0025	0.0026
Browse	1	0	0.0075	0.0075	0.0075	0.0075	0.0075	0.0075	0.0075	0.0075	0.0075	0.0075	0.0075	0.0100
Browse Cat	0	0.7	0	0	0	0	0	0	0	0	0	0	0	0.0070
Browse Reg	0	0.29	0	0	0	0	0	0	0	0	0	0	0	0.0029
Browse Cat Reg	0	0	0	0.99	0	0	0	0	0	0	0	0	0	0.0029
Srch Items Cat	0	0	0.99	0	0	0.44	0	0.74	0	0	0	0	0	0.3343
Srch Items Reg	0	0	0	0	0.99	0	0.44	0	0	0	0.74	0	0	0.1371
View Items	0	0	0	0	0	0.55	0	0	0.8	0	0	0	0	0.2436
View User Info	0	0	0	0	0	0	0	0.15	0	0.99	0	0	0	0.0747
View Bid Hst	0	0	0	0	0	0	0	0.1	0.19	0	0	0	0	0.0386
View Items Reg	0	0	0	0	0	0	0.55	0	0	0	0	0.8	0	0.0999
View User Info Reg	0	0	0	0	0	0	0	0	0	0	0.15	0	0.99	0.0306
View Bid Hst Reg	0	0	0	0	0	0	0	0	0	0	0.1	0.19	0	0.0158

Table 3: Transition Probabilities Between Various Services

users can be understood. A technique called Customer Behavior Modeling Graph (CBMG) is presented in [65]. This technique represents user behavior patterns in the form of probabilistic models. Figure 10 shows such a diagram of a typical user moving from one web page to the other. The figure consists of a set of states and transitions. The states are connected through transitions. Each state represents a web page or service provided by the application. The transition from each state to another has a probability associated with it. As shown in Figure 10, a user viewing the "Browse" web page can navigate to the "Browse Regions" page with a probability of 0.3 and to the "Browse Category" page with a probability of 0.7. Similarly a user in "Search Items" can "View Item" with 0.55 chance or go back to the "Browse Category" page. These probabilistic models can be solved using standard techniques [84].

The steady state probability (percentage of user sessions) for each service type is denoted by the vector π. The value of π_i denotes the percentage of user requests that invoke the the i^{th} service. The vector π can be obtained by using a technique is similar to the one in [83]. Once computed, the amount of load on each service type can be calculated from

the total number of user sessions. The rightmost column in Table 20 gives the steady state probabilities of each service.

After solving the models, the percentage of user calls for a particular component can be estimated. This can then be used together with the resource requirement profiled to find the overall resource requirement of the component.

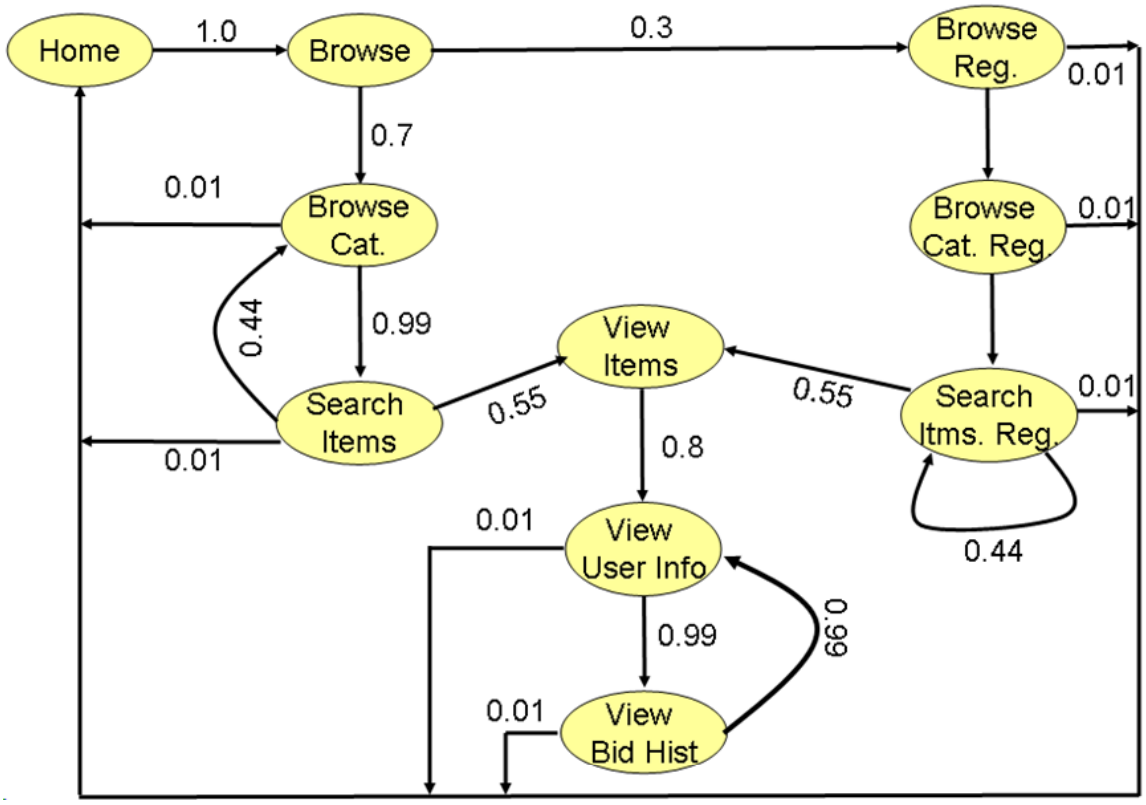


Figure 13: Customer Behavior Modeling Graph of a Typical User

IV.4 Conclusion

This chapter introduced a framework for runtime monitoring and profiling of components in a distributed component based application. The framework consists of agents that need to placed on each node and a central data collecter in one node. The resource requirement data for all the components are collected in a central respository and available upon

request. Thus such a framework can be used at runtime to collect usage data which can be used for either runtime decision making or analysis. This framework also can be used to collect profile data while the application is run with a basic dummy load. The framework collects resource usage data which are actually base level profile data.

The chapter also discusses the technique of workload modeling using the CBMG as an example. By using techniques such as CBMG, amount of hits to each component can be computed. This, along with the profile data collected using the framework will help in computing the overall resource requirement of each component for a particular workload.

Part II

Performance Estimation of Large Scale Distributed Component Based Systems

This part of the dissertation deals with performance estimation of large scale component based systems. As discussed previously, this dissertation deals with the issues of assuring QoS by using novel deployment techniques. In order to ensure QoS, there is the need to estimate the application QoS such as response time and verify it to be within some accepted bounds. One way to estimate QoS is by developing a model of the application. This part of the dissertation discusses different ways to come up with models of an application which can be used for performance estimation of various kinds of systems.

Enterprise and real time systems together comprise a large part of relevant computer systems. In enterprise systems, the average value of performance characteristics is important while in real time systems there is the need to ensure deadlines associated with the response time. Due to such different requirements, there is the need for different kind of models. This part of the dissertation discusses how various kind of models can be built for such different systems. Chapter V discusses analytical models for enterprise systems, Chapter VI presents simulation models which simulate software contention which occurs due to the conflict of different threads in a software program; Chapter VII discusses the development of analytical models for real-time systems which can be used to analyze the performance characteristics of multiple co-located tasks under a particular scheduling algorithm such as earliest-deadline first, rate monotonic etc.

CHAPTER V

ANALYTICAL MODELS FOR PERFORMANCE ESTIMATION

Chapter I has motivated the requirement of analytical models for large scale distributed component based applications. This chapter looks into how analytical modeling can be used to come up with such performance models. First a basic queuing model is prepared using the case study, RUBiS. In the next sections more specific scenarios that come up in the real world are discussed.

As discussed in Chapter I there is the need for performance estimation to place components. This estimation process involves (1)predicting the resource requirements of each component for certain application loads, (2) predicting the response time of each service for a particular placement, and (3) computing the overall resource utilization of each node. As application components move among the various nodes, the performance of each service in the application will vary. Performance also depends upon the components that are collocated.

A queuing model provides average case performance analysis of a system. It also models the interaction of collocated multiple components by modeling the queuing delay for resource contention. A deployment plan is converted into a multiple class queuing model that maps each service as a class in the model. The components of a single class that are placed in the same node are considered as a single entity. Their Service Demands are summed together. After the model outputs the results, the performance parameters of each class are mapped onto the services.

Figure 14 shows the process of creating a model of the application.

The input to such a process consists of the component placement map (mapping of application components to nodes) along with their Service Demands and the workload parameters, such as arrival rate of transactions and number of concurrent user sessions.

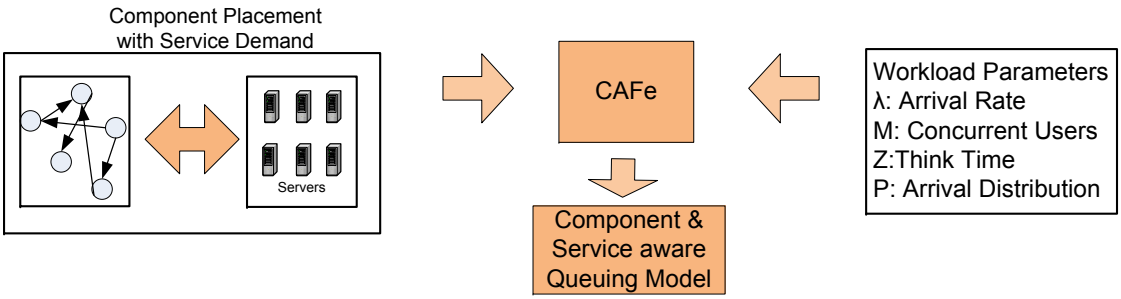


Figure 14: Create Models of Application

Depending upon the workload characteristics, a closed or an open model of the application is constructed. An open model assumes a continuous flow of incoming requests with a given average inter-arrival time between clients. A closed model assumes a fixed number of user sessions in steady state. The sessions are interactive and users make requests, then think for some time, and then make a subsequent request. If an application consists of independent requests arriving and being processed, it can be modeled as an open model. If there are inter-dependent sequences of requests coming from a single user, however, it must be modeled using a closed model.

These analytical model can be solved using standard procedures. Mean Value Analysis (MVA) algorithm [46] is used to solve closed models, while an algorithm based on the birth-death system is used to solve open models [46]. The solution to the analytical model provides the response times of the various services and also the utilization of the resources such as processor or disk usage in the various nodes.

V.0.1 Analytical Modeling of RUBiS Servlets

After the Service Demands and the steady state probability mix for each service is available, an analytical model of the application can be developed. The RUBiS benchmark assumes a client to carry out a session with multiple requests with think times in between. This type of a user behavior must be modeled with a closed model.

As soon as a client finishes, a new client takes its place. The average number of clients remains fixed. Figure 15 shows the analytical model of the RUBiS Servlets version.

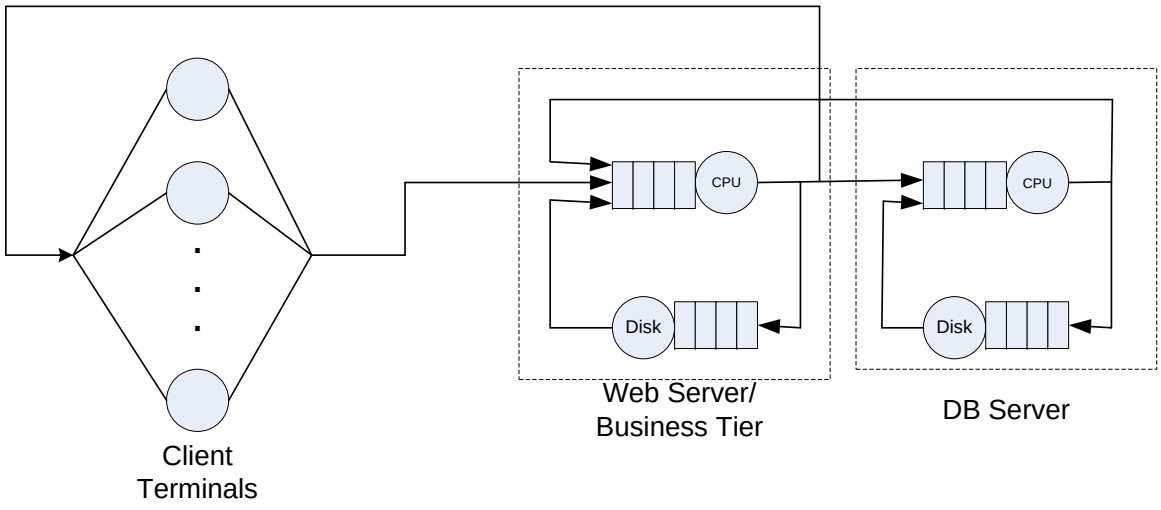


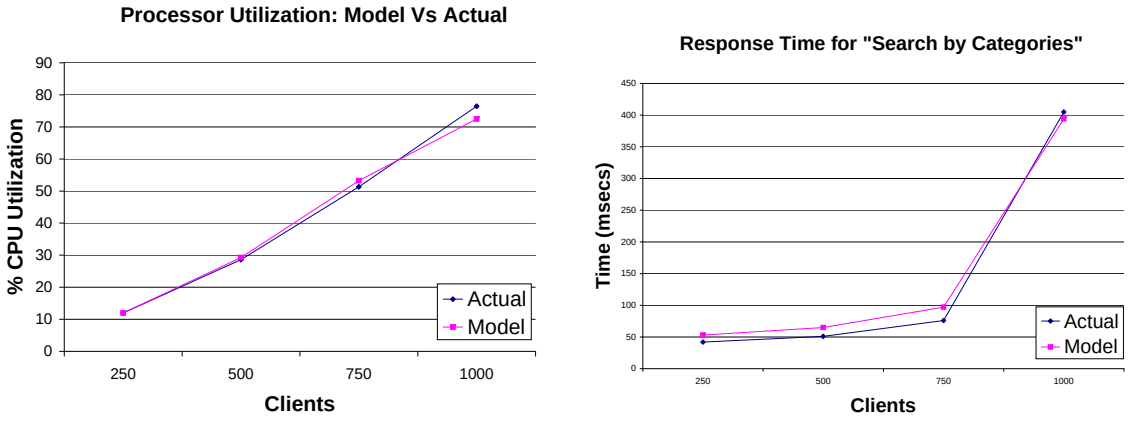
Figure 15: Closed Queuing Model for Rubis Java Servlets Version

As mentioned in Section IX.3.2, the processor is the contentious resource. Each machine is represented by two queues, one for the CPU and the other for the disk.

Figure 15 also shows two queues for each of the two node in the deployment. The first node is the Business Tier, which also serves as the web server. The second node is the Database Server. The various client terminals are represented by delay servers. A delay server is a server that does not have a queue, so clients wanting to use the server can access it directly it without waiting. This design models user think times since as soon as a response to a previous request comes back, the user starts working on the next request.

Figures 16a and 16b compare the results predicted by the analytical model to the actual results collected from running the benchmark.

The benchmark is run using progressively increasing number of clients for $250, 500, 750$ and $1,000$, respectively. The components are placed in the nodes using RUBiS's default strategy, which places all the Business Tier components in the Business Tier node and the entire database in the database server. The results in these figures show the model



(a) CPU usage

(b) Response Time

Figure 16: Validation of Analytical Model

accurately predicts the response times of the services and the processor utilizations of the nodes. This model can therefore be used by CAFe to find the placement of the components that optimizes the capacity of the deployment.

V.1 Challenges in Analytical Modeling of Multi-Tiered Applications

The previous section described how a basic component aware queuing model of an application can be developed. These models can be used for a variety of important functions, such as performance evaluation, capacity planning, configuration management, admission control, and cost analysis. One important criteria for such models is their accuracy. Accurate models aid in all of the above functions, which in turn help build better and reliable systems. Unfortunately, developing accurate system models for multi-tiered applications is hard. Consequently, often these models are approximate and may not truly represent the actual application.

Significant prior work exists that uses one or more of analytical modeling techniques for developing accurate models. For example, [70, 75, 77, 84, 85] have used analytical techniques and profiling to build models of multi-tiered web portals but have not accounted for increased system activity, such as page-faults which occur with increased load. The

emerging trend towards multiple processors/cores has also not been considered by most of these works.

An additional issue concerns databases. Most prior work fail to account for database optimizations, *e.g.*, optimizations to handle similar (*i.e.*, overlapping) queries that run concurrently. Finally, resource allocation, which is a key issue in capacity planning, has been investigated only at the granularity of an entire tier-level, however, this coarse level of granularity is insufficient in minimizing the number of and efficiently using resources in the context of modern multi-tiered systems that are made up of finer-grained components.

This section identifies three some common scenarios that occur in realistic production environments and highlights the limitation in modeling these scenarios using modeling techniques developed in recent work on multi-tiered web applications. The cases illustrated clearly suggest the need for application-specific and/or domain-specific models, which are modified versions of traditional models catered to a specific scenario. The chapter subsequently sketches preliminary ideas on a process that combines profile data and analytical modeling to develop more accurate models.

The scenarios are shown in the context of the Rice University Bidding System (RUBiS) [3] which is a prototype application of an ebay like auction site. The experiments shows how modeling techniques used in recent research for multi-tiered web applications fall short in coming up with good estimates of system parameters that in turn limit the accuracy of the developed models.

V.1.1 System Activity at Heavy Load

Recent work [75, 77, 84, 85] use queuing models to estimate performance of multi-tiered applications. But such models can introduce errors in estimating performance accurately when system activity, such as context switching and page faults, increases. To highlight this limitation and understand why these models fall short, we developed a similar queuing model and used mean value analysis algorithm to estimate response times observed

by clients of RUBiS as system activity keeps increasing. We then compared these estimates against experimentally-measured response times for the `SearchByRegion` service offered by RUBiS as shown in Figure 22a.[1]

Figure 22a shows that the two curves remain close to each other at low utilization for a small number of clients. However, as the number of clients increases (which in turn increases system activity), the disparity between the experimental and analytical results becomes significant.

This disparity is attributed primarily to the service demand used by the model, which is measured by executing only a single job [46] when there is minimal load. The service demand in the model does not account for excess system activity that takes place with a large load. This excess system activity needs to be taken into account in the model which potentially could be done by inflating the service demand as load increases. Thus our approach is to model the service demand as a function of the load and use it in the MVA calculation. This will consider the increased system activity and will improve the model estimation.

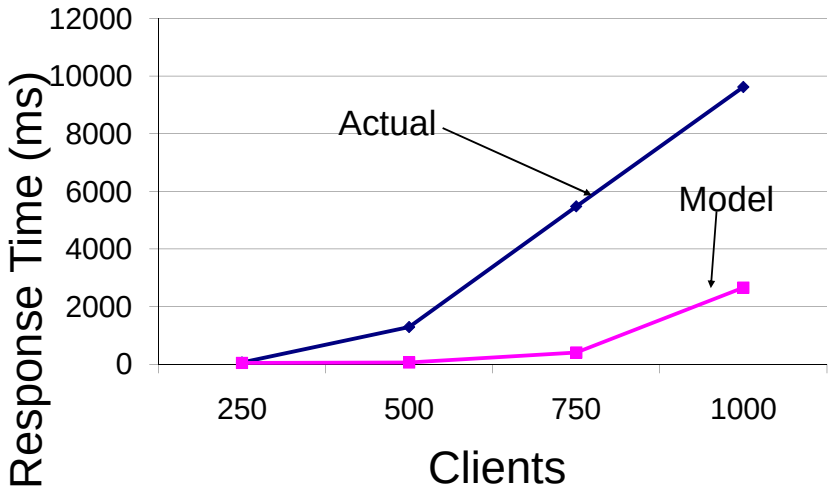


Figure 17: Comparison of Analytical vs Empirical Data

[1] Similar comparisons were made for services like `SearchByCategory` but are not shown due to lack of space.

V.1.2 Multiprocessor effects

With increasing availability and use of multi-processors and multi-cores for multi-tiered applications such as web portals, existing closed queuing network models [61, 77, 85] must now incorporate support for multiple servers. Although existing closed queuing networks can be solved efficiently using the mean value analysis (MVA) algorithm, accounting for multiple-server models requires computing the probability mass function of the queue sizes for each server. The mass function is used within MVA to calculate the total expected waiting time that a customer experiences on a server. This approach, however, significantly increases the complexity of the MVA solution. A potential solution has been suggested in [72] where a correction factor is used that estimates the server waiting time for multi-processors. But it is also suggested that this correction factor will be application and hardware specific. Thus there is the need to somehow measure such a correction factor in order to use it.

V.1.3 Dependent Transactions

Multi-tiered applications often comprise databases, which means that user invocations will result in database transactions that operate on related data. Modern day databases use different kinds of optimizations for multiple overlapping queries. These optimizations include caching of intermediate data [63] or using heuristics [62]. Such optimizations could result in unpredictable performance for concurrently executing and overlapping queries, which makes it hard to estimate the performance of multi-tiered applications at design-time. Nonetheless, it is important to model these effects since they can play a dominant role in the overall performance of the multi-tiered application. Note that such effects will be primarily application- and query-specific and can be modeled with the help of extensive profiling.

We highlight this challenge using a specific example in RUBiS and then describe how to model such behavior based on profiled data. We focus on two types of browsing queries

in RUBiS: `SearchByRegion` and `SearchByCategory`. Both these queries work on the same table named *items*. However, `SearchByRegion` also uses an additional *users* table. We observed that when the two different queries execute all by themselves (*i.e.*, no concurrency), the `SearchByCategory` is much faster than the `SearchByRegion` as shown in Figure 18 (Lines `SearchByCategory` and `SearchByRegion`).

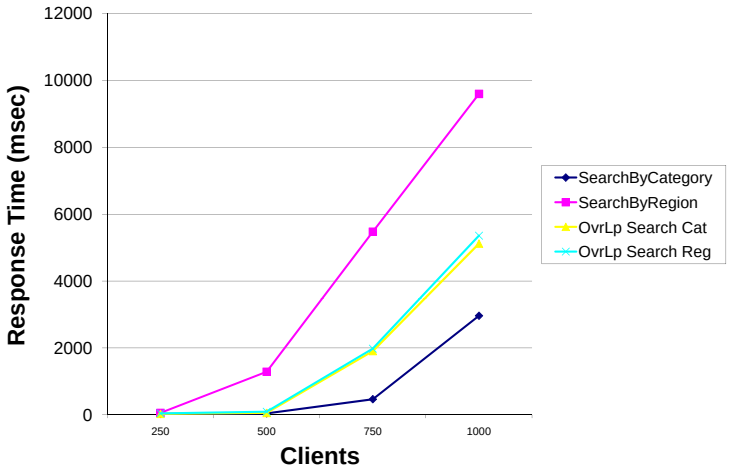


Figure 18: Concurrent Overlapping Queries Have Similar Response Times

However, when the two queries execute concurrently on the same database, their individual response times become almost the same (Lines `OvrLp Search Cat` and `OvrLp Search Reg`). With a low number of clients the response times are somewhat different but as the number of clients increase (and hence the overlapping queries increase), the response times experienced by all the clients are almost the same. We surmise that this behavior is attributed to some optimization in the databases when multiple overlapping queries run concurrently. Intuitively it seems that since both queries are using the same table, there is some locking due to a software lock such as a semaphore or a mutex.

Although we can only surmise the exact cause of the behavior shown in Figure 18, we were able to reproduce this behavior across multiple experiments. Figure 19 shows a closed queuing model where the processor on which the query executes is modeled as a queue.

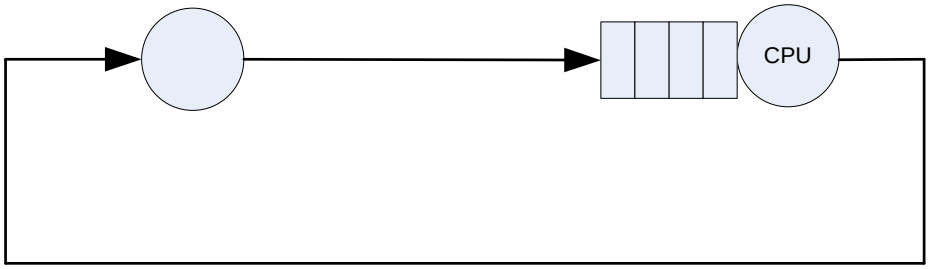


Figure 19: Traditional Closed Queuing Model

An additional delay server models the client think time. This is the normal way in which such database queries can be modeled using queuing networks. Here we assume that only one resource is being used, which is the CPU. This model also assumes there is no internal blocking and thus works properly as long as there are no optimizations.

To model the blocking effect stemming from the use of software locking for overlapped query optimizations, we introduced another resource on which each query waits for some time. This resource could be thought of as a software lock where each query waits to grab the lock before accessing the table. Thus, the newly introduced resource simulates the blocking time spent by the queries when executed concurrently. Figure 20 shows the enhanced queuing model which introduces an extra queue compared to Figure 19.

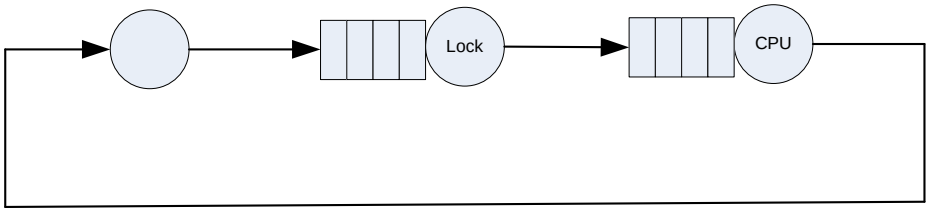


Figure 20: Additional Queue to Model Software Blocking

Configuration	Configuration Details
Configuration 1	Four 2.8 GHz Xeon CPUs, 1GB ram, 40GB HDD
Configuration 2	Quad Core 2.4 GHz Xeon CPUs, 4GB ram

Table 4: Machine Configurations Used

Having developed the model, the next step is to estimate how long a query waits on the lock. Intuitively it appears that a query of one type will wait for the query of the other type to finish. Thus, it will wait for the length of the time equal to what the other query takes to finish its job. In other words, the waiting time for the query under consideration is equal to the service demand of the other concurrently running query on that processor. This might not be the perfect way to estimate the time spent on the lock and the actual time depends upon the optimizations in the database. But our objective is to build an approximate model. Thus, we assign the service demand of the query on the lock to be equal to the service demand of the other query on the processor.

Capacity planners must identify sources of such database optimizations in their applications and use the suggested technique of modeling the extra locking queue to obtain accurate performance estimates. An alternative way of modeling this scenario would be to consider the two job classes as a single one. But then the parameters of each separate job class will not be available and cannot be used if they are required.

The model is validated and Figure 21 shows the performance estimation of the inter-dependent queries. It can be seen from the figure that our model accurately estimates the effect of the inter-dependent queries. The above queuing model is used to predict the response times of various services of RUBiS under two different hardware configurations. The hardware configurations are given in the Table 4:

Two sets of experiments are run in each machine configuration, one when a single service is run another in which multiple services(class) are run. The former exhibits the performance of each component running in isolation while the latter gives an idea of the

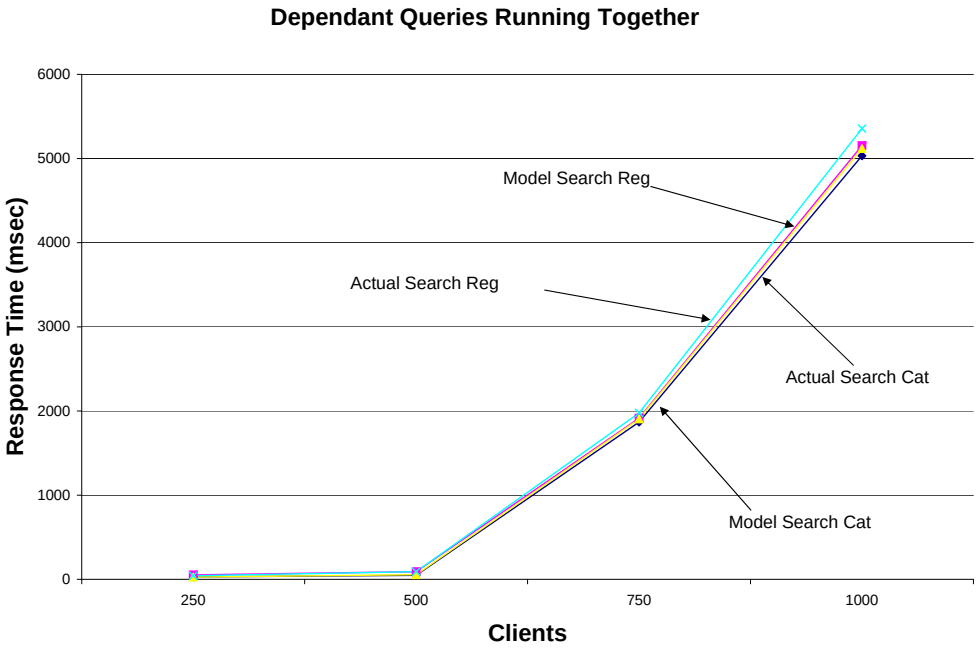
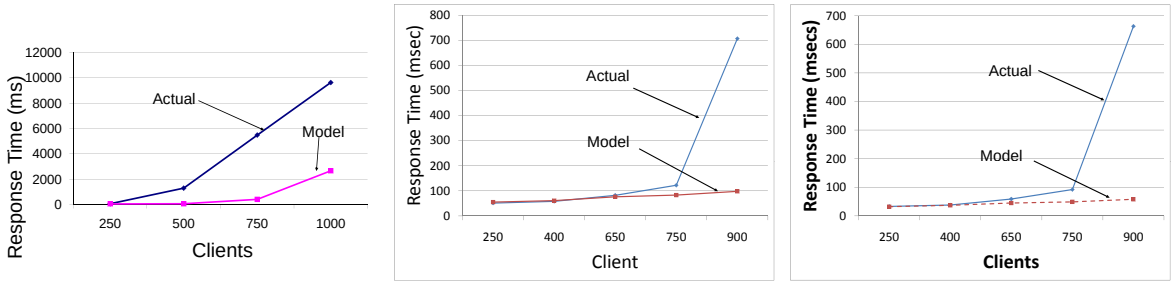


Figure 21: Model of Inter-Dependent Queries

effect of collocation of components. In both cases, we use the above queuing model to predict the response time. The parameters (service demand) of the queuing model depend upon the machines on which the application runs. The service demands are thus computed by profiling each single component on the different machines by running with a single client in the system. In previous work[14], it was seen that CPU activity of RUBiS increases with load compared to memory and bandwidth which remain fairly constant. Thus CPU is mainly considered here.

Figure 22a shows the response time when a single service of RUBiS is run with Configuration 1. Similar behavior is also seen when multiple services are run, as shown in Figure 22. Here we reproduce two of the services "SearchByCategory" and "SearchByRegion" which have higher response times. In this experiment around 12 services are running each service having 3 components. The other services also incur similar estimation errors.

Figures 23a, 23b, 23c show the response time of the same above services when multiple services are running together in Configuration 2. The model prediction is also shown.



(a) Single Service (b) Multi Service: SearchByRegion (c) Multi Service: SearchByCategory

Figure 22: Comparison of Analytical vs Empirical Data

(a) Multi Service: SearchByRegion (b) Multi Service: SearchByCategory (c) CPU Utilization

Figure 23: Comparison of Analytical vs Empirical Data In Configuration 2

Strange behavior is seen in this experiment, as shown in Figure 23c where the CPU is only loaded till around 30%. After this point, as load increases the response time shoots up even though the CPU is underloaded. The memory and bandwidth also remains much below its capacity(not shown here for lack of space). It is evident that there is some other bottleneck in the system which causes the response time to shoot up. It could be due to software contention. The queuing model understandably cannot predict this behavior since the "invisible bottleneck" is not modeled. Thus it estimates the CPU utilization to increase with load while in reality it saturates around 30% as shown in Figure 23c.

Finding the root cause of this bottleneck is hard since it might require investigating immense amount of code and analyzing various scenarios. In a real world scenario it could also be caused by third-party libraries, the code for which may not be available. In such

cases, finding the root cause is nearly impossible. In such a scenario, a basic queuing model becomes increasingly erroneous and cannot be relied upon for proper prediction. Such scenarios are extremely challenging and it is very difficult to come up with models which estimate performance characteristics in a proper way.

A straightforward way would be to profile the system with different workload and create statistical regression models. But such an approach will not help us in predicting the performance when components are arbitrarily placed in different combinations in the machines since this will require us to profile the application using every combination of the components which is clearly not possible. The next section details the solution approach followed in this work.

V.1.4 Solution: Profile driven Regression based Extended Closed Queuing Network

This section discusses the details of the modeling techniques developed by enriching basic closed queuing models with statistical regression models to come up with increased accuracy in estimating application performance.

In our approach we come up with the following steps to produce better models: (a) Profile individual components, (b) Create regression models, and (c) Extend queuing models with regression models

This helps us in estimating the performance of multiple components running together in the same machine using profile data of individual components. One one hand it leverages the strength of regression models where unique scenarios or environmental effects are captured through profiling and on the other hand uses the strength of queuing models which enables the performance estimation of multiple classes (or collocated components). The above approach ensures that the profiling is kept to no more than it is absolutely required and also leverage existing queuing models to estimate multi-component behavior which can be used for component placement decisions.

V.1.4.1 Modeling Increased System Activity

The Figures 22a, 22b and 22c show that there is increased error in model prediction at high load. This section discusses the probable reasons for such error and comes up with solutions to address them.

Queuing models can be efficiently solved using approximate Mean-Value Analysis (MVA) algorithm [46]. The main equation that is used to compute response time is given by

$$R = D_{ir} + n_i \times D_{ir} \tag{V.1}$$

where D_{ir} is the service demand of service r on device i and n_i is the number of waiting jobs on device i. The service demand gives the actual resource time used by a component while processing the job. At high load, additional activity in the machine due to system work including context switches, paging, swapping etc also adds to the response time. This excess system activity is not accounted by Equation V.1.

To further investigate this intuition we measure the number of context switches that occur per second as client population is increased (measured using Configuration 1 as given in Table 4). This data is plotted in Figure 24. It can be seen that the number of context switches per second steadily increases with increase in client population. Since context switch is one type of system activity, it clearly shows that the amount of system activity increases with clients. The challenge now is to capture and quantify this increased system activity in terms of resource usage so that it can be added to the MVA analysis which will then produce better estimation of performance parameters.

Next we measure the total CPU utilization per job as we increase client population. This measurement is done by capturing the total CPU utilization for the lifetime of the experiment and dividing it by the total number of jobs completed in the same interval. The observed total CPU utilization per job is shown in Figure 24 along with the context switches per sec. It is seen that the CPU utilization per job steadily increases along with the number of context switches per sec and becomes steady after some time. Initially, at very low load

(single client) there is nearly zero context switch/sec. The CPU utilization/job is also very less and matches with the service demand value. Consequently it can be deduced that as system activity increases the excess CPU utilization per job is due to additional system activity. Obviously such effect must be accounted for in a performance model. However, traditional queuing models do not account for this behavior.

To overcome this limitation we define a term "Overall Service Demand" (OSD) which is defined as the total resource time required to complete a single transaction. Thus the CPU utilization shown in Figure 24 is actually the OSD for the concerned service. As shown in Figure 24 the OSD has the potential to vary with load since it is the sum of the service demand and resource usage due to system activity.

Overall service demand (OSD) can be measured using the service demand law [46]. The service demand law is given as $D_i = U_i/X$ where D_i is the service demand on the i^{th} device, U_i is the utilization of the i^{th} device, and X is the total number of transactions/sec or throughput of the system. When the service demand law is used at high load it returns the OSD which is a sum of the service demand and the resource time spent due to system activity. The OSD can thus be obtained for different client population by measuring the device utilization and the throughput of the services while client size is varied. The measured values are then used with the above law to obtain the OSD.

We empirically profiled each service hosted by the RUBiS web portal by varying the client size from an initial small value to a large value. Here we assume that individual components (services) of a large, multi-tiered system are available for unit testing and profiling. We measured the processor usage and the number of successful calls for each client population size. The service demand law is then used to compute the overall service demand for each client size.

As seen in Figure 24, the overall service demand remains steady at low utilization (≤ 10) and then follows a near linear increase till around 80% utilization or 350 clients. The linear rise can be attributed to the increase in system activity as clients increase. Since

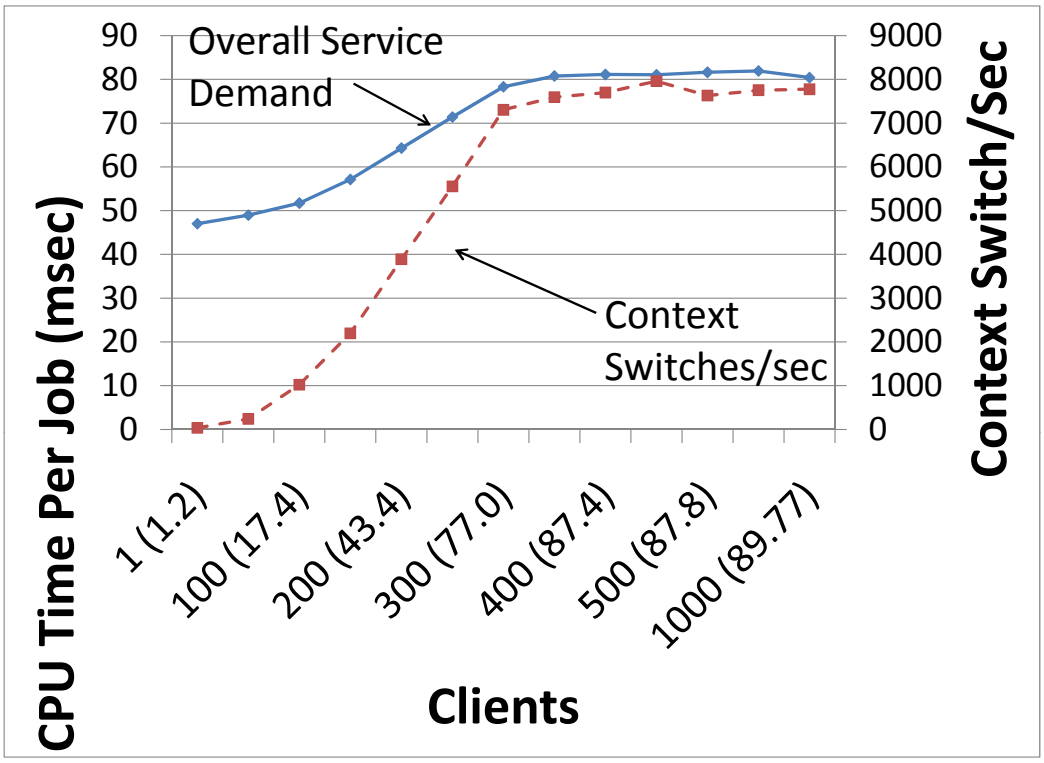


Figure 24: Overall Service Demand

each client represents a thread in RUBiS, consequently, an increase in the number of clients increases the number of threads.

This behavior is better understood from the number of context switches as utilization and clients increases. There is negligible context switching for low number of clients but increases linearly with clients until 350 clients when it becomes steady. At 350 clients, the service demand also stabilizes because the device (*e.g.*, CPU) utilizations are close to saturation (greater than 90%) and there is not much scope for any increase in system activity. We have observed similar behavior in the other services of RUBiS.

Based on these insights, the overall service demand is modeled as a load-dependent function of processor utilization which is piecewise linear. To empirically obtain accurate demand functions, the Polyfit tool provided in the Matlab Curve Fitting Toolkit is used. The

resulting function which represents the overall service demand for the `SearchByRegion` service is given by:

$$OSD_{sr}(U) = \begin{cases} 48 & \text{for } U < 8 \\ 0.4264 \times U + 45.1062 & \text{for } 8 <= U <= 85 \\ 81.62 & \text{for } U > 85 \end{cases} \quad \text{(V.2)}$$

and the function representing the service demand for the `SearchByCategory` service is given by:

$$OSD_{sc}(U) = \begin{cases} 28 & \text{for } U \leq 5 \\ 0.0457 \times U + 24.94 & \text{for } 5 <= U <= 84 \\ 52.06 & \text{for } N \geq 84 \end{cases} \quad \text{(V.3)}$$

The coefficient of determination, R^2, value for the linear fit is 0.99 for both equations indicating very good fits. Capacity planners using MAQ-PRO should adopt a similar approach to obtain accurate functions for overall service demands of individual services belonging to their applications.

The MVA algorithm used is now modified to include usage of the overall service demand instead of the original service demand which represents the actual resource time used by a transaction. Thus Equation V.1 is replaced by the following:

$$R = OSD_{ir}(U) + n_i \times OSD_{ir}(U) \quad \text{(V.4)}$$

where OSD_{ir} is the overall service demand for the r^{th} class on the i^{th} device. So the single constant value of service demand is replaced by overall service demand which takes into account the system activity in the machine.

Using the above version of MVA, we validate the response time prediction of the model

against actual measured response time under a single processor machine and is shown in Figure 25. It can be seen that for single processor machines, our extended MVA can nicely approximate the response time.

V.1.4.2 Modeling Multiprocessor Effects

Due to the increasing availability and use of multi-processors and multi-cores for large scale applications, such as web portals, existing closed queuing network models must model multi-processor effects on performance. We use the extended version of MVA explained in Section V.1.4.1 to validate the prediction under hardware Configuration 1 as given in Table 4. Figure 26 compares the model estimation with empirical measurement and shows that there is still some gap in the estimation which is investigated in this next section.

Typically multiple-server queuing models are solved by considering each multiple server as a load dependent server [46] and computing the probability mass function of the queue sizes for each server. The mass function can then be used within MVA to calculate the total expected waiting time that a customer experiences on a server. This approach, however, significantly increases the complexity of the MVA solution. There have been attempts in recent research [72] in which a simple approximate method is presented that extends MVA to analyze multiple-servers. In [72], the authors introduce the notion of a *correction factor,* which estimates the waiting time. When a transaction is executed on multi-processor machines, the waiting time for each transaction on the processor is taken to be the product of a constant factor, the service demand, and the average number of waiting clients as captured by the following formula:

$$R(N) = SD + c \times SD \times n \tag{V.5}$$

where $R(N)$ is the response time of a transaction when there are a total of N customers in the system, SD is the service demand of the job, n is the average number of customers waiting on the device, and c is the correction factor to compute the waiting time. In their

work they theoretically compute the value of the correction factor. [72] also considers a constant service demand and thus Equation V.5 need to be adjusted by using Overall Service Demand instead of service demand to incorporate increased system activity at high load. Equation V.6 shows the revised version including overall service demand (OSD).

$$R(N) = OSD(U) + c \times OSD(U) \times n \tag{V.6}$$

We surmise that such a correction factor will depend on a number of factors, such as the domain of the operation, and the service time characteristics for the underlying hardware, the cache hit ratio, memory usage levels, memory sharing etc. Therefore, the correction factor will vary with each different scenario and need to be profiled on the particular hardware. We now describe how we found the correction factor for the RUBiS example.

Capacity planners using the MAQ-PRO process should adopt a similar approach for their applications. The data needed to compute the correction factor can be extracted from the same experiments done to estimate the OSD as mentioned in Section V.1.4.1. Thus there is no need to conduct additional experiments and a single experiment will suffice for both the OSD and the correction factor. Our approach again is to profile individual components and then estimate the expected performance when any combination of the components are placed in the machines.

Referring to Equation V.6, the value of the overall service demand $OSD(U)$ can be found using the profile-based curve fitting approach explained in Section V.1.4.1. The average number of customers waiting on the CPU, n, is obtained by using standard system monitoring tools. The response time for each transaction, $R(N)$, can be obtained from the application logs or by time-stamping client calls. The only unknown in Equation V.6 is the correction factor, c, which can be obtained by solving the equation.

We ran a number of experiments for different classes of services supported by RUBiS with different client population sizes and the variable n was monitored. $R(N)$ was obtained from the RUBiS logs. The load-dependent service demands, $OSD(U)$, were obtained from

Equations V.2 and V.3. The correction factor was then computed using Equation V.6, which is presented in Table 5 for two different services in RUBiS for a 4 processor machine. Figure 27 shows the comparison of the empirically obtained correction factor to the one proposed by Suri. It clearly shows that the actual correction factor is much different and depends upon the specific scenario.

Table 5 presents the experimental values and the computation for the correction factor with different client population for the two main services in RUBiS. The inverse of the correction factor is given in the rightmost column of the table. It is termed as *CI*. It can be seen that the correction factor varies with clients or processor utilization.

Since the correction factor actually represents the multi-processor effects on performance, it should be dependant on the number of processors in the machine. To validate our hypothesis, we configured the machine to use different number of processors and repeated the experiment with 1 and 2 processors, respectively. Figure 28 shows the value of *CI* with clients for the service "SearchByCategory". Similar results were obtained for other services but are not shown due to space constraints.

The value of *CI* is interesting. It has a very high value with less load but slowly converges to a steady value at high load. The steady value appears to converge to the number of processors in the system. It can also be seen that the variation in the factor increases with increase in processors. Higher values of *CI* (*i.e.*, lower value of the correction factor) improves the response time as seen from Equation V.6. This observation indicates that the correction factor could also be indicative of the inherent optimizations such as caching that occur in the system.

This hypothesis needs further investigations and will become part of our future work. It also tells us that at high load there may not be much scope of optimization and the system behaves like a straightforward fluid flow system and can be modeled using variations of fluid flow modeling techniques as done by many recent work. [11, 38, 54]

The value of *CI* for each client population is averaged over all the services. It is then

Service	Clients Name	Service Demand (msec)	Avg Waiting	Response Time	Corr. Factor	CI CI
Search	100	51.71	2.00	54	0.022	45.16
ItemsByReg	150	57.12	2	62	0.043	23.40
	200	64.29	3	77	0.066	15.17
	250	71.4	5	103	0.089	11.29
	300	78.3	10	222	0.183	5.45
	350	80.78	40	909	0.256	3.90
	400	81.12	86	1968	0.27	3.69
	500	81.62	185	4232	0.275	3.64
Search	100	51	2	54	0.029	34.00
ItemsByCat	150	31.25	2	34	0.044	22.73
	200	33.45	2	37	0.053	18.85
	250	35.6	2	40	0.062	16.18
	300	38.38	3	47	0.075	13.36
	350	41.28	4	58	0.101	9.88
	400	43.16	5	73	0.138	7.23
	450	46.14	8	116	0.189	5.28
	500	50.88	34	513	0.267	3.74

Table 5: Correction Factors for Various Services

approximated against processor utilization. A piecewise linear function is developed to express *CI* as a function of utilization which is calculated using polyfit function in Matlab and is given by

$$CI(U) = \begin{cases} -0.5632 \times U + 38.75 & \text{for } U <= 58 \\ -0.1434 \times U + 15.71 & \text{for } 58 < U < 85 \\ 3.69 & \text{for } U \geq 85 \end{cases} \tag{V.7}$$

Equation V.7 is then used from within MVA algorithm to compute the response time in each iteration.

V.1.4.3 Modifying Mean Value Analysis Algorithm

As described in Section V.1.4.1, we develop a multi-class closed queuing model for RUBiS as shown in Figure 62. An approximate MVA algorithm based on the Schweitzer [46]

Algorithm 1: Modified Mean Value Analysis

Input:
R Number of Job Classes
K Number of Devices
$D_{i,r}$ Service Demand for r_{th} job class on i_{th} device
N_r Number of clients for r_{th} class

Output:
Response Time $R \leftarrow$ vector containing response time for all classes of jobs

begin
// Run initial MVA with lowest service demand
while $Error > \varepsilon$ **do**
// Initialization ....
for $r \leftarrow 1$ **to** R **do**
for $i \leftarrow 1$ **to** K **do**
$\mathbf{D_{i,r}} = \mathbf{OSD_{i,r}}(\mathbf{U_r})$ // Call function for Service Demand with device utilization as parameter
$R_{i,r} = D_{i,r} \times (1 + \mathbf{CI}(\mathbf{U_r}) \times n_r)$
end
$X_r = \frac{N_r}{Z_r + \sum_{i=1}^{K} R_{i,r}}$
end
// Error = Maximum Difference in Utilization between successive iterations
end
end

assumption can be used to solve this model and calculate performance values, such as response time, number of jobs in the system, and device utilizations for closed systems [46]. We developed an approximation to the original MVA algorithm as shown in Algorithm 1. Some details in the initialization phase are not shown due to space constraints.

The algorithm starts by assuming that the clients are evenly balanced across all the devices and then adjusts the clients in the various devices iteratively. In each iteration, the algorithm computes the number of clients on each device, response time, utilization and throughput of each job type. It continues this iteration until the error in the number of clients in each device reduces below a given minimum.

The boldface parts shown are the places where the original MVA algorithm is modified

to include the functions for overall service demand and refined correction factor. The function $OSD_{i,r}$ represents the service demand function for r^{th} job class in the i^{th} device while function $CI(U_r)$ is the Equation V.7. Both of these functions need device utilizations which is computed on every loop. They also need an utilization value which needs to be provided for the first iteration. For this reason, initially the first iteration is run using the lowest value of overall service demand for each service as given by Equations V.2, V.3 and the value of CI equal to the number of processors in the system.

V.2 Conclusion

This chapter presented techniques to develop analytical models of distributed component based systems. It first showed how basic queuing models can be developed which nicely estimated response time, utilizations and throughput of the system. These models work very well under low utilization, independent transactions and single processor. Unfortunately in the real world, there is mostly high utilization, dependent transactions and multiple processors. The second part of the chapter shows how traditional queuing models can be enriched with empirically collected data so that the models become more accurate and robust. The pitfall of this technique is that there is the need to profile any new application or hardware that is introduced. But this may not be a serious disadvantage as long as such events are within a limit.

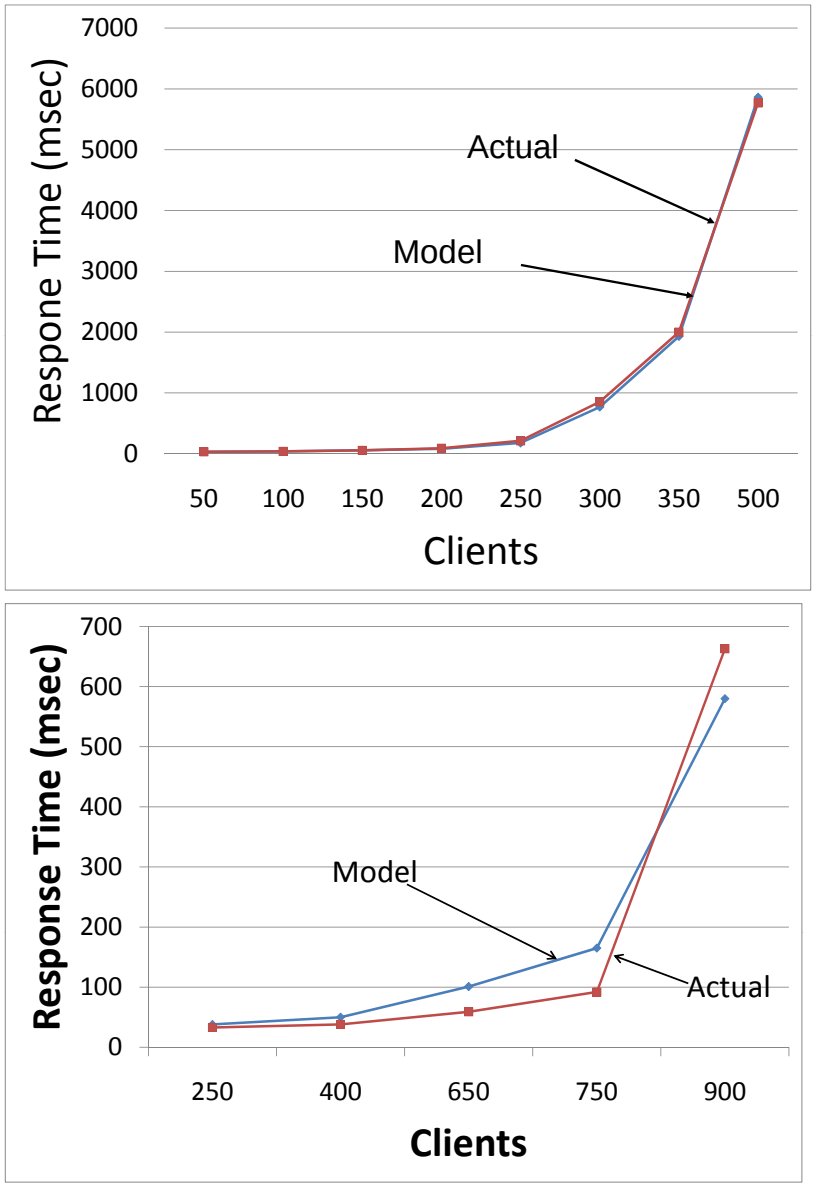


Figure 26: Response Time in Multiple Processor Machine

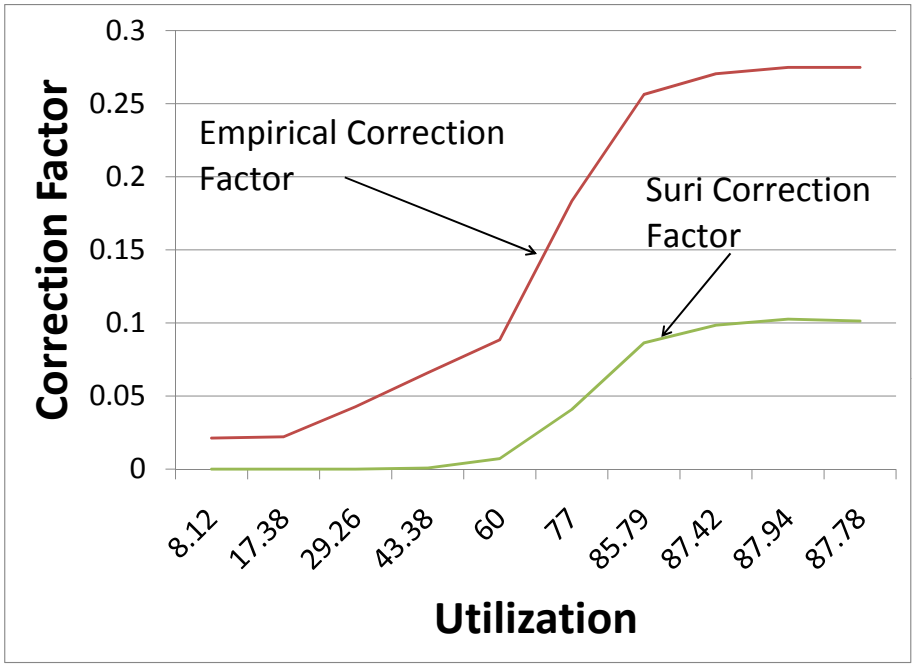


Figure 27: Comparison of Empirical Correction Factor with Suri Proposed

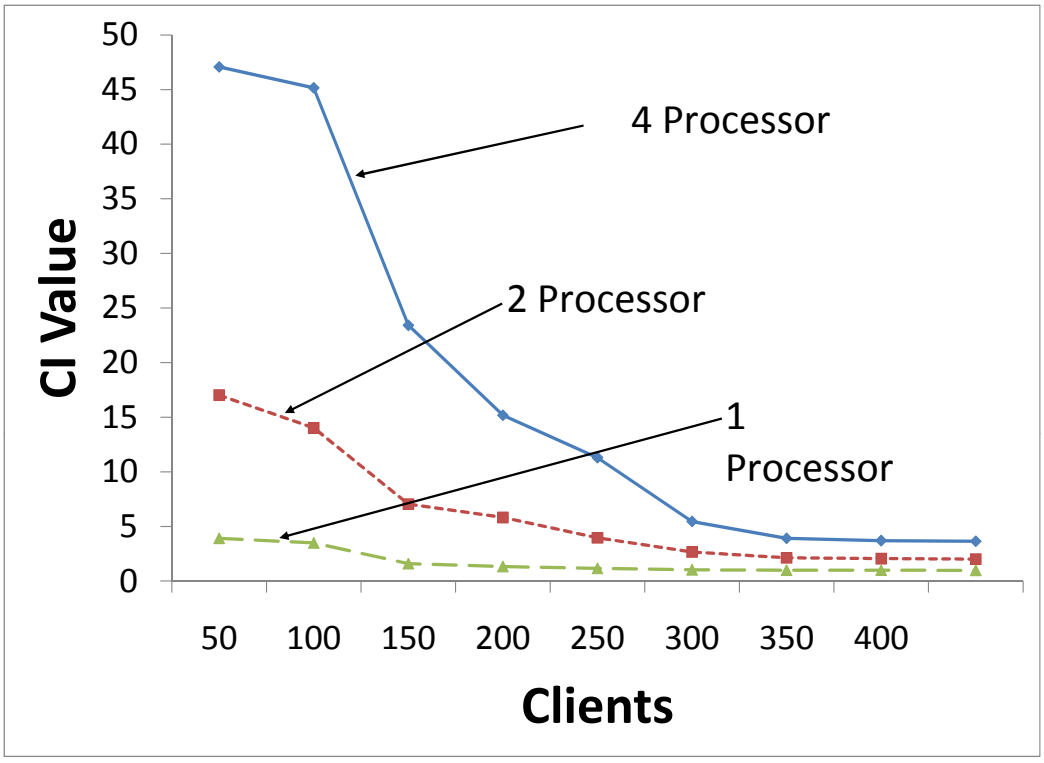


Figure 28: Inverse of Correction Factor (CI)

CHAPTER VI

MODELING SOFTWARE CONTENTION USING COLORED PETRI NETS

The previous chapter discussed about creating a performance model using queuing networks. This model is an analytical one and can be included as part of an algorithm. But analytical modeling can be approximate in some cases. If the requirement is to get more accurate models which closely mimics the actual scenario then simulation models are created. This chapter discusses simulation modeling and goes through a case study using a multi-threaded application. It shows how a simulation model is created of a multi-threaded application and how it is used to configure the application so that the objective of the application is maximized. The next section motivates the requirement of proper software configuration in the face of modern day hardware characteristics such as multiple cores/processors.

VI.1 Model Driven Application Configuration

S ervers, such as database servers or web servers, typically receive incoming requests, process them, and then returns responses to the requesting clients. One way to improve the response time of a server is to create multiple threads to service requests. Each incoming request can be assigned to a thread that processes it and prepares the response.

With the growing adoption of multi-core and multi-processor machines, software applications require multi-threading to leverage hardware resources effectively [73]. In theory, multi-threading can significantly improve system performance. In practice, however, multi-threading can incur excessive overhead due to *software contention* (*e.g.*, mutually exclusive operations needed to mediate thread access to shared data) and *physical contention* (*e.g.*, access to hardware resources, such as CPUs and memory). There is a trade-off between (1) increasing the number of threads to decrease client response time vs. (2) a larger number of threads causing bottlenecks that can increase response time.

What is needed, therefore, is a technique for selecting the optimal number of threads, which depends upon various factors including the underlying hardware, multi-threading architecture, and application logic.

In conventional multi-threaded systems, application developers and deployers make these decisions manually using their experience and intuition, which can be tedious and error-prone. Moreover, when workloads change, it is hard to estimate the effect on application performance since there is no explicit and analyzable model of application component behavior. As a result, performance problems typically emerge late in the software life-cycle during the integration phase, where they are more costly to fix.

Solution approach → Optimize an application configuration using simulation models. This chapter presents and evaluates a method for modeling the software and physical contention of multi-threaded applications to estimate the number of threads needed to produce optimal performance using a particular set of hardware resources.

This method constructs a simulation model of a complex multi-threaded application using *Colored Petri Nets* (CPNs) [27], which are a discrete-event modeling language that extends Petri nets with a "color" for each token. A CPN model of a system is an executable model consisting of different states and events, along with a notation that represents the time taken to trigger events. CPNs are suited for modeling concurrency, communication, and synchronization among different system components. Our work uses *CPN tools* [28], which help construct and analyze CPN models via an engine that conducts simulation-based performance analysis using the functional language Standard ML [48].

CPNs are used in this chapter to model simultaneous resource possession for a target tracking application containing many threads sharing multiple locks. The application is first profiled and runtime performance data is collected, which is used to parameterize the CPN model. The CPN model is then run to predict application performance under various configurations. The predictions are compared with measured data to validate the CPN

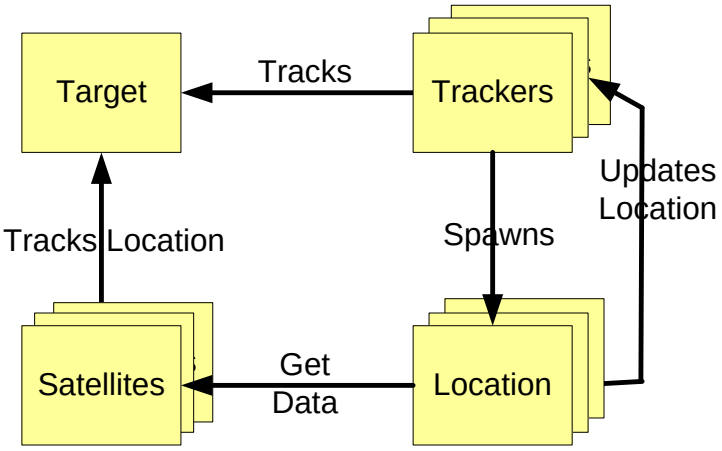


Figure 29: Active Objects in Target Tracker

model. This chapter describes the challenges that are addressed in building the CPN model and using it to predict the behavior of the target-tracking application.

VI.2 Application Case Study: Target Tracking Simulator

This section describes the application that is created and used as a case study to evaluate the work on performance prediction of multi-threaded applications.

VI.2.1 Overview of the Target Tracker

The case study involves a target-tracking simulation application composed of active objects [65], such as target, tracker, and satellites shown in Figure 29.

There can be multiple instances of trackers and satellites; each tracker collects the target's latest location from a satellite. To increase the probability of finding the target, the application must be configured with the right number of trackers and satellites.

Each active object has its own thread and executes methods of its own object, *i.e.*, there is a one-to-one correspondence between an active object and a thread. Every active object executes its application logic as shown in Figure 30. Sometimes an active object interacts with the other active objects to exchange data, *e.g.*, each tracker collects data from the satellite during every period. An active object therefore performs a periodic task that

sleeps for a specified length of time, wakes up and performs some work, and goes back to sleep, as shown in Figure 30.

As evident from the Figure 30, each active object has its own control flow and can block contending for shared data with other objects. The following active objects are defined in the application case study (shown in Figure 29):

• ***Target,*** which simulates a target that moves through an area and tries to evade its trackers. Every time it wakes up, it randomly calculates a new direction and velocity and goes to sleep again. While sleeping, it moves in a particular direction with designated velocity. There is one instance of the target in the application.

• ***Satellites,*** which gather information of the latest position of the target. Within the application, the latest coordinates of the target is placed in a global variable that each satellite reads periodically.

• ***Trackers,*** which pursue the target by obtaining its latest position via the location objects described below. Each tracker recalculates its new direction and velocity every period depending on the target's latest position. It also checks if it "hits" the target, *i.e.*, if its current position is within some small distance of the target.

• ***Tracker location updates,*** which are created by trackers for each satellite present in the application. The location objects periodically call on the satellite, obtain the latest position of the target, and update the local database within the tracker. Each pair of satellite and tracker objects are associated with a location active object.

Although the target object does not exhibit any contention with any other object, the other objects contend with each other. As shown in Figure 30, the "Update tracker DB" activity in the tracker flow contends with the "Update Data" activity in the Location flow. Likewise, the "Get new position of target" activity contends with the "Get latest target position" activity on the satellite flow. The blocking time on these locks increases when the number of objects increases which also increases the number of threads.

VI.2.2 Case Study Application Goals

Our case study application is designed to track down the target a maximum number of times. In theory it may appear that the chances of hitting the target grows with an increased number of satellites and trackers, though in practice this approach may increase contention, which can decrease tracker and satellite throughput, as well as decrease their effectiveness and increase the time to hit the target. In particular, increasing the number of active objects or threads might improve application performance but it could also degrade performance by increasing bottleneck contention. Application deployers will therefore benefit from a technique that can determine the optimal number of trackers and satellites needed to hit the target in the least amount of time.

VI.2.2.1 Predict Application Performance

The first goal of our case study is to predict the performance of the target tracker application under configurations that differ in terms of the number of tracker and the satellite objects. The notation that is used to depict each configuration is: # of target objects_# of tracker objects_# of satellite objects. Thus, a configuration of 1_2_3 means that there is 1 target, 2 trackers, and 3 satellites. As mentioned in section VI.2.1 there is a location object for each pair of tracker/satellite. As a result, the configuration 1_2_3 would have 2x3=6 location objects, resulting in a total of 1+2+3+6=12 objects. Since there is a single thread per active object, this means there are 12 threads in the application for this configuration.

The application is observed until the target performs 500 periods. The target completes one iteration of sleep and computation per period, as shown in Figure 30. The application runs two scenarios: (1) with all locks and (2) with no locks. The latter method is obviously incorrect from a functionality point of view but it quantifies the impact of contention and blocking on performance.

The accuracy of the prediction is not important; the key point is that the relative performance characteristics should be captured by the model, *i.e.*, the performance patterns/trends

should be predicted. For example, the model should tell if the average throughput of the tracker decreases or increases when a particular configuration is changed. The magnitude of the difference is less important.

VI.2.2.2 Extract Optimal Configuration

The performance data predicted by a simulation model of the application is used to choose the best configuration for the application, where "best" is defined as the greatest likelihood of the trackers hitting the target. To use the model predicted data, a utility function is used that quantifies the chances to hit the target the most number of times by maximizing the following factors:

• **Tracker activity** should maximize $N_{tr} * \mu_{tr}$, where N_{tr} is the number of trackers configured in the application and μ_{tr} is the average throughput of each tracker. This expression represents the number of times a tracker activity takes place in unit time, *e.g.*, per second.

• **Location updates** should maximize $N_{tr} * \mu_{loc}$, where μ_{loc} is the average throughput of the location object for each tracker. This expression represents how frequently the latest position is updated to the tracker.

• **Satellite throughput** should maximize $N_{sat} * \mu_{sat}$, where N_{sat} is the number of satellites configured and μ_{sat} is the average throughput of each satellite. This expression represents the number of times the satellite updates the latest location of the target.

The chance of hitting the target with N_{tr} trackers is expressed by the function $H(N_{tr})$ and is computed as:

$$H(N_{tr}) = N_{tr} * (\mu_{tr} + \mu_{loc}) + N_{sat} * \mu_{sat} \tag{VI.1}$$

The configuration that maximizes the value of this function should provide the preferred application setting, which can be computed by predicting tracker, location, and the satellite throughput and using them in the above equation.

VI.3 Experiments

This section discusses how a model of the application case study described in Section VI.2 is created and validated the model against profiled data.

VI.3.1 Application Profiling

Experiment design. The application case study is profiled under various thread configurations to collect performance data which is used to calibrate and validate the simulation model. The experiments run on a single CPU, Intel Pentium, 1.70 GHz machine with 1 GB RAM. The OS is Windows XP Professional Version 2002 with service pack 2. This application runs until the target completed 500 iterations. The time taken by the target is recorded (T_{tg}), along with the number of iterations of other objects or threads. After this data is recorded the throughput of satellite and location are measured. The throughput of the satellite is defined as N_{sat}/T_{tg}, where N_{sat} is the number of iterations of a satellite. Likewise, the throughput of the location is N_{loc}/T_{tg}, where N_{loc} is the number of location iterations.

To capture the throughput and response time of different threads, the activities of their associated active objects are profiled. Application methods of the target object were instrumented to include timestamp recording. Instrumentation code was inserted into the satellite and tracker objects to count the number of iterations.

Experiment results. After inserting the instrumentation code, the application case study is run for 13 different thread configurations and collected the profiled data. The results are shown in Table 6. Each row of the Table 6 contains the data recorded for a single configuration.

Analysis of results. The results in Table 6 show variability which is non-intuitive. For example, the data for the configuration 1_0_1 (with 1 target and 1 satellite) in the table shows a throughput of 3.70 iterations/sec for the satellite active object, whereas the throughput of the satellite active object in configuration 1_0_2 (*i.e.*, with 1 target and 2

With Mutex					Without Mutex			
Config	Target run time (secs)	Satellite Through-put (period-s/sec)	Tracker through-put (period-s/sec)	Location Through-put (period-s/sec)	Target run time (secs)	Satellite Through-put (period-s/sec)	Tracker Through-put (period-s/sec)	Location Through-put (period-s/sec)
1_0_0	140	_	_	_	140	_	_	_
1_0_1	135	3.706	_	_	135	3.71	_	_
1_0_2	130	3.85	_	_	130	3.84	_	_
1_0_3	135	3.7	_	_	135	3.7	_	_
1_1_1	138	2.12	65.89	2.69	139.2	3.58	61.89	3.01
1_2_1	137	1.29	67.85	1.44	135.3	3.68	29.5	3.12
1_3_1	138	0.91	68.79	0.99	131.77	3.79	18.54	3.19
1_1_2	144	2.15	51.43	2.62	131.74	3.78	54.31	3.18
1_2_2	144	1.26	54.49	1.39	130.7	3.81	23.32	3.17
1_3_2	145	0.89	55.92	0.95	153.2	3.24	9.61	2.68
1_1_3	144	2.18	42.96	2.7	132.6	3.76	44.64	3.10
1_2_3	145	1.28	47.20	1.42	170.39	2.94	10.73	2.43
1_3_3	145	0.91	48.95	0.97	212.5	2.37	4.24	1.91

Table 6: Profiled Data from the Application

satellites) is 3.85 iterations/sec. The throughput for satellite objects therefore increases as the number of satellites increase. When the number of satellites increases to 3, however, the throughput decreases since CPU utilization increases due to higher contention.

Such effects can also be seen from the response time of the target in Table 6. For example, when the target active object runs on its own (1_0_0) the time taken to complete 500 iterations is 140 secs, where when a satellite active object runs concurrently with it (1_0_1) the time reduces to 135 secs. This variability arises either from cache effects or operating system jitter. Caching could cause this difference since the target and the satellite active objects perform similar arithmetic computations, so as the number of satellite objects increase the cache effects become apparent until the CPU utilization reaches a certain threshold, after which the response time starts to increase.

There was no real time scheduling used in the experiments so fluctuations in performance could also arise from OS jitter. Petrini et al [55] and Kramer et al [37] show how OS jitter can cause variability in performance. For simplicity, the term “cache effects” or “OS jitter” will be used to refer to such variability in the chapter.

VI.3.2 Colored Petri Net Model Construction

The simulation model of the application case study using Colored Petri Nets (CPNs) is now explained. Figure 31 shows a screenshot of the CPN tool and our application modeled using CPN. The four aspects of the application that are part of the system modeling process include (1) *modeling application flow*, which models the logic of each object similar to the workflows shown in Figure 29, (2) *modeling lock contention*, which models the waiting and acquiring on the software locks, *i.e.*, process scoped mutexes, also known on Windows as "critical sections," (3) *modeling resource access*, which models the concurrent access of the physical resources by each thread, and (4) *modeling cache effects/OS jitters*, which models the variability in computation time due to simultaneous threads performing similar work on the CPU. Below, the modeling of these four aspects is elaborated.

VI.3.2.1 Modeling Application Flows

Colored Petri nets model application flows via *places*, *transitions*, and *tokens*. Each transition moves tokens from the input places to the output places. The placement of a token in a place indicates the location of control within the application thread.

Figure 32 shows the application flow of the thread in the active object. In this figure places are connected through transitions. Whenever the input places has a token, the connected transition can fire and move the token. Control therefore moves from each place to the next corresponding to the workflow shown in Figure 29.

Figure 32 shows how sleep is used to implement a delay that simulates the interval where the task fires. Transition firing times of the second and third transitions model physical device access, which is the CPU in this case. As seen in the figure, when the device access is completed control flows back to the starting position.

VI.3.2.2 Modeling Lock Contention

Colored Petri nets can also model contentions. For example, Figure 33 shows a portion of a CPN model where the threads in the satellite and location active objects contend for a shared lock. The place named "lock" represents the software lock, which is available if a token is present in that place. The places in the thread flow named "Wait on lock" model the thread waiting on the lock. If the token is available, the transition on a single thread is executed and the token moves out of the place "lock," which causes the other thread to block until the token again becomes available.

VI.3.2.3 Modeling Resource Access

CPNs can model resources (such as the CPU) similarly to locks. Multiple objects contend for the CPU, but only one thread at a time can access it. A place is therefore created in the model to represent the CPU and every object has a connection to it.

Since the CPU is accessed by all threads, the model becomes visually cluttered. A feature of hierarchical nets of the CPN tool can be used, however, to move the place representing the CPU to a different page of the CPN model. It is then referred from every flow. The broken arrows connecting the two places shown in the Figure 32 represent the underlying contention for the CPU. Figure 34 shows the model of the CPU.

VI.3.2.4 Modeling Cache Effects/OS Jitters

Cache effects/OS jitters were observed during profiling, as discussed in Section VI.3.1. These effects should be incorporated within the CPN model so the model predicted performance data is as close to the actual values as possible. Figure 35 gives an empirical formula that is implemented within the place representing the CPU. This formula calibrates the execution time of a thread running on the CPU. The formula decreases the execution time of a thread as the inter-arrival time between threads decreases.

The 'tint' variable in the formula represents the current inter-arrival time. If 'tint' is less

than 180 the execution time is modified to 94% of the original. In the extreme, if it is less than 35, the execution time is modified to 40% of the original. The percentage numbers above were computed by calibrating the CPN model via repeatedly running it with the data from configurations 1_0_0, 1_0_1, 1_0_2, 1_0_3 in Table 6. The various percentage values were tweaked multiple times until the response time of the target thread in the model converged to the empirical data.

VI.3.3 Calibrating the Model

The techniques described in Section VI.3.2 helped implement the CPN model of the application. It is then calibrated using the profile data gathered as described in Section VI.3.1. Some of the profile data are used as a training set to tune the model parameter; the rest of the data are used to validate the model. The data for the configurations 1_0_0, 1_0_1, 1_0_2, 1_0_3 in Table 6 are used to train the model. These timing data were used to tune the formula to model the caching shown in Figure 35. The model is repeatedly run with the different configurations and the various percentage values in the formula is tweaked multiple times to converge to the above values shown in Figure 35.

Once the model is properly calibrated, it is run for the remaining configurations. For each configuration, the response time of the target thread and the throughput of the satellites and the location threads are calculated. Table 7 gives the resulting model prediction data,

VI.3.4 Model Validation

The results from profiling the application (Section VI.3.1) is now compared against the model prediction results (Section VI.3.2). These results are explained from the perspective of two conflicting factors: (1) the CPU hardware resource bottleneck and (2) the software lock contention due to shared data accessed by various threads. Results are presented with different thread configurations on the x-axis and the runtime performance metric on the y-axis.

With Mutex					Without Mutex			
Config	Target run time (secs)	Satellite Through-put (period-s/sec)	Tracker Through-put (period-s/sec)	Location Through-put (period-s/sec)	Target run time (secs)	Satellite Through-put (period-s/sec)	Tracker Through-put	Location Through-put (period-s/sec)
1_0_0	140	_	_	_	140	_	_	_
1_0_1	135	3.69	_	_	135	3.70	_	_
1_0_2	130	3.83	_	_	130	3.85	_	_
1_0_3	135	3.69	_	_	135	3.69	_	_
1_1_1	132	3.59	73.12	2.77	130	3.83	51.22	3.26
1_2_1	132	3.69	77.41	1.54	135	3.70	25.67	3.16
1_3_1	132	3.79	76.74	1.02	144	3.49	11.06	2.87
1_1_2	143	3.78	30.58	2.25	143	3.48	8.30	2.89
1_2_2	144	3.81	38.49	1.35	170	2.96	4.94	2.18
1_3_2	144	3.24	39.49	0.92	191	2.57	4.05	1.66
1_1_3	153	3.76	7.19	1.54	170	2.94	4.62	2.05
1_2_3	162	2.94	13.10	1.12	209	2.37	3.39	1.26
1_3_3	162	2.37	13.79	0.79	237.77	1.99	2.79	0.95

Table 7: Model Predicted Data

VI.3.4.1 Target Thread Response Time

Figure 36 shows that the response time of the thread in the target active object remains nearly constant as the number of objects are varied in the application case study. This result occurs for two reasons (1) the target does not contend with other objects, so it does not face any extra blocking as the number of other objects increases and (2) as the number of objects increases, the threads in these objects block each other due to software locks, which keeps the CPU relatively free so the target thread can use the CPU when needed.

This result seems non-intuitive since the underlying hardware is a single CPU machine. It seems reasonable that increasing the number of threads in an application running on a single CPU should increase the overhead and reduce the performance of each thread. The results in Figure 36, however, show how the performance of a thread that uses no software locks will increase when more threads that *do* use locks are added to the application.

VI.3.4.2 Throughput of Satellite and Tracker

Figure 37 shows the satellite thread behavior with locks in the system. Each set of three configurations in this graph (*e.g.*, data for configuration 1_1_2, 1_2_2 and 1_3_2) should

be considered together. Between the former configurations the number of location threads are increased, which increases contention and decreases throughput since the threads now spend more time blocked on the locks. The location thread also exhibits a similar trend as the satellite data, as shown in Figure 38.

Tracker throughput is shown in Figure 39. The error percent in model data is larger compared to other data, but the general trend of the application behavior is captured. For example, in each set of three successive readings with one satellite (1_1_1, 1_2_1 and 1_3_1), two satellites (1_1_2, 1_2_2 and 1_3_2), and three satellites (1_1_3, 1_2_3 and 1_3_3) the throughput increases as the number of trackers increase. This application behavior trend helps identify the optimal thread configuration. The accuracy of the prediction is less important since it is only important in determining if a configuration is better than another, not how much better they are.

VI.3.4.3 Performance Metrics with the Locks Removed

For this experiment all the locks in the application are removed, which clearly compromised its behavior since shared data could be corrupted due to simultaneous modifications by multiple threads. The locks are removed, however, to compare the performance of each thread and show the impact of using locks in the system. The CPN model is also modified and used to predict the performance of the system. The model predicted data is shown along with the measured data in the Figures 41, 42 and 40.

Figure 41 shows the target thread response time, which increased as the number of objects increased. In this case, when the number of other objects increased they do not block each other and directly contend for the CPU, which increases the waiting time of the target at the CPU and its response time. Figure 42 shows the behavior of the satellite thread when there are no locks in the system. When the data in Figure 42 is compared with Figure 37, it is clear that throughput degrades less as the number of threads or objects increase due to

the fact that there are no bottleneck due to locks. Nevertheless, the throughput still goes down due to the increased CPU contention.

VI.3.4.4 Model Prediction

Although the CPN model accurately predicted the underlying trend in application behavior in the experiments described above there were errors in the model prediction. Some specific points have inconsistencies, *e.g.*, configuration 1_1_3 seems to indicate problems since the throughput of tracker and location predicted by the CPN model is much less than the actual value. Figures 39, 38 and 40 show that the model prediction differs significantly from the actual data. Potential reasons for these differences include (1) there is increased OS activity due to context switching or other activities that increase the throughput of the thread and/or (2) some form of cache effects cause this behavior. Overall, however, the CPN model mimics the application behavior, so developers and deployer can use these models to estimate application behavior accurately.

VI.4 Application Configuration

This section demonstrates how the performance data predicted by the model can be leveraged to optimize application thread configurations. In particular, our case study used the results presented in Table 7 to find the optimal thread configuration. To verify the decision made using the model, the application is profiled and the number of hits made by the trackers are calculated for each configuration.

Equation(VI.1) is used from Section VI.2 to compute the hit chance value for each configuration, as shown in Table 8. The average throughput values of tracker and satellite are used from the model predicted data in Table 7. Table 8 shows configuration 1_3_1 maximizes the trackers hitting the target, as explained in Section VI.2, so this configuration

Config	Tracker Num.	Tracker Through-put (period-s/sec)	Location Through-put (period-s/sec)	Satellite Number	Satellite Through-put (period-s/sec)	Hit chance
1_0_0	0	_	_	0	_	0
1_0_1	0	_	_	1	3.69	3.69
1_0_2	0	_	_	2	3.83	7.67
1_0_3	0	_	_	3	3.69	11.09
1_1_1	1	73.11	2.77	1	2.32	78.20
1_2_1	2	77.41	1.54	1	1.38	159.28
1_3_1	3	76.74	1.01	1	1.36	234.63
1_1_2	1	30.58	2.25	2	2.18	37.19
1_2_2	2	38.49	1.35	2	1.37	82.43
1_3_2	3	39.49	0.91	2	1.26	123.75
1_1_3	1	7.19	1.53	3	2.16	15.22
1_2_3	2	13.10	1.12	3	1.30	32.37
1_3_3	3	13.79	0.79	3	1.13	47.18

Table 8: Target Hit Chances for Various Configurations

should thus be optimal. To verify whether this configuration is optimal, the running application was then profiled to record the number of times the trackers hit the target, as shown in Table 9.

The validity of Equation(VI.1) needs to be verified as a right quantifier of the application performance. The measured value of tracker, location, and satellite throughput are therefore used to compute the value of the equation for each configuration (omitted due to lack of space). Using these values each configuration is ranked and it matches exactly with the ranking given by the data from actual hit counts(Table 9) except one configuration, 1_2_3. This result proves that Equation(VI.1) is a reasonable estimator of the application performance.

This table shows that configuration 1_3_3 has the highest number of hits, which validates that the configuration chosen using the modeled data and the utility function given by equation(VI.1) is optimal. Comparing the data shown in Table 8 and Table 9, there exists quite a few discrepencies in the ranking of the configurations. This is due to the error in

Config	Tracker 1	Tracker 2	Tracker 3	Total Hits
1_0_0	_	_	_	0
1_0_1	_	_	_	0
1_0_2	_	_	_	0
1_0_3	_	_	_	0
1_1_1	212	_	_	212
1_2_1	127	142	_	269
1_3_1	220	222	230	672
1_1_2	163	_	_	163
1_2_2	111	121	_	232
1_3_2	183	190	179	552
1_1_3	130	_	_	130
1_2_3	159	144	_	303
1_3_3	148	153	161	462

Table 9: Runtime Target Hit Occurrences

the prediction of the throughput of the various active objects. If the error is reduced, the prediction will be more accurate.

The results above show how a simulation model can be used to determine the optimal configuration of threads for our case study application. Combining simulations with profiling helps application deployers optimize the performance of application thread configurations without the need for tedious and error-prone manual effort.

VI.5 Related Work

Prior work has explored techniques for modeling software contention using analytical techniques and modeling thread contention using Petri nets. This section compares and contrasts our work with this related work.

In [44] and [46] two queueing network models are created: (1) a *hardware queueing network* model of the physical contention and (2) a *software queueing network* model of the software contention. Each model is solved iteratively until the results from the two converge to within a predefined value.

Our CPN-based approach uses a simulation model rather than an analytical model to improve model accuracy. Although simulation models require more time to predict performance [46] they are appropriate for our purposes since the models are analyzed before application deployment. Our solution is also based upon modeling of the application flow and does not require detailed knowledge of queueing-theoretic techniques or simultaneous resource possession. Domain experts with good knowledge of the application can therefore readily create a simulation model using CPN.

Queuing Petri Nets are used in [36] to model the performance of distributed component-based systems. [36] conducts a case study of the performance evaluation of a J2EE application server and then presents a performance evaluating method for modeling thread contention in a load balancer used with the application server. The focus in [36] is on modeling the number of threads in a thread pool for the load balancer. In contrast, our work

models the thread contention caused from software locks and hardware resources, which is complementary to the work in [36].

Analytic performance models of software servers are developed in [45], which also studies the thread contention due to usage of thread pools. [45] develops a queuing-theoretic analytical model to obtain the optimal number of threads in a software server that uses a fixed number of threads in a pool. The underlying assumption in the use case is that each service provided by a thread does not contend with any other thread for software locks. Unlike our work with CPNs, [45] does not evaluate the problem of software contention due to software locks.

A Petri net model of an application is presented in [31], which captures software contentions and models software locks in a manner similar to ours. The main difference is that [31] does not consider the case of multi-level resource contention, *i.e.*, a thread performs its entire computation once it acquires a software lock. In contrast, in our approach a thread waits for a software lock and then contends for the hardware resource, which is more representive of common multi-threading scenarios.

Simulation-based performance of web servers [81] has created a simulation based model of a web server. [81] models physical resources, such as CPU, disk, and network, but does not consider the complex interation between software resources and hardware resources. In contrast, our approach also models both these resources.

VI.6 Concluding Remarks

The work presented in this chapter describes a technique that is developed to model and simulate software contention. Colored Petri Nets (CPN) is used to validate the model data with the results captured by profiling the application. CPN models the non-determinism inherent in the case of multiple threads contending on a single lock. Profiling is performed to measure application runtime performance and the resulting data is validated against data

predicted by the CPN model. The results show that the CPN model accurately predicts the pattern of behavior in the application within certain error limits.

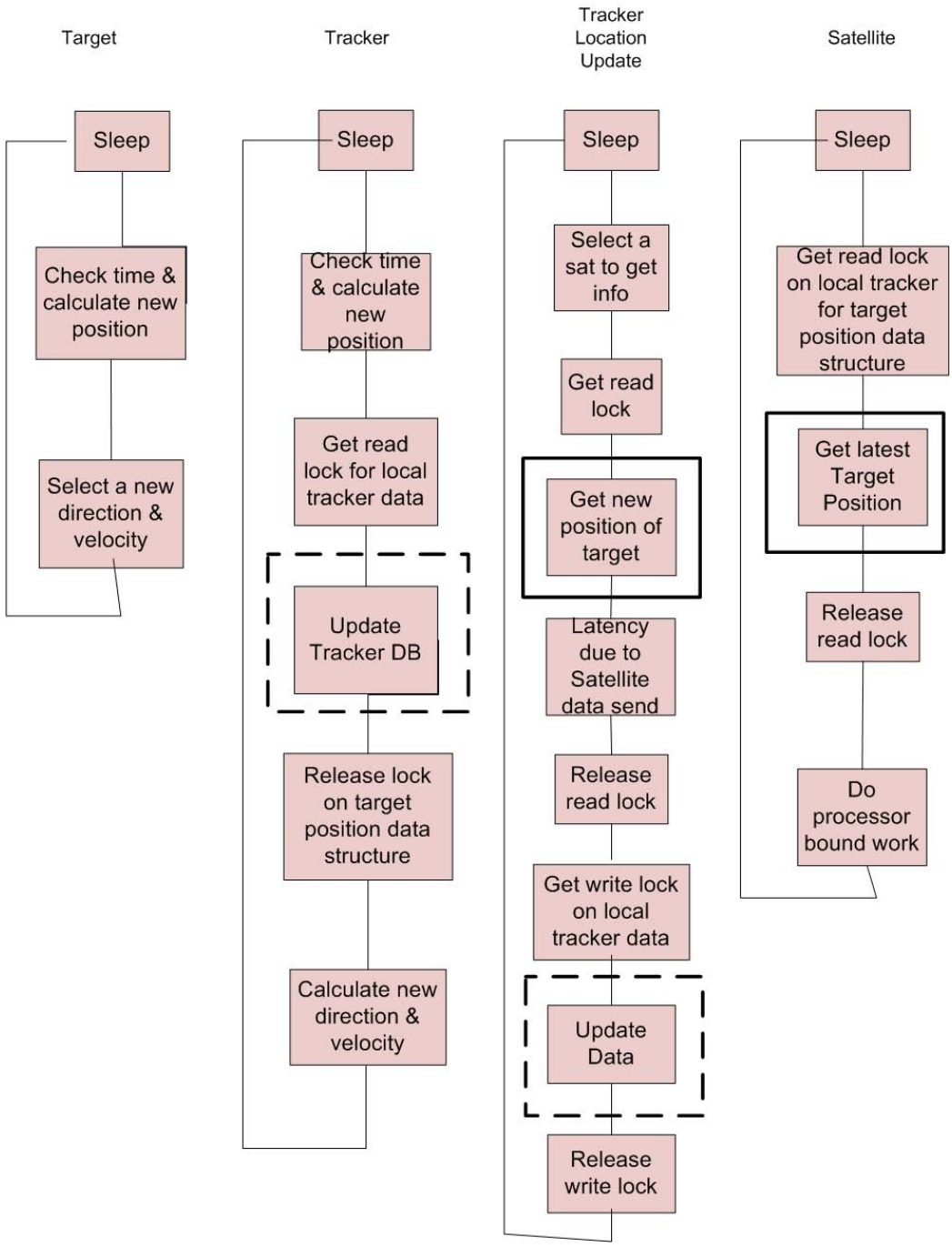


Figure 30: Application Logical Flows in the Target Tracking Simulator

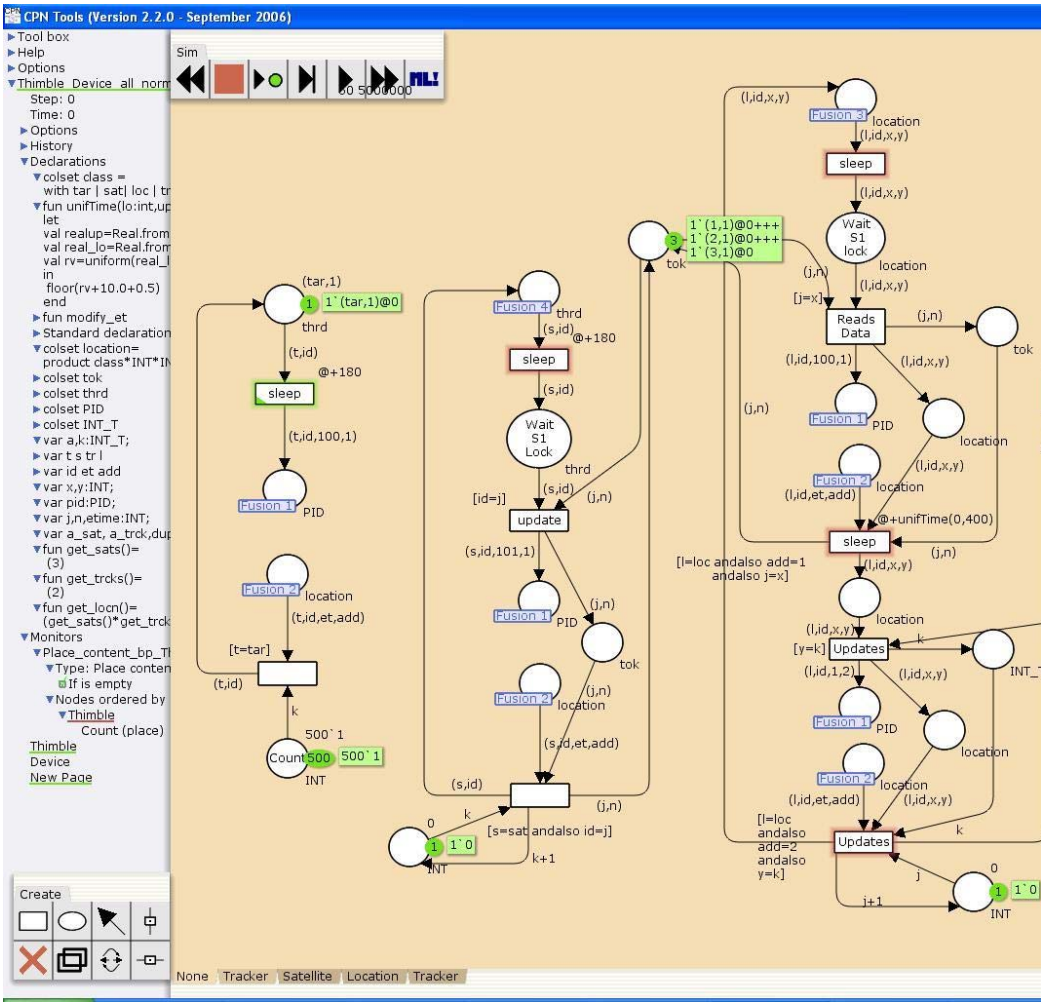


Figure 31: CPN Model of Application Case Study

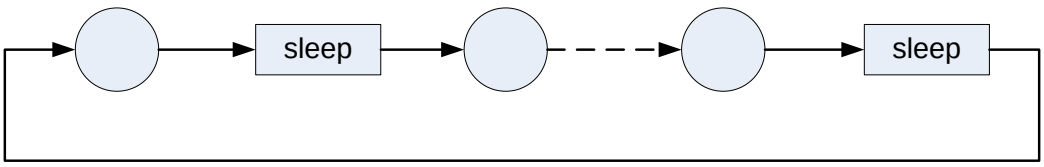


Figure 32: A CPN Model of Target's Active Object Thread

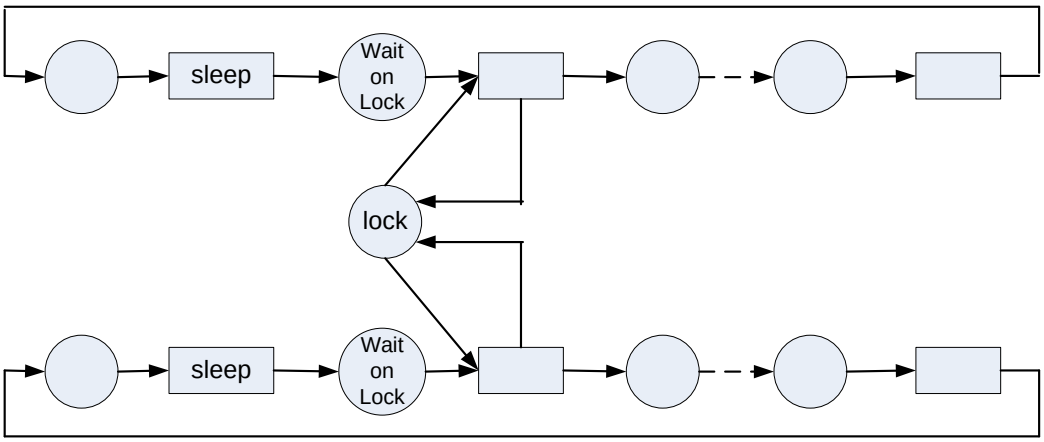


Figure 33: Contention Model for Software Lock

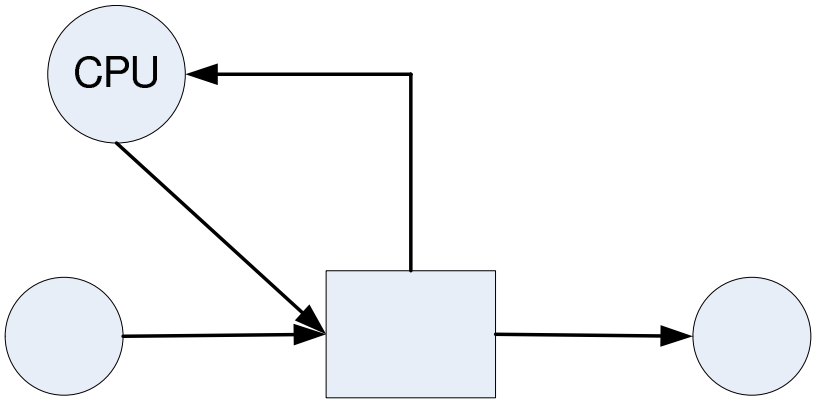


Figure 34: The CPN Model of CPU

```
if (tint < 35)
  then
   val mod_et = modify_et(et,40)
   else
     if (tint < 45)
          val mod_et = modify_et(et,80)
     else
         if (tint < 180)
            val mod_et = modify_et(et,94)
          else
            val mod_et = modify_et(et,100)
```

Figure 35: The Formula for Cache Effects

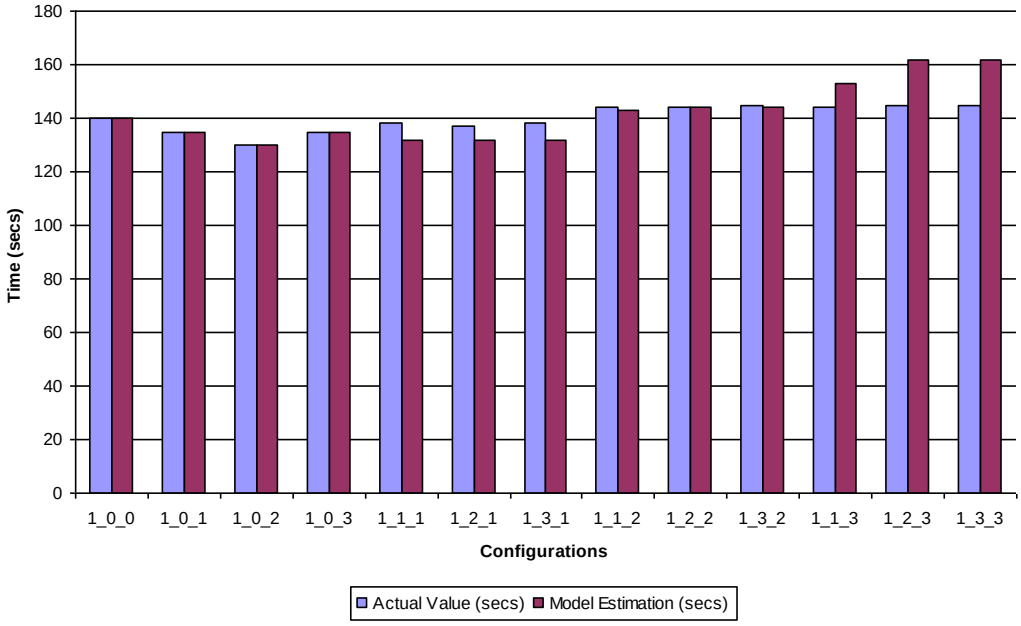


Figure 36: Response Time of Target Thread with Locks

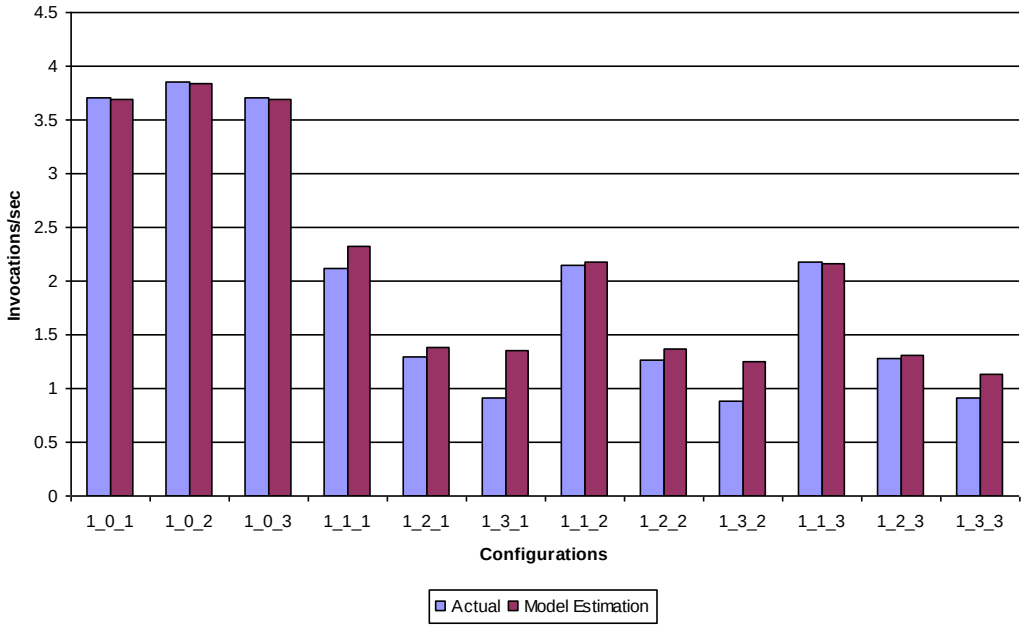


Figure 37: Throughput of Satellite Thread with Locks

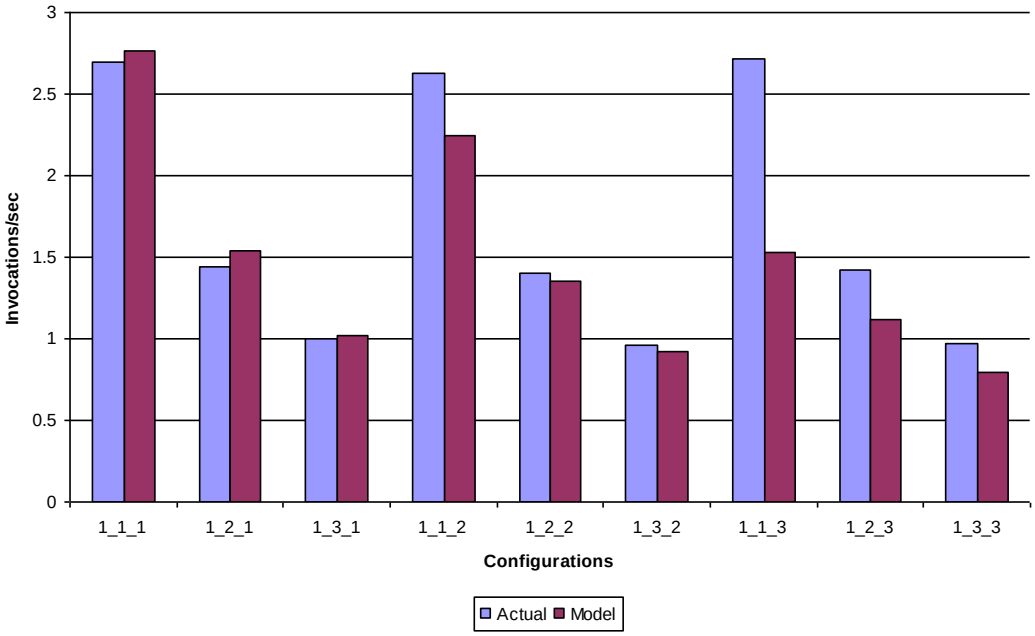


Figure 38: Throughput of Location Thread with Locks

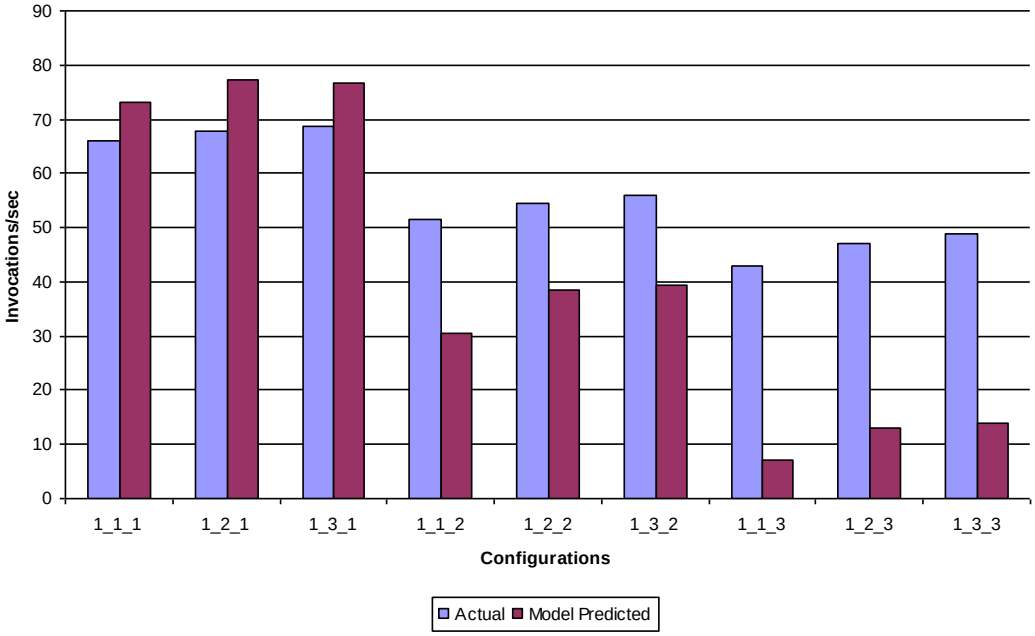


Figure 39: Throughput of Tracker Thread with Locks

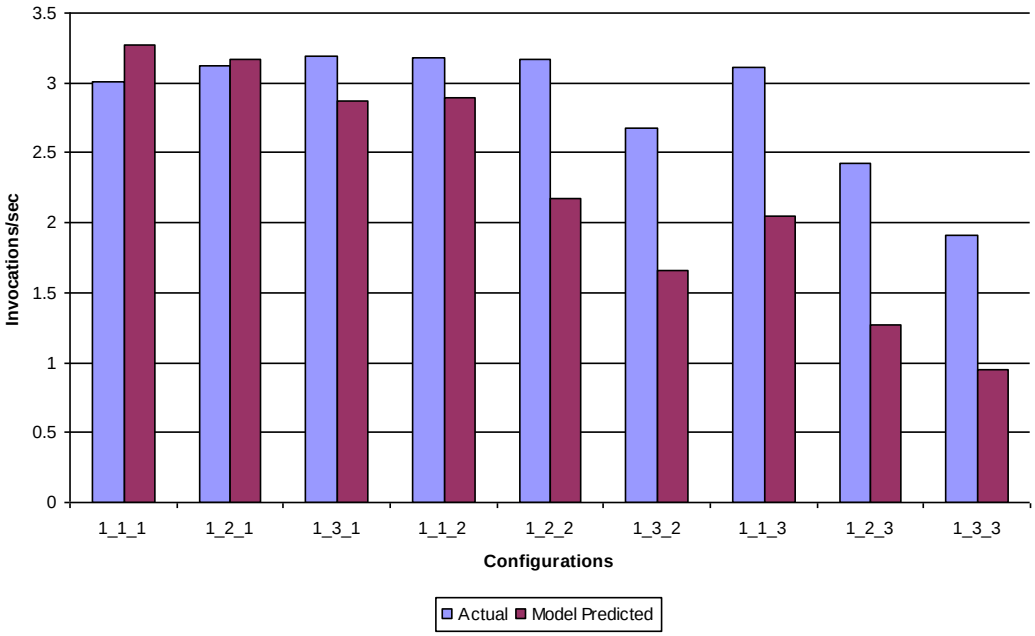


Figure 40: Throughput of Location Thread without Locks

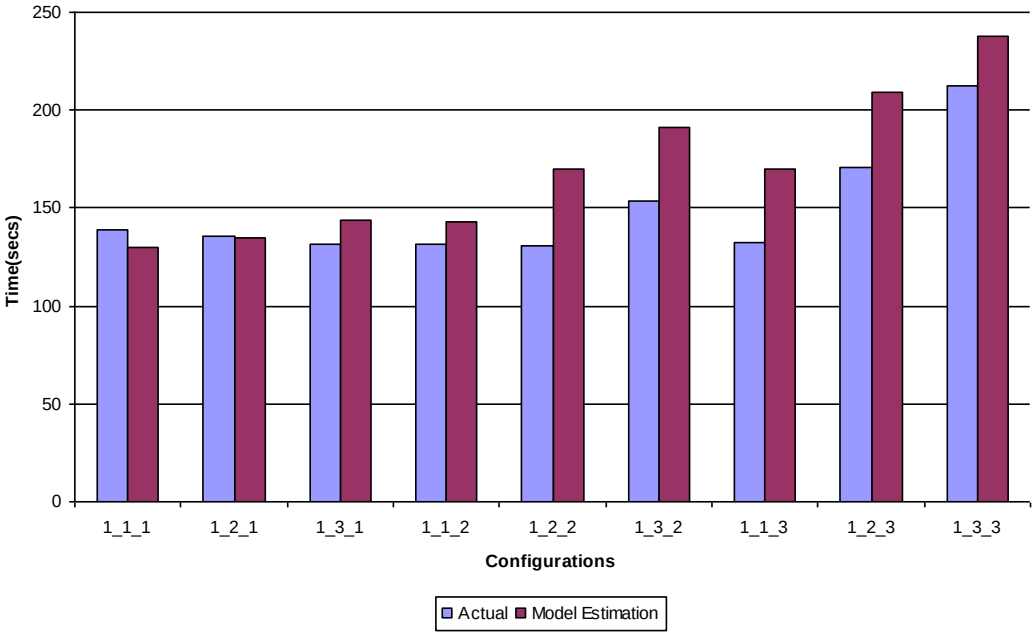


Figure 41: Response Time of Target Thread without Locks

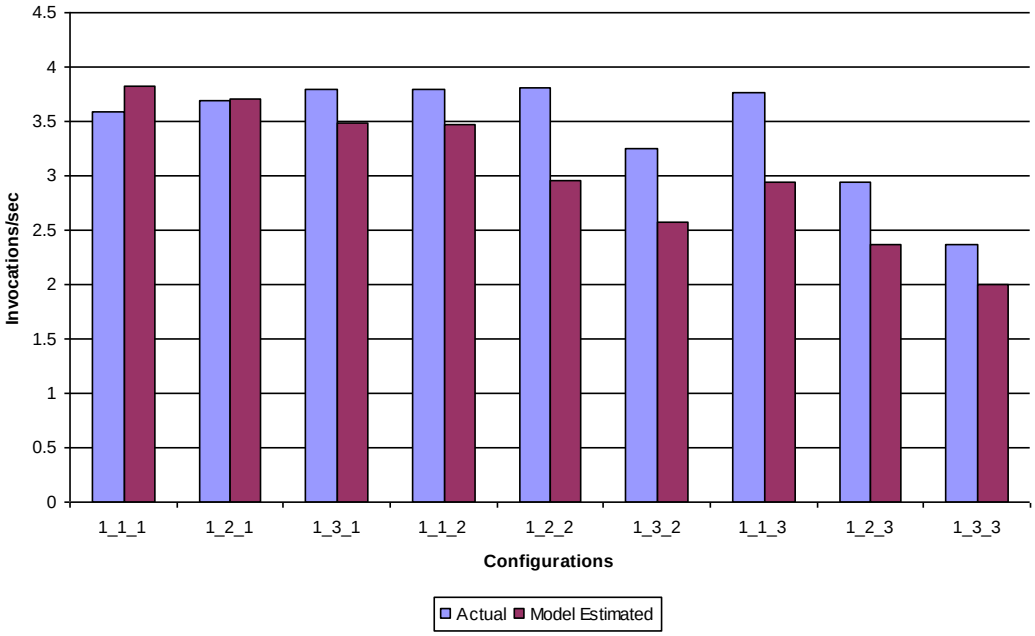


Figure 42: Throughput of Satellite Thread without Locks

CHAPTER VII

MODELING OF REAL TIME SYSTEMS

This chapter looks at creating performance models of real time systems. Real time systems has the property of completing the tasks withing a specified deadline. Hard real-time systems incur an unacceptable cost when any deadline is missed. Soft real-time systems can tolerate some degree of missed deadlines to either (1) improve resource utilization and system throughput or (2) provide some degree of quality-of-service (QoS) when the task arrival, deadline, and/or execution times cannot be known deterministically.This chapter presents a novel technique for modeling soft real-time systems. The next section discusses basic real time concepts.

VII.1 Real Time Systems

Soft real time systems such as live audio-video or wireless sensor networks are becoming increasingly popular in our society. Such systems can tolerate some number of deadline misses and still provide some value as long as deadline misses are within an acceptable bound. These systems typically function in an uncertain environment where the underlying infrastructure, such as wireless communication, is unpredictable. This causes data exchange between distributed entities of the system to be somewhat unpredictable, which in turn causes inter-arrival times of events to be variable. Because input to such systems (*e.g.*, events captured by a sensor or frame sizes of audio/video signals) are variable, the processing time of each event may vary across non-trivial ranges.

This chapter presents a formal technique of modeling and analyzing soft real time systems. The method of stages [39] is used to model the variability and uncertainty present in arrival and execution times of all tasks in the system. This technique, Method of Stages based Analysis of soft Real Time systems (MoSART), models a soft real time system and

estimates the deadline misses and other important metrics such as response time, throughput, and resource utilizations for each task. Recent work [18, 33] in this area only calculates response time distributions not other system parameters. Others have considered arrival time as deterministic [41]. MoSART is useful to predict various system parameters given a particular deployment of the tasks onto a set of nodes.

VII.2 Motivating Example

This section presents two motivating examples which illustrates how variability in task arrival and execution times can impact system performance. Such variability is a direct result of the uncertainty/non-determinism present in the environment in which such systems operate.

A highly loaded system: Consider a highly loaded system consisting of two tasks, one which is highly variable (*i.e.*, Task 2) in its arrival times, while the other task (*i.e.*, Task 1) has less variance in its arrival times. Both tasks have the same average arrival and execution times. Subsequent arrivals are assumed to impose a deadline for the previous task arrivals. This system is simulated and the deadlines met for each task is obtained when each task is given scheduling priority. The task details are given in Table 10.

	Inter-Arrival Time		Execution Time			
Tasks	Mean (secs)	C.V.	Mean (secs)	C.V.	Deadline Met % Priority Task 1	Deadline Met % Priority Task 2
Task 1	10.00	0.32	6.0	0.32	86.91	44.60
Task 2	10.00	1.00	6.0	0.32	27.66	55.83
Total	_	_	_	_	57.28	50.22

Table 10: A highly loaded system

Although the tasks are identical except in their second moment (*i.e.*, coefficient of variation (CV) = standard deviation/mean), giving priority to the less variable Task 1 improves the

number of met deadlines by over 14%. This indicates that scheduling based on the variance in the tasks impacts the number of deadlines met/missed.

A lightly loaded system: Now consider the exact same set of tasks but with one exception. The mean inter-arrival time of the tasks is increased to make the system lightly loaded. As before, the two tasks are identical except for their second moment. The results are shown in Table 11. In this case the number of deadlines met is better when priority is given to the task with higher variance. This result is opposite to what was seen in the earlier, highly loaded, system. Though this is a simple example, it clearly indicates that both the variance and the system load should be considered when making scheduling decisions. Simplistic techniques (*e.g.*, RM, EDF, LL), that only consider the mean and ignore the variance and the system load, are suboptimal. Therefore, it is necessary to develop a technique capable of modeling the variance in the tasks as well as the variability in the system load. Such a technique can then be used to estimate crucial system parameters such as the deadline miss rate, response time, and resource utilization of a given system.

	Inter-Arrival Time		Execution Time			
Tasks	Mean (secs)	C.V.	Mean (secs)	C.V.	Deadline Met % Priority Task 1	Deadline Met % Priority Task 2
Task 1	30.00	0.32	6.0	0.32	99.96	98.80
Task 2	30.00	1.00	6.0	0.32	77.43	82.03
Total	_	_	_	_	88.69	90.42

Table 11: A lightly loaded system

VII.3 Problem Formulation

In this chapter a real-time system is viewed as a set of N independent tasks. Each task T_i consists of a series of job arrivals. $J_{i,j}$ represents the jth instance of task T_i and has a deadline that corresponds to the next arrival of the task, along with an expected execution time. Both the inter-arrival time and the execution time are variable and are considered

to be random variables with a probability distribution. It is assumed that the distribution of both is available from historical data. The problem is to estimate the percentage of deadlines being missed/met, the expected task response time, and the resource utilization when a set of tasks is assigned to a processor, given a specified scheduling algorithm(*e.g.*, rate monotonic, earliest deadline first).

VII.4 Modeling Approach

This section describes the methodology underlying MoSART. In its simplest form, the system is viewed as a single processor (*i.e.*, resource) that is shared by several tasks. For simplicity, and without loss of generality, it is assumed that the deadline of a job coincides with the subsequent job arrival from the task, (*i.e.*, each job $J_{i,j}$ is expected to finish before the next job $J_{i,j+1}$ of task T_i arrives).

VII.4.1 Workload Modeling

Workload modeling consists of three parts: the inter-arrival time, the deadline, and the execution time. For simplicity reasons, since the deadline is assumed to coincide with the next arriving task, only the inter-arrival time and the execution time processes need to be modeled explicitly.

As illustrated in Section VII.2, it's important to model the variance present in task parameters. Thus both the inter-arrival time and the execution time of tasks are treated as random variables with a probability distribution. Phase-type distributions [39] are used to approximate these distributions. An Erlang distribution using the method of stages (MOS) is a special form of a phase type distribution which helps in modeling distributions with a coefficient of variation (CV, which is the ratio of the mean to the standard deviation) less than 1. In the case of a distribution having a CV greater than 1, an analogous technique using a hyper-exponential method of stages can be used. However, in this chapter, the discussion is restricted to distributions with CVs less than 1.

The MOS is composed of a series of exponential stages, each with the same mean (λ). The probability density function (pdf), mean, variance, and coefficient of variance (CV) for a k-stage Erlang distribution are given in Table 12.

PDF	Mean	Variance	CV
$\frac{(k\lambda)^k}{(k-1)!}x^{k-1}e^{-k\lambda x}$	$\frac{1}{\lambda}$	$\frac{1}{k}(\frac{1}{\lambda})^2$	$\frac{1}{\sqrt{(k)}}$

Table 12: Erlang distribution parameters

When k is equal to 1, the pdf reduces to that of an exponential distribution, which has a CV of 1. Increasing the value of k decreases the variance of the distribution.

VII.4.2 System Modeling

This section describes the task representation, system model parameters, resource allocation, task arrivals, task execution, and task deadlines. Together with a specified scheduling algorithm, a comprehensive modeling framework is provided.

VII.4.2.1 Task Representation

The system workload is viewed as a collection of tasks. Jobs belonging to a particular task are statistically identical to each other, but the jobs of different tasks may have different arrival and service (*i.e.*, execution) characteristics. Each task is modeled by a set of arrival stages and a set of execution stages.

The arrival stages model the time that the next job in a particular task arrives to the system. Completion of all the arrival stages indicates an arrival of a new job. The execution stages represent the execution time of a job in a particular task and completion of the last

execution stage indicates job completion. A job will meet its deadline if its execution stages complete before the arrival stages complete.

Figure 43 shows the representation of a particular task stream. In this example, there

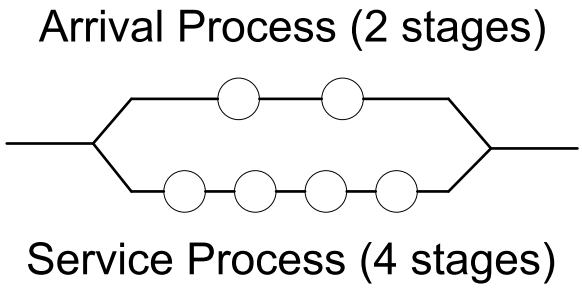


Figure 43: Task Representation

are two arrival stages and four execution stages. The arrival stages are statistically identical to each other. Similarly, the execution stages are statistically identical to each other.

A detailed description and example of the state space model is developed in subsequent sections and also in [60]. Since each stage within a particular arrival or execution process is statistically identical, and the time spent in a stage is assumed to be exponentially distributed, the completion time of each process has an Erlang distribution. Therefore, the underlying state space model is a Markov chain. Specifying the current stage of each process for each task completely describes the current system state.

VII.4.2.2 Model Parameters

The state space model depends on the following parameters.

- N - number of workload tasks.
- A_i - number of arrival stages for task i jobs.
- S_i - number of execution stages for task i jobs.

• μ_i - execution rate for task i jobs. Thus, $1/\mu_i$ is the average execution time for a job of task i to complete all of its execution stages. Similarly, $1/(S_i\mu_i)$ is the average time that a job of task i spends at each of its execution stages.

• λ_i - arrival rate for task i jobs. Thus, $1/\lambda_i$ is the average inter-arrival time for task i jobs. In $1/\lambda_i$ time task i completes all its stages, triggering a new arrival (and signalling a deadline). Naturally, $1/(A_i\lambda_i)$ is the average time that a job of task i spends at each of its arrival stages.

• P - number of execution resources(*e.g.*, processors).

• scheduling algorithm - algorithm used to determine the resource allocation among multiple concurrent tasks, example, Rate Monotonic etc. MoSART assumes that task allocation can change at each stage boundary.

VII.4.2.3 State Space Model

This section describes the state space model in the context of an example system of tasks. It is assumed that the system is pre-emptive and that a scheduling decision is made at every stage completion. Without loss of generality, a single processor system is assumed. The set of tasks given in Table 13 is used. The arrival events of task 1 are modeled using 2 stages while the execution is modeled using 4 stages. This is because the CV of each event is $1/\sqrt{(\#ofstages)}$. The stage values are shown in the CV column in Table 13. Similarly, the arrival of task 2 is modeled using 3 stages and execution with 5 stages. Thus, $N = 2$, $A_1 = 2$, $A_2 = 3$, $S_1 = 4$, $S_2 = 5$, and $P = 1$.

	Inter-Arrival Time			Execution Time		
Tasks	Mean(secs)	CV	Rate (Jobs/min)	Mean(secs)	CV	Rate (Jobs/min)
Task 1	10.0	$0.7(1/\sqrt{(2)})$	6.0	4.0	$0.5\ (1/\sqrt{(4)})$	15.0
Task 2	15.0	$0.57(1/\sqrt{(3)})$	4.0	5.0	$0.45(1/\sqrt{(5)})$	12.0

Table 13: Task Parameters

Figure 44 shows a portion of the underlying state space model that shows the possible state transitions due to an arriving task. The arrival process for task 1 has a 2-stage (*i.e.*,

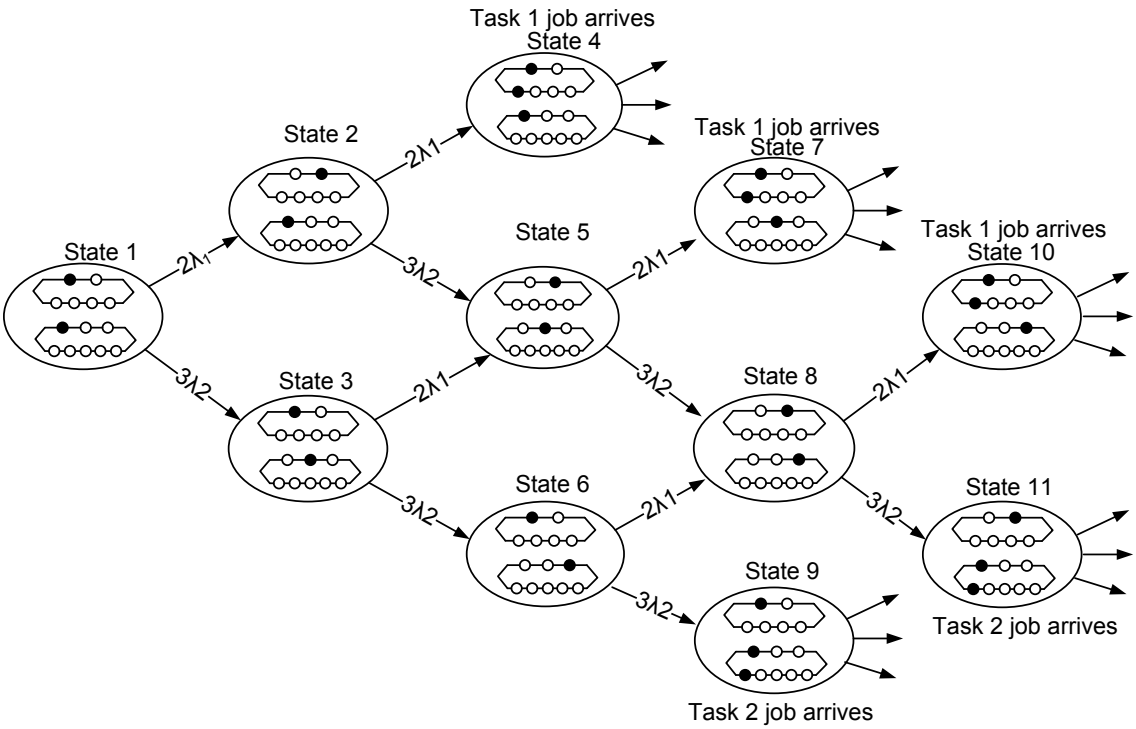


Figure 44: Task Arrivals

the top series of stages) Erlang distribution with an arrival rate of $\lambda_1(6.0)$. Thus, the rate leaving each stage is $2\lambda_1(12.0)$. Job 2 tasks have a 3-stage (*i.e.*, the third series of stages) Erlang arrival distribution with an arrival rate of $\lambda_2(4.0)$. Thus, the rate leaving each stage is $3\lambda_2(12.0)$.

The second and fourth series of stages represents the execution process for task 1 and task 2, respectively. The solid black circles (stages) in Figure 44 represent the currently active stages. For instance, in state 1, the arrival processes of the two tasks are both executing their first stage and no jobs are executing in the system .

The current stage of each process determines the current state of the job. Collectively,

the state of each task represents the overall system state. For example, $\{(1,1),(2,0)\}$ represents the state at state 7. This means the state of the first task is $(1,1)$ while that of the second is $(2,0)$. The first task arrival process is less than 50% complete and execution is less than 25% complete. Similarly second task arrival is between 33.3% and 66.6% complete while the execution has not started yet. The completion of any stage results in a change in the system state.

Task Arrival: The arc labels in Figure 44 indicate the rates at which the various state transitions occur. Initially, in state 1, neither task has arrived yet and both of the tasks are in their first stage of arrival. None of the execution processes are active. The arrival process of task 1 can complete its first arrival stage at rate $2\lambda_1$, causing the system to move to state 2. Alternatively, the arrival process of task 2 can complete its first arrival stage at rate $3\lambda_2$, resulting in a transition to state 3. From state 2, the system can either move to state 4 if the arrival process of task 1 completes its 2nd stage, or to state 5 if the arrival process of task 2 completes its 1st stage. In state 4, the arrival process of task 1 has just completed both its arrival stages, which represents the arrival of a new task 1 task to the system. Consequently, the execution process of task 1 becomes active (indicated by the black circle in task 1's execution process) and the subsequent arrival/deadline process for task 1 is restarted at stage 1.

Task Completion and Missed Deadlines - A race between stages: Figure 45 shows another portion of the underlying state space model. In state 2, three transitions are possible: (1) the arrival/deadline process of task 1 can complete a stage with rate $2\lambda_1$ which models a missed deadline due to the arrival of a new task 1 and the system enters state 5, (2) the execution process of task 1 can complete a stage with rate $4\mu_1$ and the system enters state 6, or (3) the arrival/deadline process of task 2 can complete a stage with rate $3\lambda_2$ and the system enters an unshown state indicated by the tiny arrow. If transition 1 occurs before 2 or 3, the task misses a deadline (as shown in the figure). Similarly, the transition from state 6 to state 11 represents an execution completion for task 1. This figure

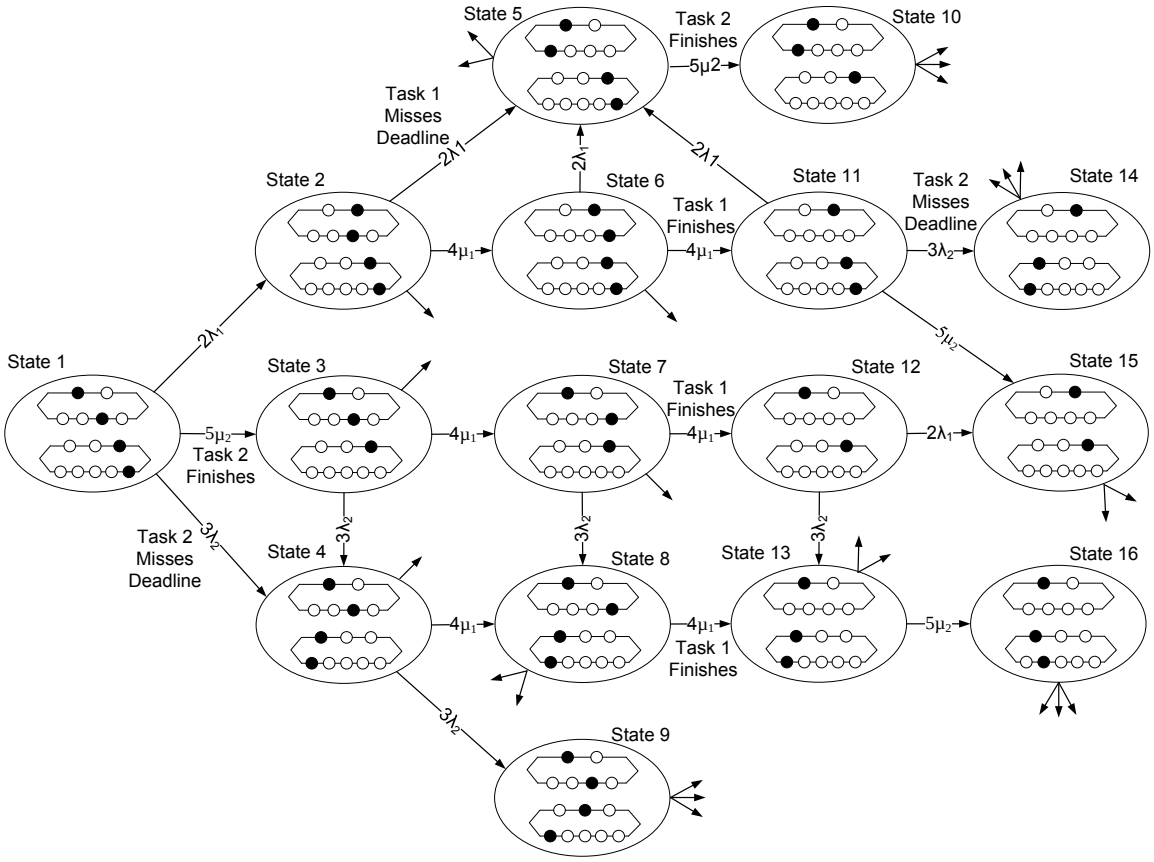


Figure 45: Task Execution/Deadline - EDF

also shows the same for task 2 (*i.e.*, a met deadline in going from state 1 to 3 and a missed deadline in going from state 1 to 4). A job can terminate in two ways, either the last stage of its arrival completes or the last stage of its execution process completes. If the execution stages completes first, then the job meets its deadline. On the other hand, if the arrival stages completes first, then the job misses its deadline. The model can therefore be thought of as a horse race. At task arrival time, two "horses" (*i.e.*, an arrival/deadline horse and a execution horse) begin racing. Each goes at its own pace and begins running through its successive stages. The first horse crossing the finish line (*i.e.*, its last stage) wins. If the execution process horse wins, the task successfully meets its deadline. If the arrival/deadline horse wins, the task misses its deadline. This approach requires a synchronization between the arrival and the execution processes. This is what distinguishes MoSART from other approaches such as matrix-analytic methods [39] who also use phase-type distributions.

Modeling Scheduling Algorithms: The scheduling algorithm modeled in Figure 45 is earliest (expected) deadline first. In the given set of tasks, the arrival/deadline stages have the same expected time in both the tasks. The same is true for the execution stages of each task. This allows the reader to simply count the number of remaining arrival/deadline stages to determine which task has the earliest expected deadline.

For example, in state 1, task 1 has 1 remaining arrival stage until its next arrival/deadline and task 2 has no remaining arrival stages until its next arrival/deadline. Thus, the expected deadline for task 2 is earlier than that of task 1 and for this reason, task 2 is allocated the processor to execute. Since task 1 has no allocated resources, the execution process for task 1 cannot proceed. Task 2's execution can continue as shown by the transition from state 1 to state 3 with a rate of $5\mu_2$. In state 4, however, task 1's deadline is 1 stage away, while task 2's deadline is 2 stages away. Thus, the processor is given to task 1. When the resource is allocated to task 1 (task 2), the average execution time for each stage is $1/4\mu_1$ ($1/5\mu_2$). Note that the arrival/deadline process for a task always continues to advance regardless of

any allocated resources. The various states and transitions in the state diagram are self-evident and further details can be found in [60].

VII.5 A Complete Example

This section presents a complete solution to a simple example to illustrate the modeling methodology. Table 14 presents the task parameters. The set of parameters are deliberately

	Inter-Arrival Time					Execution Time				
Tasks	Mean (secs)	Rate (jobs/min)	CV	Stages	Rate/stage (jobs/min)	Mean (secs)	Rate (jobs/min)	CV	Stages	Rate/stage (jobs/min)
Task 1	10.0	6.0	0.7	2	12.0	5.0	12.0	0.57	3	36.0
Task 2	6.0	10.0	1.0	1	10.0	2.0	30.0	0.7	2	60.0

Table 14: Task parameters

kept simple to illustrate the full solution of the example. Section IX.4 presents results for more complex sets of tasks. Figure 46 presents the complete Markov chain state space for this example. The detailed model analysis technique is presented in [60].

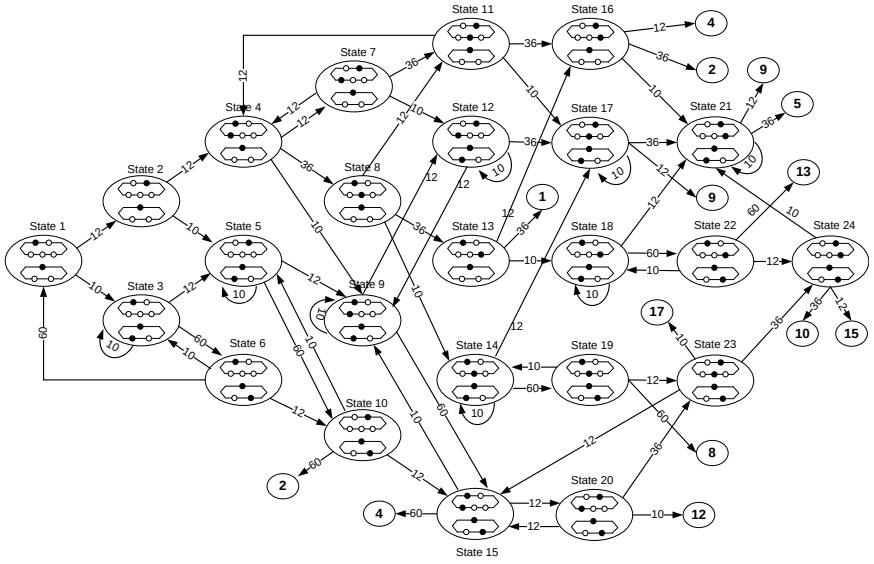


Figure 46: A Simple Example using EDF scheduling

Table 15 shows the resulting performance metrics. The data obtained while using the

Earliest Deadline First			
Tasks	Deadline Misses/min	Deadlines Met/min	Utilization
Task 1	1.97 (33%)	4.03 (67%)	0.39 (39%)
Task 2	3.30 (33%)	6.70 (67%)	0.25 (25%)
Total	5.27 (33%)	10.73 (67%)	0.64 (64%)
Rate Monotonic			
Task 1	2.32 (39%)	3.68 (61%)	0.37 (37%)
Task 2	2.66 (26.6%)	7.34 (73.4%)	0.27 (27%)
Total	4.97 (31%)	11.02 (69%)	0.64 (64%)
Least Laxity First			
Task 1	1.91 (32%)	4.09 (68%)	0.40 (40%)
Task 2	3.88 (38.8%)	6.12 (61.2%)	0.24 (24%)
Total	5.79 (36%)	10.21 (64%)	0.64 (64%)

Table 15: Example performance metrics

rate monotonic and least laxity first scheduling algorithms is also included.

The results in Table 15 show that the tasks have a high percentage of deadline misses, though the utilization of the processor is not high. The reason for this result is the high variability of the inter-arrival times and the execution times for each task in this example. The CV of the arrival and execution processes vary between 1.0 and 0.57. The variation observed in practice is typically lower.

VII.6 Broader Methodology Benefits

MoSART gives a platform to (1) conduct comprehensive sensitivity analysis, and (2) search for new, optimal scheduling algorithms.

VII.6.1 Sensitivity Analysis

By holding certain parameters constant while varying others, new insights and rules-of-thumb are possible using MoSART. In this sensitivity analysis, three systems are evaluated: (1) system A with a high utilization of 83%, (2) system B with a medium utilization of 67%, and (3) system C with a low utilization of 47%. Two tasks are modeled, with parameters shown in Table 16. For each set of parameters, the MoSART technique provides the per-

	Task 1		Task 2		
System	Arr. Rate (jobs/min)	Exec. Rate (jobs/min)	Arr. Rate (jobs/min)	Exec. Rate (jobs/min)	Expected Util
A	5	15	6	12	83.34
B	5	30	10	20	66.67
C	5	30	6	20	46.67

Table 16: Task parameters

centage of missed deadlines. The results of applying this sensitivity analysis to systems A, B, and C are shown in Figure 47, 48, and 49. The following conclusions can be derived from these figures:

• **Lower variability improves performance.** The performance of the system improves as the variability within the system decreases(see figure 47).

• **No algorithm is uniformly optimal.** In most cases, EDF outperforms other algorithms but in some cases RM and LLF also perform better (Figures 47, 48, and 49).

• **The scheduling algorithm choice is more important for higher utilized systems.** Figure 47, demonstrates a more distinct difference between the different algorithms when the utilization is higher.

• **Variance causes low utilized systems to miss deadlines.** Systems with low utilization(*e.g.*, 47%) can expect relatively high deadline miss ratios(e.g., above 20%) if the arrival and execution time variability is high(*e.g.*, CV = 1.0).

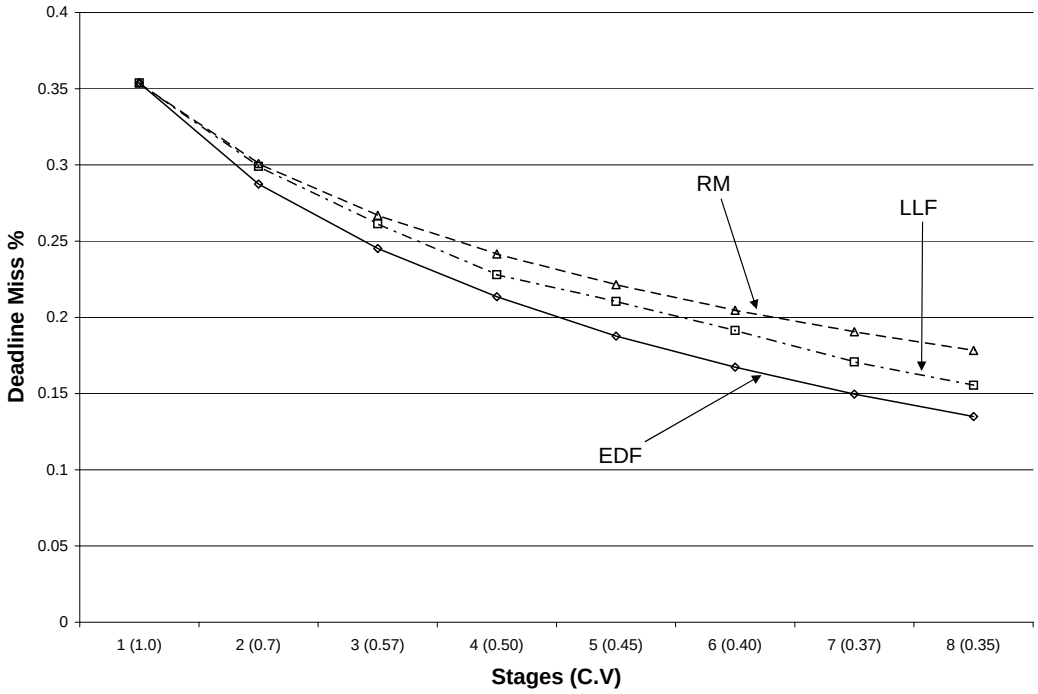


Figure 47: System with 83% Utilization

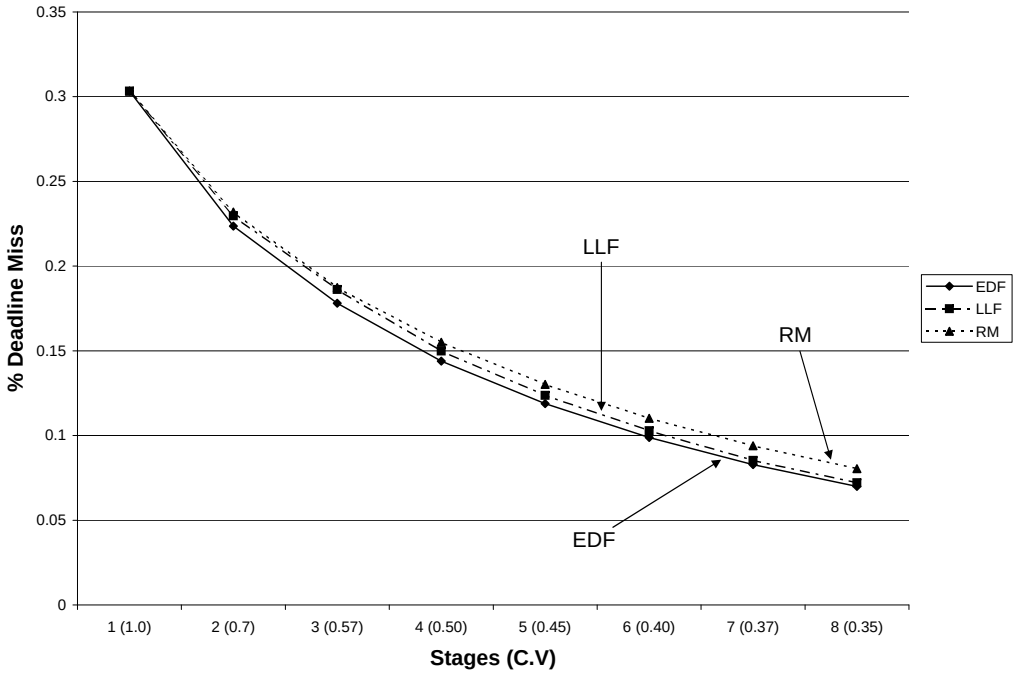


Figure 48: System with 67% Utilization

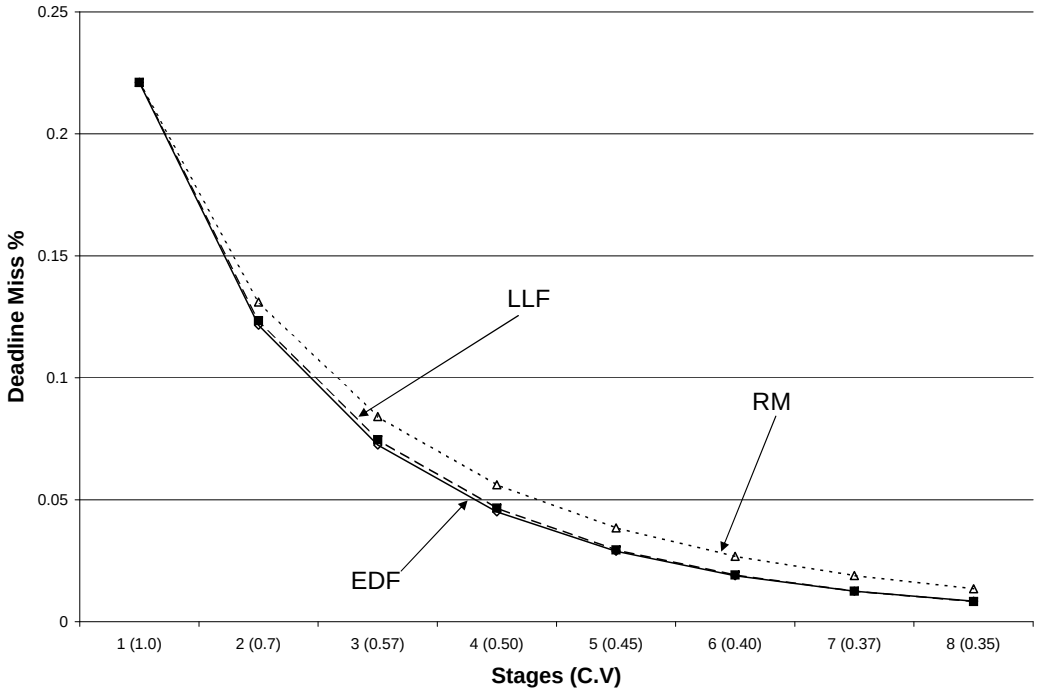


Figure 49: System with 47% Utilization

• **Missed deadlines lower the processor utilization.** As more deadlines are missed, the actual processor utilization is lower than its theoretical maximum, as shown in Figure 50.

VII.6.2 New Optimal Scheduling Algorithms

Markov Decision Processes [56] together with MoSART can be used to search for better scheduling algorithms. The decision to give the processor to the tasks at each state can be made depending on optimizing a global objective function like overall deadline miss. Such a technique is used to find a better algorithm shown in Figure 52.

The modeling approach presented here not only analyzes existing scheduling algorithms, but can also be used to search for new algorithms. The modeling methodology leads to a state space model, where in any particular state, the scheduling algorithm dictates which of the competing tasks receives execution from the processor.

For example, if two tasks are competing for the processor, under EDF the task with

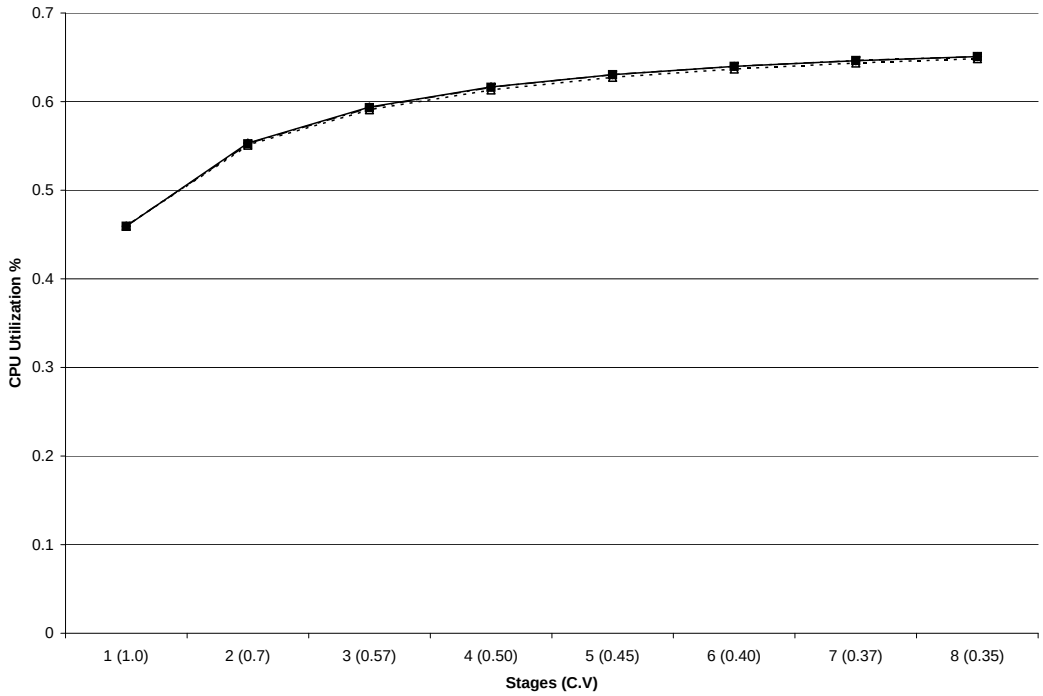


Figure 50: Utilization versus Variance

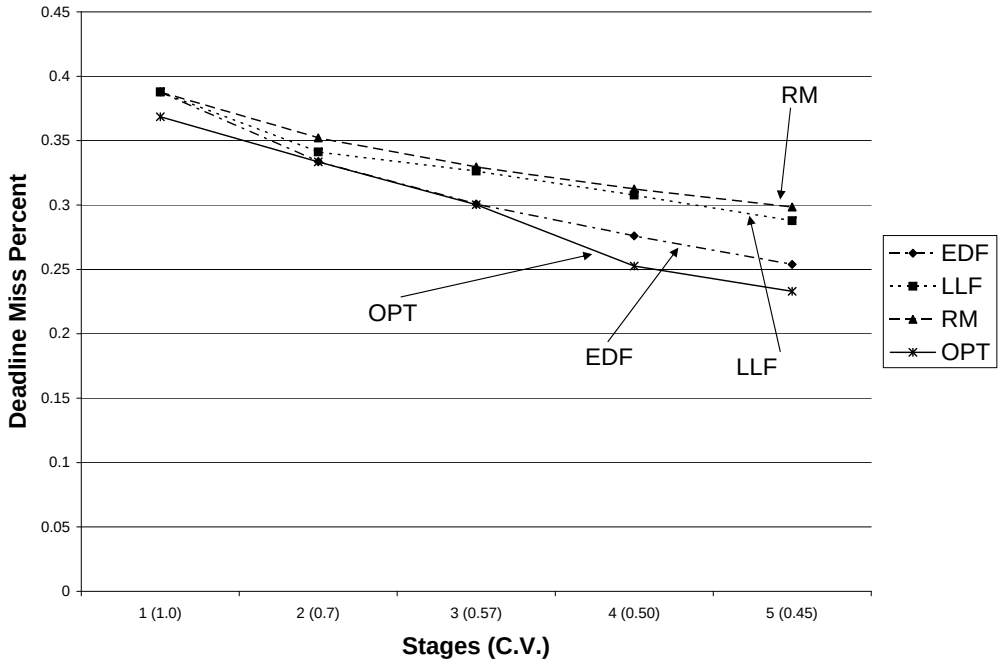


Figure 51: Optimal Algorithm (93% Util.)

the fewest number of arrival/deadline stages left is assigned the processor, which may not be optimal. If the first task's deadline is imminent—but several execution stages have not completed—the probability that the task will ultimately meet its deadline may be quite low. In this case, if the processor is allocated to the first task according to the EDF policy, not only will it likely miss its deadline, but because the processor wasted its time on a lost cause, the second task might also miss its deadline. It would be better to "sacrifice" the first task to save the second task. Such a scheduling policy is heavily state dependent, and depends upon the state (*i.e.*, the variability of the arrival and executions processes, as well as the overall system load) represented by each of the competing tasks.

The state space methodology therefore lends itself directly to searching for new, optimal, scheduling algorithms. Abstractly, in every state, an unknown probability can be assigned to how much of the processor is allocated to each of the competing tasks. In a two-task system, this leads to a p-vector, with one element per system state. Each scheduling algorithm has a unique p-vector. For example, in Figure 46, the 24-element p-vector representing EDF is given by $[xx010011001110011001 1011]$, where x represents a state where no task is in the system (and, thus, the processor is not allocated to either task), 1 represents a state where the processor is allocated to Task 1, and 0 represents a state where the processor is allocated to Task 2. Elements in the p-vector could be any number between 0 and 1, which represents the fraction of the processor allocated to Task 1. A p-vector value of 0.5 therefore represents a state where processor sharing occurs between the two tasks.

The goal of the scheduling algorithm is to make the best scheduling decision at each state of the system so that the given objective is optimized. By finding (or calculating) a p-vector that optimizes a particular objective function (*e.g.*, the minimal number of missed deadlines), and realizing that each p-vector corresponds to a particular scheduling algorithm, new optimal scheduling algorithms can be discovered.

To demonstrate the finding of such a p-vector, consider the specific example in Section VII.5. The Matlab optimization toolkit is used to compute an optimal p-vector (*i.e.*, an

optimal algorithm) for the system. This algorithm makes scheduling choices at each state of the system such that the total deadline miss percentage is minimized. The deadline miss percentage for this optimal algorithm is plotted in Figures 52 and 53.

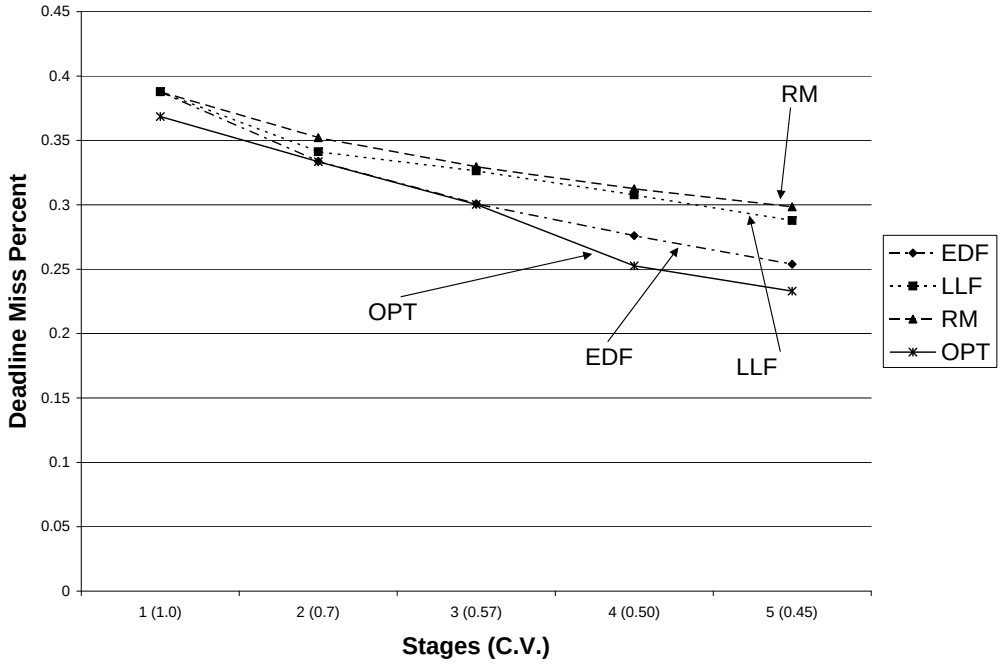


Figure 52: Optimal Algorithm (93% Util.)

The optimal p-vector algorithm is computed for two systems, one with system utilization of 83% and one with 93%. The optimal algorithm performance is plotted with that of other popular algorithms.

These figures demonstrate that there exists state dependent scheduling algorithms that outperform EDF, LLF, and RM. In general, the higher the system utilization—and the more deterministic the system—the higher the potential for improvement by using an optimal state dependent scheduling algorithm. For example, the right side of Figure 52 demonstrates that the optimal algorithm outperforms EDF by 8.2%, which outperforms LLF by 11.8%, which outperforms RM by 3.5%. By examining the p-vectors for the optimal algorithm, EDF, LLF, and RM, some (but not all) of the differences can be attributed to the

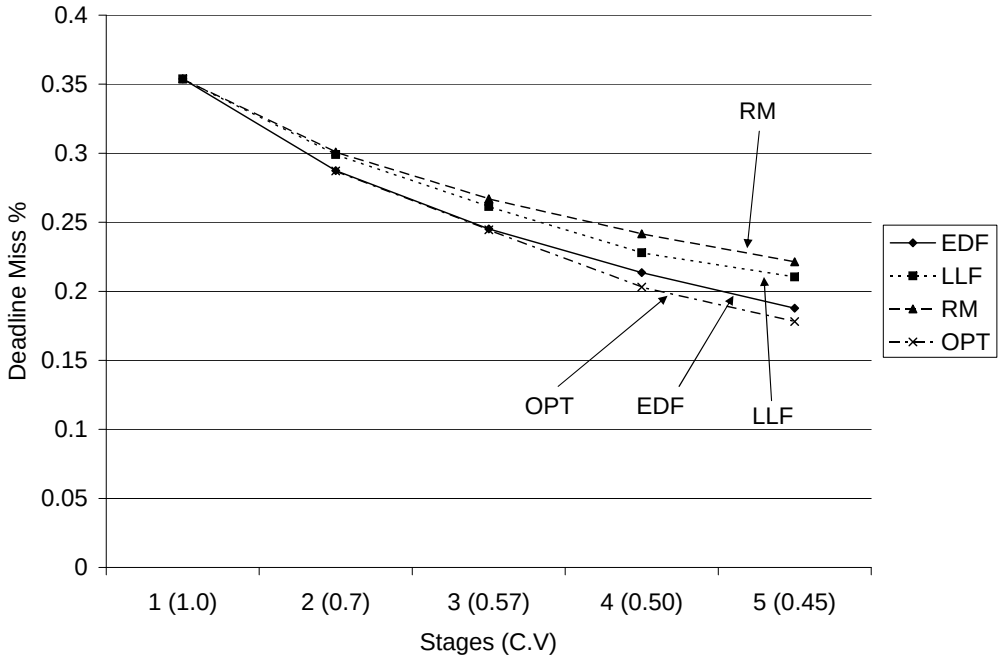


Figure 53: Optimal algorithm (83% Util.)

sacrificing of one task to save the other task. Such investigations for better state dependent scheduling algorithms is a topic of continuing research.

VII.7 Concluding Remarks

Various soft real time systems such as wireless sensor networks function under severe, unpredictable, and uncertain environments. Such conditions cause the arrival rates and processing times of events to be variable. It is important to model this variability to accurately estimate the various characteristics within soft real time systems. This chapter presents a novel technique, MoSART, for modeling and analyzing soft real time systems with variability in the inter-arrival and execution time of events.

MoSART uses the method of stages to model the inter-arrival, deadline, and execution times. It also presents an intuitive ”race" between arrival and deadline stages to model the number of deadlines met or missed for any particular job. A limitation of the proposed technique is state-space explosion. However, approximations, bounds, and simulations can

be directly applied. Addressing such limitations, applying the methodology to a wider set of system parameters, evaluating a richer set of scheduling algorithms, conducting a more extensive experimental validation, and using the methodology to discover new scheduling algorithms are promising directions for future research.

Part III

Application Placement

The previous chapters discussed the challenges of measuring component resource requirement and performance estimation. This part of the dissertation discusses how those techniques can be combined with component placement strategies to come up with efficient resource allocation. It also discusses heuristics for bin-packing for multiple dimensions. It then discusses how all of the above techniques can be applied to a solve resource allocation problem in a modern data center.

Chapter VIII carries out a detailed study of different bin packing heuristics such as first-fit, best fit etc. for multiple dimensions and finds interesting results where it found that each one is better than the other under different circumstances. Chapter IX presents a component placement framework which uses a performance model and profiling data to place components onto nodes. It then goes onto to develop a component replication algorithm where the findings in chapter VIII are used. Finally in chapter X, the framework is applied onto modern day data center resource planning where a cloud computing infrastructure is used to increase and decrease the resources for the application.

CHAPTER VIII

MULTI-CAPACITY RESOURCE ALLOCATION IN DISTRIBUTED COMPONENT BASED SYSTEMS

This chapter deals with resource allocation in distributed component based systems. In the first section the resource allocation problem is posed as a bin packing problem. Bin-packing is a NP hard problem. Many heuristics have been proposed in the research literature to solve the problem. This chapter carries out a detailed study of various bin-packing heuristics and identifies the different conditions under which each heuristic works best.

VIII.1 Resource Allocation As A Bin Packing Problem

Applications often require multiple resources to execute properly, and need timely allocation of those resources to maintain required QoS. System resource utilization is a function of input workload and the required QoS of applications, so runtime utilization may vary significantly from estimated values. Moreover, system resource availability, such as available network bandwidth and battery power, may also be time variant. Numerous algorithms have been developed, studied, and analyzed for use in resource allocation. For example, Srivastav and Stangier [69] provide a solution to the resource-constrained scheduling problem, which is related to the multi-dimensional bin-packing problem.

In particular, bin-packing algorithms provide a natural solution to many resource allocation problems. The classical bin-packing problem packs a set of n items into m bins each with a maximum capacity C, such that the sum of the items in any bin does not exceed C. In the context of resource allocation, resources (*e.g.*, processors) form the bins, and items map to tasks (*e.g.*, components) that require a specified amount of resources.

This chapter presents an empirical study of widely used bin-packing algorithms, focusing on the applicability of these algorithms in the context of resource allocation in DRE systems. For each algorithm, the following is studied and analyzed: (1) how effective the algorithm is in finding a feasible allocation under stringent time limitations, depending on the input application characteristics, and (2) how useful additional computation to find an allocation is for different application characteristics.

Empirical comparison of heuristic performance. As mentioned above this chapter empirically evaluates bin-packing algorithms that is used to make the initial and subsequent resource allocations. Since complete bin-packing algorithms can be computationally expensive, different heuristic schemes are studied in a multi-capacity bin-packing framework to simplify the allocation task. The goal is to determine resource allocation heuristic *performance patterns*, *i.e.*, the likelihood of a heuristic finding an allocation for different classes of input. These performance patterns can be used to (1) select appropriate resource allocation algorithms based on the input data set at runtime and (2) determine how much computation to expend on each.

To determine the performance of multi-capacity extensions to common bin-packing heuristics, a series of experiments with problems drawn from various input distributions are run. These experiments used two-capacity bins, applicable to the case of system nodes with two resource attributes, such as CPU and memory. The performance metrics considered are "number of successes" in a fixed number of runs. The extension to additional resources for these heuristics is straightforward from the two-capacity implementation. The size of each of the two bin capacities are set to 100, representing 100% of the resource. In analyzing the results, three orthogonal dimensions to the cases being tested are considered, as described below.

• **Heuristic** is the performance of each algorithm/heuristic on the generated problems. The extensions to multi-capacity bin-packing of the popular best-fit, first-fit, and worst-fit heuristics are evaluated.

• **Sorting method** is the method used to sort the items before applying the above heuristics. Sorting items by decreasing size before packing the bins often provides better performance in traditional bin-packing. In the experiments, items are also sorted in decreasing order of size, but because the problems are multi-capacity ones, the sorting criteria is non-trivial. Several definitions for a scalar size value (combining or comparing the multiple dimensions) could be used, including sum, product, sum of squares, and maximum component. The current experiments focus on two scalar definitions of size: sum and maximum component.

• **Item distribution** is the characterization of the distribution from which item sizes are drawn. Different solution methods and heuristics may be more/less applicable to particular item size distributions. One goal of the experiments is to determine if/when these heuristics are more effective based on characterization of input item sizes. For these experiments uniform distributions were used with various mean values. The total amount of slack between the capacities of the bins and the sizes of the items are compared. For example, a problem with 10 bins of capacity (100,100) and a slack of exactly 10 percent in each dimension, would have a set of items whose sizes sum to (900,900).

VIII.1.0.1 Problem Generation

Three input parameters characterize the problems generated for a given set of test runs: (1) number of (100,100) capacity bins, (2) range of item sizes, and (3) percentage slack allowed (as a range) in the generated problems. These experiments use two-capacity bins/-items, with the item's size in each dimension independently drawn from uniform distributions. For example, with 10 bins, 0-70 for item sizes, and 5-10 as the allowable percentage of slack, the set of problems generated would have items with an average size of $\sim$35 in each resource and total size for the sum of items would be between 900 and 950 in each resource.

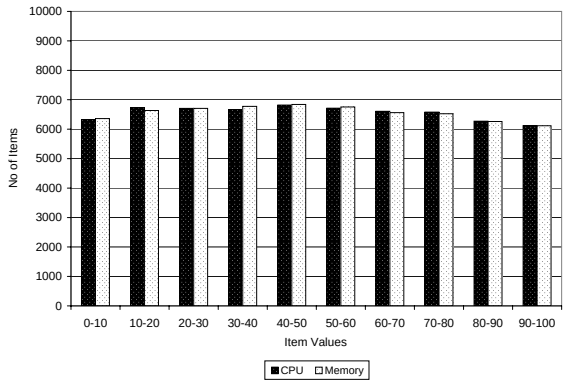


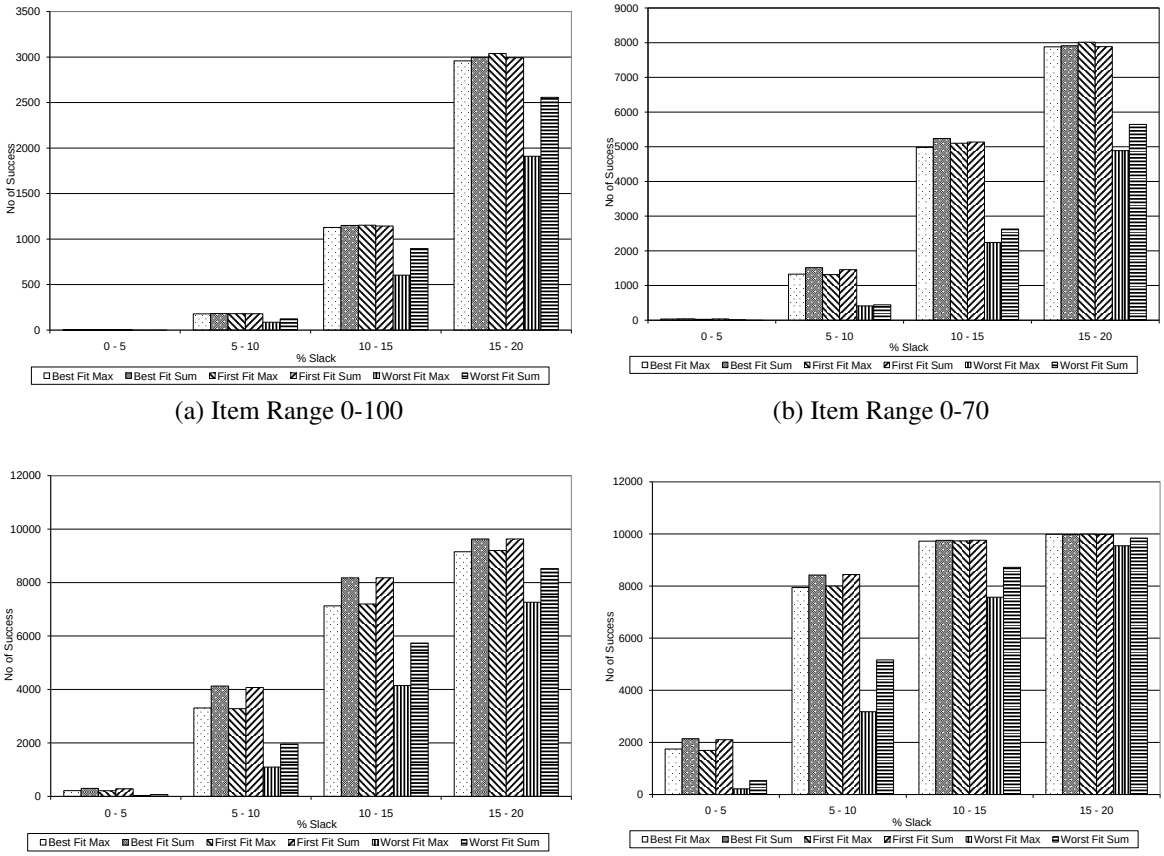
Figure 54: Distribution of Items (0-100 range)

To generate the problems, *rejection-sampling* [59] is used, which samples from an arbitrary distribution f(x) using some standard distribution g(x) that is easy to sample. Items are generated with the constraint that the sum of the item sizes should be less than the sum of all bin capacities by an amount within the range of allowable slack. Items are generated from the specified distribution until their sum was within the allowable range of slack or was greater than the maximum value, in which case that set of items was rejected. The generated set of items thus meets global bin capacity constraints, but a valid allocation is not guaranteed, *i.e.*, the problem may or may not be solvable.

10,000 instances of the problems are generated that were run with the different heuristics. The number of bins used was 4 because using more bins made the running time of the complete algorithm too large to generate results in a reasonable amount of time. The running time was primarily a problem for distributions with smaller average item size because there were more items in total. For the other distributions, the experiments were performed with 10 bins and obtained results that followed the same patterns identified in the 4 bin experiments presented here.

Figure 54 shows a representative frequency distribution of the item sizes in one of the two dimensions for a problem with 0-100 item distribution and slack between 0% and 5%.

The problem sets for the other distributions and slack values also closely match their specified uniform distributions and are not included.



(a) Item Range 0-100

(b) Item Range 0-70

(c) Item Range 0-50

(d) Item Range 0-30

Figure 55: Performance Comparison of Different Heuristics

VIII.1.0.2 Analysis of Results

Figure 55a shows the relative performance of the different heuristics with a uniform distribution of item sizes between 0 and 100 for each dimension. This result shows that the best-fit and first-fit heuristics outperform the worst-fit ones overall (for this distribution). Moreover, the choice of sorting criteria does not make a large difference, as shown by the similar results for both sum and maximum component sorting when used with each heuristic.

The results for the items with ranges 0-70 and 0-30 show the same patterns, and are presented in the Figures 55b and 55d. The results from the 0-50 distribution exhibit a slightly

different behavior as shown in Figure 55c. For these problems, there is a significant difference in performance between sorting criteria, with the sum sorting criteria outperforming the maximum component sorting criteria.

The results for these experiments clearly show a performance pattern for each heuristic across different input distributions, which RACE exploits during runtime resource allocation. For example, if the input distribution is roughly uniform with a mean item size of 25, these results would direct RACE to first employ the best-fit heuristic with sorting based on sum of the items, before expending computation on the other heuristics. Conversely, for uniform distributions with a mean item size near 50, RACE would prefer to first try the first-fit heuristic with sorting based on maximum component size.

VIII.1.1 Point of Diminishing Returns for Running Resource Allocation Algorithms

Problem. The set of multi-capacity bin-packing heuristics tested in these experiments make a single attempt at finding an allocation. Extending these heuristics (*e.g.*, with backtracking or local search), however, is likely to produce better results in many cases at the cost of additional computation. Still, this is not consistently the case across all input distributions tested. RACE must therefore determine whether additional computation would significantly enhance its chance of finding a solution.

Item Size	% Slack			
	0 - 5	5 - 10	10 - 15	15 - 20
0 - 30	34.84	97.96	99.97	100
0 - 50	10.57	65.49	96.14	99.67
0 - 70	26.44	65.68	93.02	99.14
0 - 100	100	94.93	99.34	99.64

Table 17: Success Rate of Heuristics on Solvable Problems

Solution → Empirical study of heuristics performance on solvable problems. Determining where additional computation can and cannot yield better performance requires further analysis of our experimental results. By identifying which input distributions yielded relatively low rates of success relative to total solvable problems, the cases in which additional computation would likely benefit heuristic performance are found. Table 24 summarizes the combined performance of the heuristics on the *solvable* problems from each distribution. To determine whether a solution existed for each problem, a complete algorithm was implemented that searched all combinations for packing the items. Due to the running time of this algorithm, the number of bins in the experiment are limited to 4.

The success rate of the heuristics (computed for the subset of problems for which there existed a valid allocation) varies with the distribution ranges as well as with the slack in the bins. It is clear from the results that the heuristics perform quite well when the slack is relatively large (*i.e.*, greater than 10%). When there is very little slack (*i.e.*, 0-5% slack), however, the heuristics mostly perform poorly, with performance increasing with greater slack.

The exception to this trend is the 0-100 range items where there is a preponderance of medium-to-large size items (relative to bin capacity). In that case, the heuristics perform extremely well, with the unexpected result that they perform best with very little slack. These results suggest that when there are a significant number of large items and very little slack, the problem may only be solvable in one way (or a small number of ways) that is immediately found by the heuristics as they attempt to pack the bins without exceeding their capacity. The success rate diminishes a little with greater slack but is still quite high (99%). This may mean that there are only a few ways to allocate the large items to bins, but a number of different ways to attempt to pack the remaining smaller items. Due to these additional possibilities, the heuristics may be less likely to find a valid allocation with their single attempt. Moreover, the hardest problems are those where the item sizes range from

small to medium or medium-large (*e.g.*, 0-50 and 0-70 in these experiments). The 0-30 range is easier because there are many small items, allowing many valid allocations.

The analysis above can be employed to determine when to terminate a particular algorithm. For example, when the components tend toward medium-to-large resource requirements, and the heuristics fail to find an allocation, it can be assumed that the components most likely cannot be allocated, so there is no point in expending additional computation searching for a solution. Conversely, if the component resource requirements are all between 0% and 50% with little slack (0-10%), then the heuristics can be run with backtracking for a longer duration before terminating the execution.

VIII.2 Concluding Remarks

This work presented in this chapter provides an empirical evaluation of several multi-capacity bin-packing heuristics for resource allocation to identify performance patterns associated with these heuristics. These patterns provide a basis for any adaptive resource management framework to select an appropriate suite of resource allocation methods based on the resource requirement characteristics of application components. This selection can be done at design time or runtime. The lessons learned from the work can be summarized as follows:

Use a suite of heuristics. Analysis of the heuristics presented in Section IX.4 shows that the performance of a given heuristic depends on (1) the sorting method used to order the items and (2) the distribution of the item sizes and slack (difference between the bin capacities and the sum of item sizes). Moreover, no heuristic consistently out-performs all others. To increase the likelihood of successful runtime resource allocation, an adaptive resource management framework, should employ a suite of algorithms/heuristics that execute in parallel.

Spend time wisely in searching for an allocation. In addition to using each of these heuristics as a single-shot attempt to find an allocation, further computation may be fruitful

in certain cases where the heuristics do not immediately find a solution. For example, the results suggest there is little benefit to using additional computation when the input contains a preponderance of medium and large components relative to node capacity (*e.g.*, with the 0-100 distribution the heuristics found a solution almost every time one existed).

Similarly, when there is a great deal of slack between component resource requirements and total system resources, the heuristics were likely to find a solution, if one existed, and further computation would not improve performance. Moreover, when the heuristics are extended to perform multiple attempts at finding an allocation (*e.g.*, through backtracking or local search), our results suggest that the most efficient solution will be to provide some of the heuristics more computational resources/time than others.

Classify input to dynamically create weighted heuristic suite. Based on our experiment results, it appears that an effective and efficient process for allocating system resources to application components involves (1) inspecting and analyzing component resource requirements to classify the input item distribution and (2) weighted selection of a suite of allocation algorithms/heuristics that are most likely to find a valid allocation of system resources. While the relative weight given to the heuristics could be set based on system-wide availability of components (*e.g.*, at design time), a more flexible solution is to dynamically adjust the weights at runtime as applications are provided for allocation. The input application can be characteried based on component resource requirements and adjust the weights dynamically to efficiently find a valid allocation.

In future work, the performance of multi-capacity bin-packing heuristics will be tested on a wider range of input distributions, including normal distributions with a variety of means and variances. Input patterns for which particular heuristics are likely/unlikely to succeed with additional backtracking or local search computation are to be classified. This classification will help support a wider range of systems and applications by improving the efficiency and effectiveness of dynamic resource allocation.

The implementations of the resource allocation heuristics evaluated in this chapter are

available as open-source software from deuce.doc.wustl.edu/Download.html and www.dre.vanderbilt.edu/labjar/Allocation, respectively.

CHAPTER IX

COMPONENT ASSIGNMENT FRAMEWORK FOR QOS ASSURANCE

This chapter describes the development of a framework for assigning components onto the nodes in a way that will maximize the utility of the application. It will try to handle the maximum number of clients while keeping the amount of resources used at a minimum. This chapter presents the *Component Assignment Framework for distributed component based pplications* (CAFe), which is an algorithmic framework for increasing capacity of a web portal for a fixed set of hardware resources by leveraging the component-aware design of contemporary web portals. The goal of CAFe is to create a deployment plan that maximizes the capacity of the web portal so their performance remains within SLA bounds. It consists of a mechanism to predict the application performance coupled with an algorithm to assign the components onto the nodes.

IX.1 Problem Formulation and Requirements

As discussed above , the problem CAFe addresses involves maximizing the capacity (user requests or user sessions) of a web portal for given hardware resources, while ensuring that the application performance remains within SLA bounds. This problem can be stated formally as follows: The problem domain consists of the set of n components C, $\{C_1, C_2, ...C_n\}$, the set of m nodes P $\{P_1, P_2...P_m\}$, and the set of k services $\{S_1, S_2...S_k\}$. Each component has Service Demand D $\{D_1, D_2...D_n\}$. Each service has response time RT $\{RT_1, RT_2...RT_k\}$. The capacity of the application is denoted by either the arrival rate, λ for each service $\{\lambda_1.....\lambda_k\}$ or the concurrent number of customers M $\{M_1, M_2, ...M_k\}$. $SU_{i,r}$ gives the utilization of resource r by component i. The SLA gives an upper bound on the response times of each service $k\{RT_{sla,1}...RT_{sla,k}\}$.

CAFe must therefore provide a solution that places the n components in C to the m

nodes P such that the capacity (either λ or M) is maximized while the response time is within the SLA limit $RT < RT_sla$. To achieve this solution, CAFe must meet the following requirements:

Place components onto nodes to balance resource consumption in polynomial time. Application components must be placed onto the available hardware nodes such that application capacity is maximized while the performance is within the upper bound set by a SLA. Since this is an NP-Hard problem [74] it is important to find out efficient heuristics that can find good approximate solutions. Section IX.2.1 describes how CAFe uses an efficient heuristic to place the components onto the available nodes and also ensuring that the resource utilization is balanced.

Estimate component requirement and application performance for various placement and workload. To place the components in each node, the resource requirement of each component is required. Moreover, for each placement strategy, the application performance must be compared with the SLA bound. Both vary with workload and particular placement. Therefore there is the need of a workload- and component-aware way of component resource requirement and application performance estimation. Section VI.3.2 describes how CAFe develops an analytical model to estimate component resource requirement and application performance.

Co-ordinate placement routine and performance modeling to maximize capacity. For each particular placement, the application performance need to be estimated to check if it is within the SLA limit. Conversely, performance estimation can only be done when a particular placement is given. Placement and performance estimation must therefore work closely to solve the overall problem. Section IX.2.2 describes how CAFe designs a algorithmic framework to co-ordinate the actions of a placement routine and an analytical model in a seamless fashion.

IX.2 CAFe: A Component Assignment Framework for Multi-Tiered Web Portals

This section discusses the design of CAFe and how it addresses the problem and requirements presented in Section IX.1. CAFe consists of two components: the placement algorithm and analytical model, as shown in Figure 56. The input to CAFe includes the

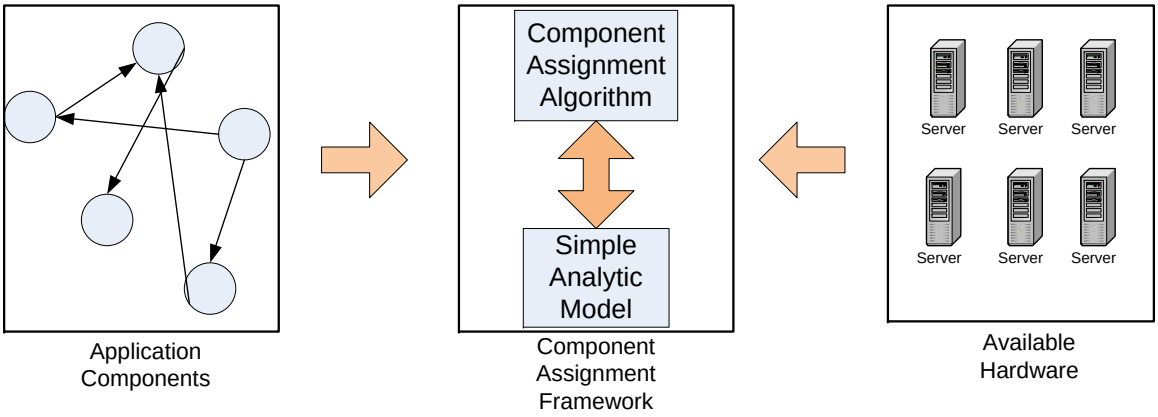


Figure 56: The CAFe Component Assignment Framework Architecture

set of application components and their inter-dependancy, as shown by the box on the left in Figure 56 and the set of available hardware nodes shown on the right. The output from CAFe is a deployment plan containing the mapping of application components to nodes. This mapping will attempt to maximize the application capacity. The rest of this section describes each element in CAFe.

IX.2.1 Allocation Routine

As mentioned in Section IX.1, there is a need to develop efficient heuristics to place components onto nodes. Placing components to hardware nodes can be mapped as a bin-packing problem [15], which is NP-Hard in the general case (and which is what our scenarios present). Existing bin-packing heuristics (such as first-fit and best-fit) can be used to find a placement that is near optimal.

Chapter I describes the intuition behind the allocation routine, which is that performance increases by balancing the resource utilization between the various nodes and not allowing the load of any resource to reach 100%. Worst-fit bin packing is used since it allocates items to bins by balancing the overall usage of each bin.

Algorithm 2 gives the overall allocation routine, which is a wrapper around the worst-fit

Algorithm 2: Allocate

C: Set of components
foreach *L:Set of Components that should remain local* **do**
D← Sum the Service Demands of all components in L
Replace all components in L with D in C
end
$DP = worst\_fit\_bin\_packing(C,P)$

Algorithm 3. Algorithm 2 groups together components that are constrained to remain co-located in one machine. For example, they could be the database components that update the tables and need to be connected to the master instance of a database and hence msut be allocated to a single node. Algorithm 2 sums the Service Demands of the components that must be collocated and then replaces them by a hypothetical single component. A call to the $worst\_fit$ routine is made at the end of algorithm 2.

The worst-fit routine is given in Algorithm 3. The components are first ordered accord-

Algorithm 3: Worst-Fit Bin Packing

begin
$Order\_Components(C)$ // Order the Components by resource requirement
;
foreach $C_i \in C, 1 \leq i \leq |C|$ **do**
// For all components
$P_k \leftarrow$ The node with the maximum slack
Place C_i on to P_k
end
end

ing to their resource requirements (Line 3). The algorithm then runs in an iterative fashion. At each step an item is allocated to a bin. All bins are inspected and the least utilized bin is selected.

IX.2.2 Algorithmic Framework to Co-ordinate Placement and Performance Estimation

Section IX.1 also discusses the need for close co-ordination between placing the components and performance estimation. To meet this requirement CAFe provides a framework that standardizes the overall algorithm and defines a standard interface for communication between the placement and performance estimation. This framework also allows the configuration of other placement algorithms, such as integer programming and different analytical models like models based on worst case estimation. Different algorithms or analytical models can be configured in/out to produce results pertaining to the specific application domain or scenario.

CAFe uses an analytical model of the application and a placement routine to determine a mapping of the application components onto the available hardware nodes. CAFe attempts to maximize the capacity of the web portal, while ensuring that the response time of the requests is within the SLA bounds.

Algorithm 4 describes the component assignment framework.

The algorithm alternates between invoking the placement algorithm and an evaluation using the analytic model. In each step, it increases the number of clients in the system by a fixed amount and rearranges the components across the nodes to minimize the difference in resource utilization. It then verifies that the response time is below the SLA limit using the analytical model and iterates on this strategy until the response time exceeds the SLA limit.

CAFe takes as input the details (such as the Service Demands of each component on each resource) of the N components to deploy. Service Demand is the amount of resource

Algorithm 4: Component Assignment Framework

Input:
$C \leftarrow$set of N components to be deployed,
$D \leftarrow$set of Service Demands for all components, $D_{i,r} \leftarrow$Service Demand of component i on the device r
$P \leftarrow$set of K available nodes
RT_{sla} set of response time values for each service as specified by the SLA
Output:
Deployment plan $DP \leftarrow$ set of tuples mapping a component to a node,
M: Total Number of concurrent clients
$RT \leftarrow$set of response times for all components
RT_i: Total response time of service i
U_r: Total Utilization of each resource r in the nodes
$SU_{i,r}$: Utilization of resource r by component i
$SU \leftarrow$ set of resource utilization of all components
$Incr$: Incremental capacity at each step
$Init_Cap$:Initial Capacity
begin
Intially, $DP = \{\}, M = Init_Cap, SU = U, RT_i = \sum_r D_{i,r}, incr = Incr$;
while $incr > 10$ **do**
while $\nexists i : RT_i > RT_{sla,i}$ **do**
// Check if any service RT is greater than SLA bound
;
$DP = Allocate(SU, P)$ // Call Placement routine to get a placement
$(RT, SU, U) = Model(M, D, DP)$// Call model to estimate performance for current placement
$last\_M \leftarrow M$ // Save the previous capacity
$M \leftarrow M + incr$ // Increment the capacity for the next iteration
end
// At least one service has Response Time greater than SLA bound for current capacity
$M \leftarrow last\_M$ // Rollback to previous iteration's capacity
$incr \leftarrow incr/2$ // Decrease incr by half
$M \leftarrow M + incr$ // Now Increase capacity and repeat
end
// while($incr > 10$)
end

time taken by one transaction without including the queuing delay. For example, a single Login request takes 0.004 seconds of processor time in the database server. The Service Demand for Login on the database CPU then takes 0.004 seconds. As output, the framework provides a deployment plan(DP, estimated response (RT) time and total utilization (U) of each resource.

The initial capacity is an input ($Init\_Cap$, Line 4). The value of M is set equal to $Init\_Cap$. This capacity is an arrival rate for an open model or "number of users" for a closed model. The capacity (M) is increased in each step by an incremental step *incr* (Line 4) which also can be parameterized ($Incr$). At each iteration, the response time of all services is compared with the SLA provided upper bound (inner while loop at Line 4).

Inside the inner loop, the framework makes a call to the Allocate module (Line 4), which maps the components to the nodes. This mapping is then presented to the Model

(Line 4) along with the Service Demand of each component. The Model computes the estimated response time of each service and the utilization of each resource by each component. It also outputs the total utilization of each resource.

The inner loop of Algorithm 4 exits when response time of any service exceeds the SLA provided upper bound (*i.e.*, M reaches maximum capacity), at which point *incr* is set to a lower value (one-half) and the algorithm continues from the previous value of M (Line 4). If the inner loop exits again, the value of *incr* is lowered further (Line 4). The algorithm ends when the value of *incr* is less than 10.

The output of the algorithm is that value of M, which yields the highest capacity possible and also a deployment plan (DP) that maps the application components onto the nodes. Though not provably optimal, the algorithm is a reasonable approximation. An optimal algorithm would require an integer programming routine [66] to obtain the mapping of the components to the nodes. Such an implementation would be NP-Hard, however, and thus not be feasible for large applications. CAFe therefore uses an intuitive heuristic based on the popular worst-fit bin packing algorithm [15].

IX.3 Experimental Evaluation

IX.3.1 Rice University Bidding System

This section describes our experimental evaluation of CAFe, which used the Java servlets version of the Rice University Bidding System (RUBiS) [3] to evaluate its effectiveness. RUBiS is a prototype of an auction site modeled after ebay that has the features of an online web portal studied in this chapter. It provides three types of user sessions (visitor, buyer, and seller) and a client-browser emulator that emulates users behavior.

A RUBiS session is a sequence of interactions for the same customer. For each customer session, the client emulator opens a persistent HTTP connection to the Web server and closes it at the end of the session. Each emulated client waits for a certain think time before initiating the next interaction. The next interaction is determined by a state transition

matrix that specifies the probability to go from one interaction to another one. The load on the site is varied by altering the number of clients.

CAFe requires an analytical model of the application. Once the model is constructed and validated, it can be used in CAFe to find the appropriate component placement. The steps required to build the model are (1) compute Service Demand for each service provided for each customer type such as visitor or buyer and (2) build a customer behavior modeling graph of user interactions and calculate the percentage of requests for each service. For our experiments, a workload representing a set of visitor clients were chosen, so the workload consists of browsing by the users and is thus composed of read-only interactions. The components for each service in RUBiS is given in Table 18.

Service Name	Home Page	Browse Page	Browse_Cat	Browse_Reg	Br_Cat_Reg
Business Tier	BT_H	BT_B	BT_BC	BT_BR	BT_BCR
DB Tier	_	_	DB_BC	DB_BR	DB_BCR

Service Name	Srch_It_Cat	Srch_It_Reg	View_Items	Vu_Usr_Info	Vu_Bid_Hst
Business Tier	BT_SC	BT_SR	BT_VI	BT_VU	BT_BH
DB Tier	DB_SC	DB_SR	DB_VI	DB_VU	DB_BH

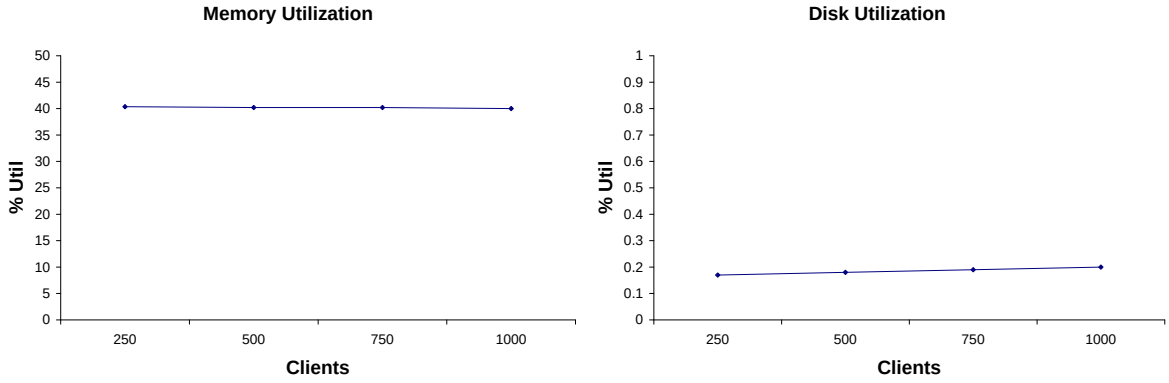
Table 18: Component Names for Each Service

The RUBiS benchmark was installed and run on the ISISLab testbed (`www.isislab.vanderbilt.edu`) at Vanderbilt University using 3 nodes. One for the client emulators, one for the "Business Tier" and the other for "Database Tier". Each node has 2.8 GHz Intel Xeon processor, 1GB of ram, and 40GB HDD running Fedora Core 8.

IX.3.2 Computing Service Demand

The Service Demand of each of the components must be captured to build an analytical model of the application. The RUBiS benchmark was run with increasing clients and its

effect on various CPU, memory, and disk were noted. The memory and disk usages are shown in Figures 57a and 57b.



(a) Memory Utilization (b) Disk Utilization

Figure 57: The Utilization of Memory and Disk for RUBiS Benchmark

Disk usage is low ($\sim$0.2%) and memory usage was $\sim$40%. Moreover, these utilizations remained steady even as the number of clients are increased. Conversely, CPU usages increased as number of clients grew (the Actual line in the Figure 16a).

The results in Figures 57a and 57b show that CPU is the bottleneck device. The Service Demands were computed for the CPU and the disk. Since memory was not used fully, it is not a contentious resource and will not be used in the analytical model. Moreover, the CAFe placement routine ignores disk usage since it remains steady and is much less than CPU usage. The CAFe placement routine thus only uses one resource (CPU) to come up with the placement.

RUBiS simplifies the calculation of Service Demand. It includes a client-browser emulator for a single client and makes requests on one service at a time. During the experiment, the processor, disk and memory usages were captured. After the experiment finished the Service Demand law [46] is used to calculate the Service Demand for that service. In some services (such as “Search Items in Categories”) the Service Demand is load dependent. For

such services the number of clients was increased and the Service Demands were measured appropriately.

The Service Demands of CPU for all the services measured in such a way are given in Table 19. Each service in RUBiS is composed of multiple components, with a component

Service	Business Tier Component(secs)	DB Server Component(secs)	Description
home	0.002	0	Home Page
browse	0.002	0.0	Browse Main Page
browse_cat	0.0025	0.0005	Browse Categories
browse_reg	0.0025	0.0005	Browse Regions
br_cat_reg	0.003	0.0007	Browse Categories in Regions
Srch_it_cat	0.004	0.028	Search Items in Categories
Srch_it_reg	0.0021	0.027	Search Items in Regions
view_items	0.004	0.0009	View Items
vu_usr_info	0.003	0.001	View User Info
vu_bid_hst	0.004	0.004	View Bid History

Table 19: CPU Service Demand for Each Component

in the middle (Business) tier and one in the Database Tier. Each component has its own resource requirements or Service Demands.

IX.3.3 Customer Behavior Modeling Graph

For the initial experiment, the workload was composed of visitor type of clients. A typical user is expected to browse across the set of services and visit different sections of the auction site. A transition probability is assumed for a typical user to move from one service to the other.

The various transition probabilities are given in Table 20.

Here element $p_{i,j}$ (at row i and column j) represents the probability of the i^{th} service being invoked after the j^{th} service is invoked. For example, a user in the web page “browse_cat”(browsing categories) has a 0.0025% chance of going to the “home” page and a 99% chance for moving on to "Search_it_cat"(searching for an item in a category).

	home	browse	browse_cat	browse_reg	br_cat_reg	Srch_it_cat	Srch_it_reg	view_items	vu_usr_info	vu_bid_hst	view_items_reg	vu_usr_info_reg	vu_bid_hst_reg	Probabilities
home	0	0.01	0.0025	0.0025	0.0025	0.0025	0.0025	0.0025	0.0025	0.0025	0.0025	0.0025	0.0025	0.0026
browse	1	0	0.0075	0.0075	0.0075	0.0075	0.0075	0.0075	0.0075	0.0075	0.0075	0.0075	0.0075	0.0100
browse_cat	0	0.7	0	0	0	0	0	0	0	0	0	0	0	0.0070
browse_reg	0	0.29	0	0	0	0	0	0	0	0	0	0	0	0.0029
br_cat_reg	0	0	0	0.99	0	0	0	0	0	0	0	0	0	0.0029
Srch_it_cat	0	0	0.99	0	0	0.44	0	0.74	0	0	0	0	0	0.3343
Srch_it_reg	0	0	0	0	0.99	0	0.44	0	0	0	0.74	0	0	0.1371
view_items	0	0	0	0	0	0.55	0	0	0.8	0	0	0	0	0.2436
vu_usr_info	0	0	0	0	0	0	0	0.15	0	0.99	0	0	0	0.0747
vu_bid_hst	0	0	0	0	0	0	0	0.1	0.19	0	0	0	0	0.0386
view_items_reg	0	0	0	0	0	0	0.55	0	0	0	0	0.8	0	0.0999
vu_usr_info_reg	0	0	0	0	0	0	0	0	0	0	0.15	0	0.99	0.0306
vu_bid_hst_reg	0	0	0	0	0	0	0	0	0	0	0.1	0.19	0	0.0158

Table 20: Transition Probabilities Between Various Services

The steady state probability (percentage of user sessions) for each service type is denoted by the vector π. The value of π_i denotes the percentage of user requests that invoke the the i^{th} service. The vector π can be obtained by using a technique is similar to the one in [83]. Once computed, the amount of load on each service type can be calculated from the total number of user sessions. The rightmost column in Table 20 gives the steady state probabilities of each service.

IX.3.4 Analytical Modeling of RUBiS Servlets

The analytical model of the RUBiS application is developed as described in detail in V.0.1. It is then used in the CAFe Algorithm as described below.

IX.3.5 Application Component Placement

It is now described how CAFe iteratively places components onto hardware nodes. The SLA is assumed to have set an upper bound of 1 sec on the response time of all services. Algorithm 4 is used from Section IX.2, which considers CPU as the only resource since both memory and disk usage is minor compared to CPU usage, as described in Section IX.3.

The first iteration of Algorithm 4 uses the initial Service Demands for each application component. The Service Demands are given in Table 19. The set of all Service Demands and the available nodes (in this case 2) are used by the *Allocate* Algorithm 2. This algorithm in turn invokes the *worst_fit_bin_packing* described in Algorithm 3, which places components on the two nodes.

As mentioned in Section IX.3, there are two nodes used for RUBiS benchmark: Business Tier Server (BT_SRV) and the Database Server (DB_SRV). The name of the nodes are given since the default deployment of RUBiS uses the BT_SRV to deploy all the business layer components and DB_SRV to deploy the database. In fact, such a tiered deployment is an industry standard [22].

Table 21a shows the placement of the components after the first iteration of Algorithm 4 in CAFe. The mapping of the components to the nodes, the total number of clients (100),

BT_SRV		DB_SRV	
Component	CPU Util	Component	CPU Util
DB_SC	0.02783	DB_SR	0.02690
BT_VI	0.00405	BT_SC	0.00417
DB_BH	0.00400	BT_BH	0.00400
BT_UI	0.00300	BT_BCR	0.00325
BT_BC	0.00245	BT_BR	0.00253
BT_H	0.00200	BT_SR	0.00210
DB_UI	0.00100	BT_B	0.00200
DB_VI	0.00095	DB_BCR	0.00075
DB_BC	0.00055	DB_BR	0.00047

(a) Component Placement

Service	Business Tier Component%	DB Server Component%	Response Time
home	0.007	0.000	0.002
browse	0.029	0.000	0.002
browse_cat	0.025	0.005	0.004
browse_reg	0.010	0.002	0.004
br_cat_reg	0.014	0.003	0.005
Srch_it_cat	1.980	13.190	0.049
Srch_it_reg	0.380	5.260	0.041
view_items	1.940	0.490	0.006
vu_usr_info	0.480	0.120	0.005
vu_bid_hst	0.310	0.310	0.009

(b) Utilization and Response Time

Table 21: Component Placement and RUBiS Performance After Iteration 1

and the Service Demands of the components are used to build the analytical model. It is then used to find the response time and processor utilization of the two servers, given in Table 21b. The response time of all the services is well below the SLA specified 1 sec. CAFe iterates and the processor utilization of each component found in the previous iteration is used in the *Allocate* routine.

In the second iteration, the *Allocate* Algorithm 2 produces the placement shown in Table 22.

In the third iteration, the number of clients, M is increase to 300. The placement computed by CAFe remains the same, however, and the response times of the two services

BT_SRV		DB_SRV			
Component	CPU Util	Component	CPU Util	Component	CPU Util
DB_SC	13.19	DB_SR	5.26	BT_B	0.029
		BT_SC	1.97	BT_BC	0.025
		BT_VI	1.94	BT_BCR	0.014
		DB_VI	0.49	BT_BR	0.010
		BT_UI	0.48	BT_H	0.007
		BT_SR	0.39	DB_BC	0.005
		BT_BH	0.31	DB_BCR	0.003
		DB_BH	0.31	DB_BR	0.002
		DB_UI	0.12		

Table 22: Iteration 2:Component Placement by Allocation Routine

"Search By Category" and "Search by Region" increase with each iteration as shown in Table 23.

Iteration	Clients	home	broBTe	broBTe_cat	broBTe_reg	br_cat_reg	Srch_it_cat	Srch_it_reg	view_items	vu_usr_info	vu_bid_hst
1	100	0.002	0.002	0.004	0.004	0.005	0.049	0.041	0.006	0.005	0.009
2	200	0.002	0.002	0.004	0.004	0.005	0.050	0.041	0.006	0.005	0.009
5	500	0.002	0.002	0.004	0.004	0.005	0.058	0.044	0.007	0.005	0.010
10	1000	0.002	0.002	0.005	0.005	0.006	0.088	0.049	0.007	0.006	0.011
15	1500	0.002	0.002	0.005	0.005	0.007	0.689	0.055	0.008	0.007	0.012
16	1600	0.003	0.003	0.006	0.006	0.007	1.119	0.057	0.008	0.007	0.012
17	1550	0.003	0.003	0.006	0.006	0.007	0.899	0.056	0.008	0.007	0.012
18	1575	0.003	0.003	0.006	0.006	0.007	1.011	0.057	0.008	0.007	0.012
19	1563	0.003	0.003	0.006	0.006	0.007	0.956	0.056	0.008	0.007	0.012

Table 23: Successive Iterations:Response Time of Each Service

At the value of $M = 1600$, the response time of the service "Search by Category" crosses the SLA limit of 1 sec as shown in iteration 16 in Table 23. At that point *incr* variable in Algorithm 4 is reduced by half to 50 and M is reduced to the previous value of 1500. The algorithm continues from that point. Thus in iteration 17, value of M is 1550 In a similar way, for M equal to 1563, the response time of "Search by Category" is just below 1 sec (iteration 19). This response time is the maximum capacity of the application under a SLA response time of 1 sec. Figure 58 shows the comparison in the response time of the service "Search By Category," which is the bottleneck service.

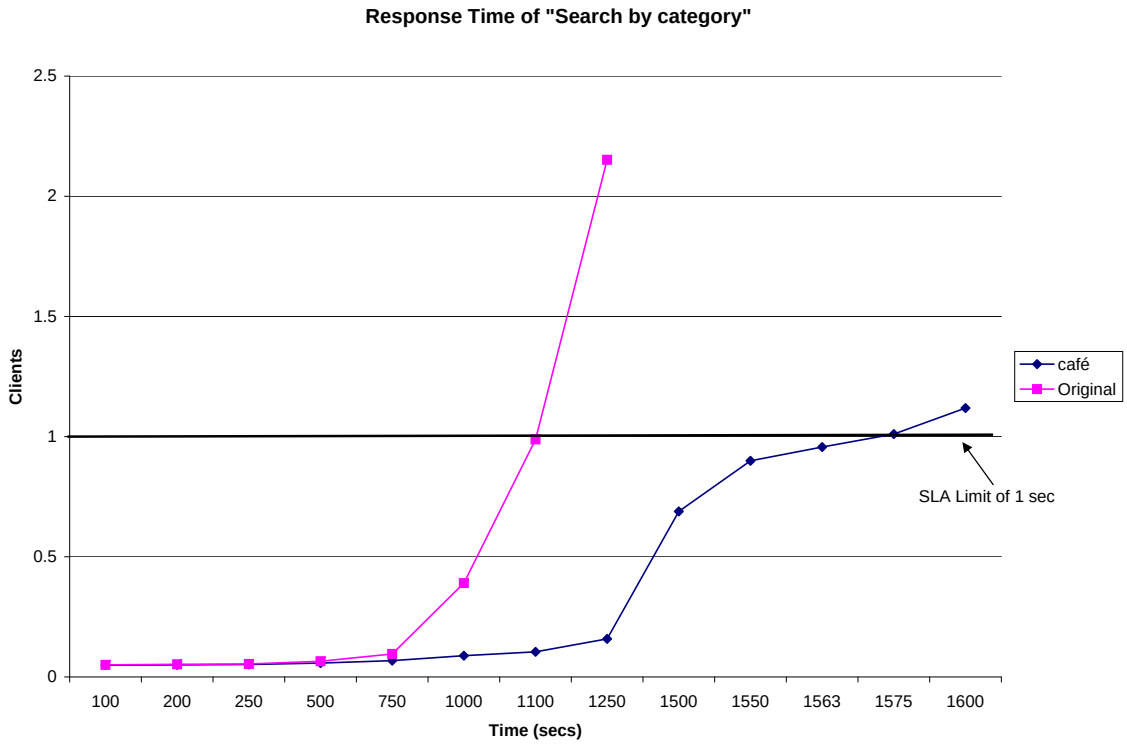


Figure 58: Response Time with Increasing Clients

IX.3.6 Implementation of the CAFe Deployment Plan

It is now described how RUBiS uses the new deployment plan recommended by CAFe and empirically evaluate the performance improvement compared with the default tiered architecture used by RUBiS. This plan assigns all the Business Tier components in the BT_SRV and the entire database in the DB_SRV. The deployment suggested by CAFe is shown in Table 22, where component DB_SC is contained in one node and all the others are kept in the other node. The component DB_SC is the database component of the service "Search By Category," which is a read-only component that invokes a select query on the database.

One way to implement this assignment is to run a master instance of the database along with all the other components and run a slave instance of the database in the machine where DB_SC is run. The corresponding deployment is shown in Figure 59. In this figure there are two instances of the Database: the master instance is run in the machine BT_SRV and a

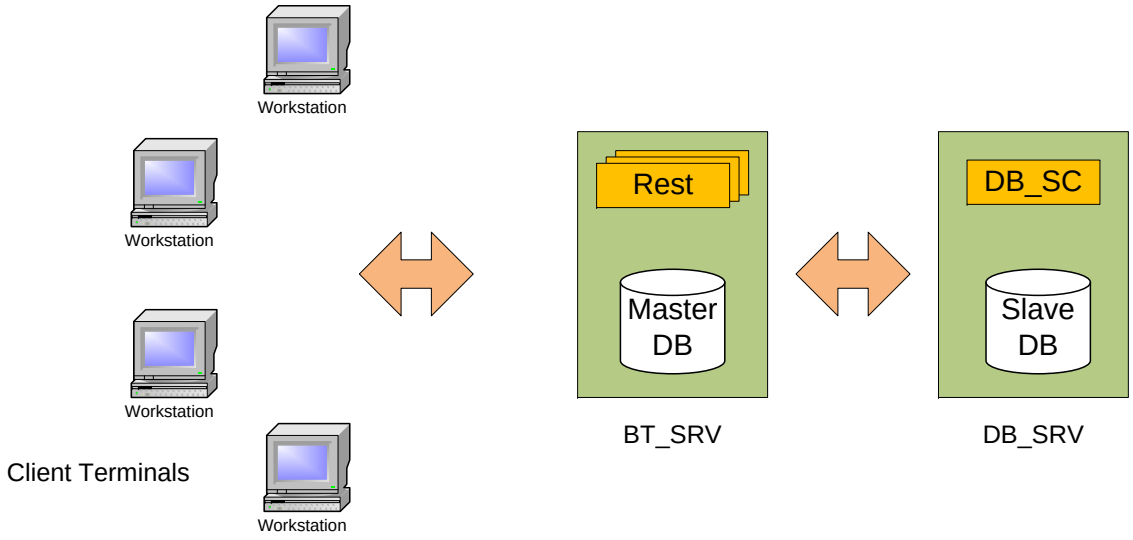
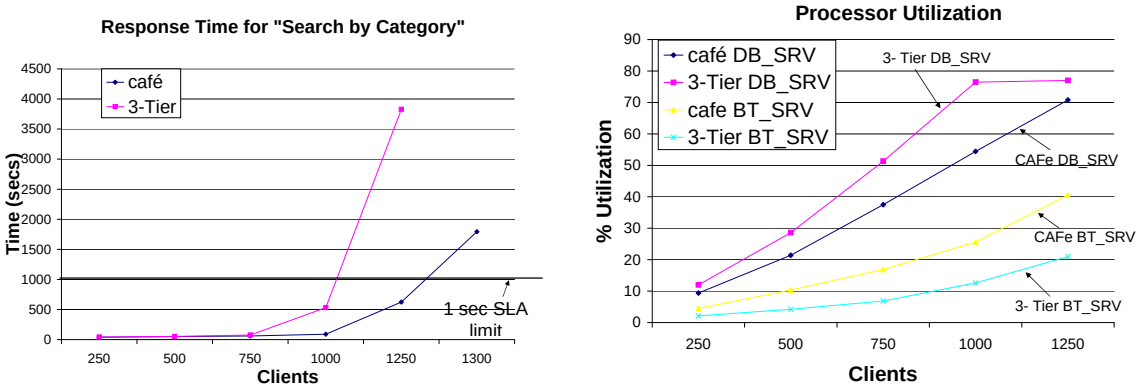


Figure 59: Deployment of CAFe Suggested Assignment

slave instance is run in DB_SRV. All Business Tier components and the web server run in BT_SRV. These components make the database call on the master instance in BT_SRV. Only component DB_SC (which belongs to service "Search By Category") makes the database call to the slave instance (in DB_SRV). The component DB_SC is thus moved to DB_SRV, while all other components run in the BT_SRV.

Figure 60a shows the response times of the most loaded service "Search By Category" for the CAFe deployment. By comparison, the original response time with the 3-Tier de-



(a) Response Time

(b) CPU Util

Figure 60: Performance of CAFe Installation

ployment is also provided. The comparison shows that the CAFe deployment increases the capacity of the application. The solid line (Time = 1.0) parallel to the Client axis signifies the SLA limit of 1 sec. The response times for the 3-Tier deployment crosses the line just above 1,000 clients. In contrast, the CAFe deployment the response time graph crosses the line at just over 1250 clients, which provides an improvement of $\sim$25% in application capacity.

Figure 61 shows the processor utilization for the two cases. In the CAFe installation the DB_SRV is less loaded than in the 3-Tier deployment. The BT_SRV utilization also shows the CAFe installation uses more CPU time than in the 3-Tier installation. This result is expected since CAFe tends to balance out the component utilizations across the given machines.

CAFe's balancing is not perfect,however, since DB_SC (the database component of the "Search By Category" service) consumes more processor time than all other components. This result indicates that load balancing the DB_SRV on multiple components and moving the components to different machines may be advantageous.

IX.3.7 Algorithm for Component Replication and Placement

Its shown in the previous section how some components take up more resources that others. It depends upon the amount of user hits on it. Accordingly it may become necessary to deploy multiple instances of the software components that implement the highly loaded services so that the total load can be balanced between different instances. An important question stems from determining which components need to be replicated for load. balancing and how many resources are needed to map these components on to the resources.

Manifestation in RUBiS: Figure 61 shows the previous figure with the lines corresponding to 3-tier processor usage removed. The processor utilization of the two servers in RUBiS are shown. In `DB_SRV` there is only one component `SearchItemsByCat` which takes up 70% of processor time when the number of clients reach around 1,300.

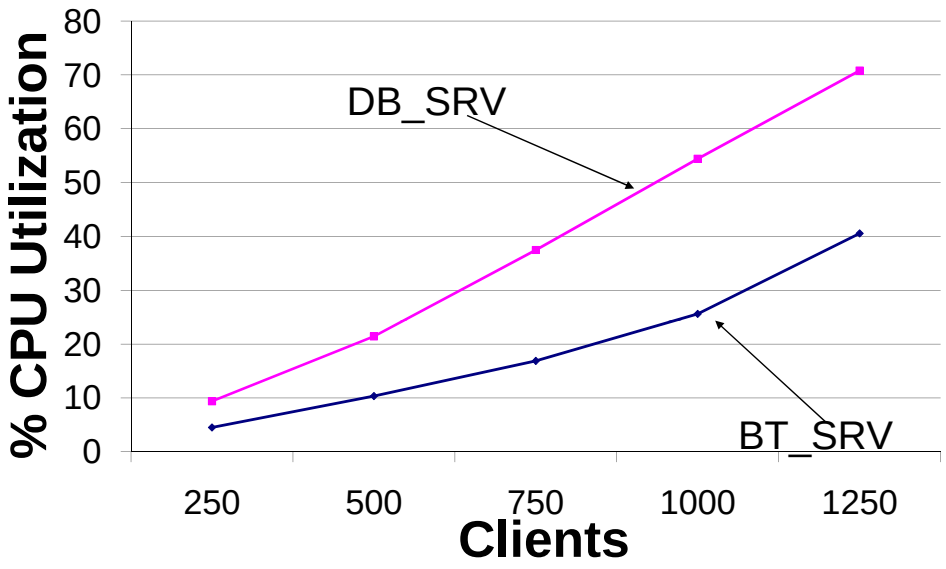


Figure 61: CPU Utilization

At the same time, it is observed that the other machine is loaded only upto 40%. Thus, there is an imbalance in resource usage. Allocation algorithms developed in prior research [30, 34] cannot improve this situation since the single instance of component `SearchItemsByCat` takes up significant CPU. To improve this situation, there should be another instance of `SearchItemsByCat` component and the load could be distributed between the two. One of the components could then be placed onto `BT_SRV`, so that the overall load ($70+40=110$) is balanced between the two servers (55 each). This will make it possible to handle more clients since now the utilization of both servers can be increased to around 70.

Thus by component replication and controlling the amount of load on a component, the number of resources required can be controlled and utilized by the component. The resource required by a component is referred to as the size of the component. The challenge now is to figure out the size of each component instance that will help in balancing the load. This is a non-trivial problem. The problem is more acute when trying to determine

component placement at design-time. Here accurate models which estimate the component resource requirements as well as performance for a particular placement are needed.

MAQ-PROWESS Solution: The solutions are presented for determining the replication requirements and placement decisions for software components that implement the different services offered by the web portal. The lower bound on the total number of machines required for a web portal can be calculated from the expected processing power required in the following way:

$$\lceil Ld \rceil / m, \tag{IX.1}$$

where Ld is the total processing power required (sum of the cpu requirement of all the components) and m is the capacity of a single machine

The problem of allocating the different components onto the nodes is similar to a bin-packing problem [15]. The machines are assumed to be bins while the components are items, where the items need to be placed onto the bins. It is well-known that the bin-packing problem is NP hard [74]. Thus, popular heuristics like first-fit, best-fit or worst-fit [15] packing algorithms need to be used to come up with the allocation.

It has been shown that these algorithms provide solutions which are around 2 times poorer than the optimal solution in the worst case. Thus the number of bins required could go up to $\lceil 2 \times Ld \rceil / m$. This would obviously mean that significant slack or idle processing power will remain in the machines.

Chapter VIII presented an extensive study on the performance of the different bin-packing heuristics under various environments. It was found that the average size of items affected the heuristics results. This is shown in Table 24. Here all quantities are mentioned in terms of percents. The percents are percentage of a single bin size. So an item size of 20% means that the resource requirement of a component is 20% of the total CPU time. The table shows the probability of finding an allocation of the given items onto the bins with different values of slack (difference between total bin capacity and total item sizes) and different item sizes.

Item Size	% Slack			
	0 - 5	5 - 10	10 - 15	15 - 20
0 - 30	34.84	97.96	99.97	100
0 - 50	10.57	65.49	96.14	99.67
0 - 70	26.44	65.68	93.02	99.14
0 - 100	100	94.93	99.34	99.64

Table 24: Success Rate of Heuristics on Solvable Problems:Courtesy Chapter VIII

For example, the entry of the second column and first row is 97.96%. This means that if there are items with average size of 30% (row value) of bin size and slack between 5 to 10 % (column) of bin size, then the chance of finding an allocation is 97.96%. This also means that if the item sizes are around 30% then the heuristics can find an allocation using up to around 10% more space than the total item sizes. Thus the expected number of machines required would be $\lceil 1.1 \times Ld \rceil / m$ which is much less than the worst case of 2 times the total item sizes.

This insight is used in the component replication and allocation algorithm developed in this chapter. Thus, the component sizes are kept around 30% which means the component resource requirement is kept around 30% of total processor time. This can be done by figuring out the number of clients that drive the utilization of the processor to 30% due to that component and allowing only this many clients to make calls on a single component instance. This can easily be implemented by a sentry at the web server level which monitors the incoming user requests. Algorithm 5 describes the component replication and placement algorithm. It performs a number of functions as follows:

- **Capacity Planning:** It computes the number of machines required for a target number of customers. It also minimizes the number of machines.
- **Replication & Load Balancing**: It computes the number of replicas for each component and how the load on a component is to be distributed to the individual replicas
- **Component Placement**: It computes the mapping of the different components onto the nodes.

Algorithm 5 uses two subroutines, `Placement` and `MVA`. Placement places the components onto the machines by using a bin packing heuristic. In this work a worst-fit heuristic has been used. MVA is the Mean Value Analysis algorithm that uses the enhanced analytical models that are developed to compute performance characteristics of a closed queuing network. It returns the response time of the different transaction classes along with the utilization of each component and each machine.

Algorithm 5: Replication & Allocation

```
begin
    // Initially, use 2 machines in a tiered deployment
    ;
    // All business logic components in first machine
    ;
    // Database in second machine, Default Deployment Plan DP
    ;
    N = init_clients
    (RT,SU,U) = MVA (DP, N) // Compute Initial Component Utilizations
    (DP) = Placement (SU, P) // Find a placement of the components
    ;
    while N < Target do
        (RT,SU,U) = MVA (DP, N)
        if ∃i : SU_i > 30 then
            Replicate (i); // Create New instance of Component i
            ;
            // Place new component on same machine as i
            (RT, SU, U) = MVA (DP) // Calculate new response time
            (DP) = Placement (SU, P) // Update Deployment Plan
        end
        if ∃i : RT_i > RT_SLA then
            // add new machine
            P = P + 1
            (DP) = Placement (SU, P) // find new placement
        end
        N += incr // Increase Clients for next iteration
    end
end
```

Initially, the algorithm starts with a default set of components needed for each service, uses a tiered deployment, and assumes a low number of clients, say, 100 (Line 5). A 3-tiered deployment will traditionally use a machine per tier. The components of each type are placed in the respective machines. The algorithm starts by estimating the performance characteristics of the application and placing the different components onto the given machines (Lines 5 & 5).

Next, the algorithm enters an iterative loop 5increasing clients with each iteration until

the target number of clients is reached. At every iteration MVA is used to estimate the performance requirement (Line 5). If any component reaches 30% utilization (Line 5), then another instance of the component is created and initially placed in the same machine as the original. Then MVA is invoked to estimate performance and the components are again placed onto the nodes. Similarly, if at any point the response time of any transaction reaches the SLA bound (Line 5), then another machine is added to the available machine and the placement heuristic is invoked.

This iterative process continues until the target number of clients is reached. Since the heuristic is one of the popular bin packing heuristics and the components are kept within a maximum of 30% resource utilization, it is ensured that near-minimum resources are used.

IX.4 Evaluating The Replication And Placement Algorithm

This section presents results that empirically evaluate the two properties, *i.e.*, minimizing resources and supporting increased number of clients, of the deployment plan for RUBiS web portal produced by Algorithm 5.

Recall that Algorithm 5 requires an analytical model of the system to compute the component placement. Using the process described in Chapter V, a multi-class closed queuing model shown in Figure 62 was developed for a scenario comprising two machines. One machine acts as the joint web server and business tier server while the other operates as the database server. A queue is modeled for each of the resources in the machines, *i.e.*, CPU and disk. Each service is modeled as a job class.

MVA is used to solve the closed queuing model. It uses the load dependent service demand functions (see Equations V.2 and V.3). Since multiple processors are used, the MVA algorithm is enhanced with the adjusted value of the service rate of waiting jobs on the CPU using the correction factor explained in Section V.1.4.2.

An extra queue for the database server is modeled to account for the software contention or locking that occurs due to overlapping queries being executed concurrently as discussed

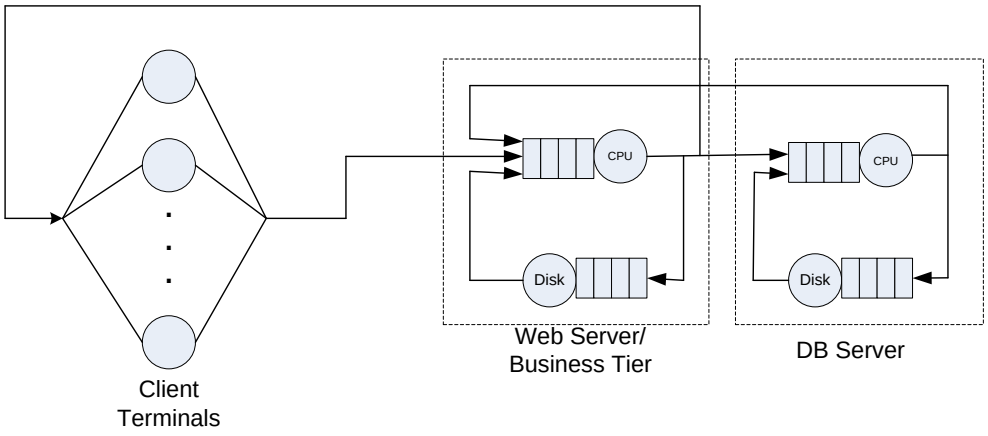


Figure 62: Queuing Model of RUBiS Scenario

in Section V.1.4.2. Thus, only the overlapping queries will visit this queue and will suffer some additional delay, which simulates the delay due to waiting on a software lock.

1. Minimizing and Efficiently Utilizing Resources:

In a traditional tiered deployment, each tier is considered atomic and hence all its functionality must be deployed together. In contrast, for a component-based system where services are implemented by assembling and deploying software components, it is possible to replicate and distribute individual components over the available resources. It is argued that this flexibility can make better usage of resources compared to a traditional tiered architecture.

Figure 63 compares the number of machines required to support a given number of clients for a range of client populations. For every value of client population that was experimented with, the response time of the client requests remained within the SLA-prescribed bound. It can be seen that for a majority of the cases our algorithm finds an allocation of the components that uses a reduced number of machines compared to the traditional tiered deployment.[1]

Table 25 shows the response times and the utilizations of the different processors for a total client population of $2,000$. A tiered deployment requires 4 machines to serve $2,000$

[1]The experiments were conducted in our ISISLab http://www.isislab.vanderbilt.edu/

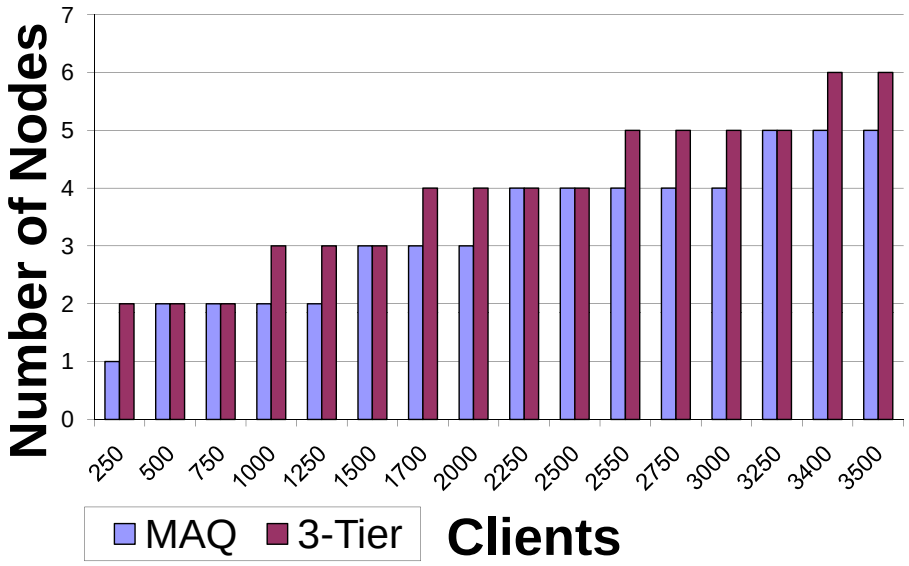


Figure 63: Node Usage of Tiered and MAQ-PROWESS

clients, while MAQ-PROWESS requires only 3 machines – an improvement of 25%. The table clearly shows that in the tiered deployment Node 3 is mostly idle(17.47%*utilized*). MAQ-PROWESS identifies idle resources and intelligently places components resulting in a minimum of idle resources.

		Node Utilization			
Deployment	Response Time (msec)	Node 1	Node 2	Node 3	Node 4
Tiered	270	51.06	79.08	17.47	78.86
MAQ-PROWESS	353.5	87.32	57.41	65.04	

Table 25: Response Time and Utilization

Figure 64 shows the resulting allocation of the different components in the deployment of RUBis web portal. Using multiple instances of components and distributing them in an intelligent way helps in effective utilization of available resources.

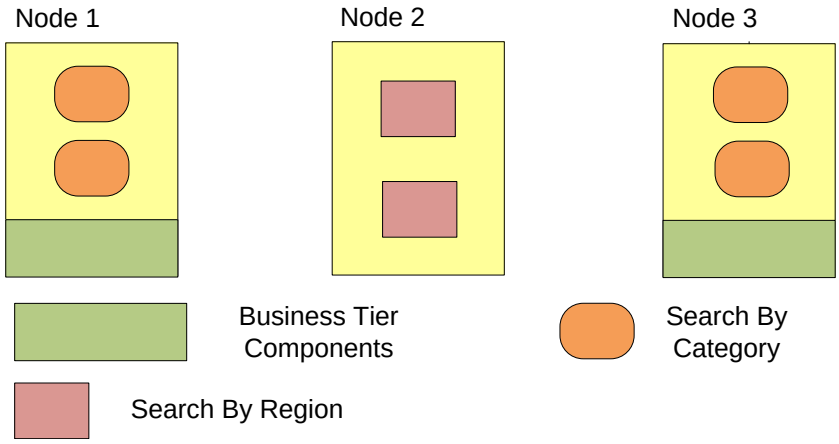


Figure 64: Allocation of Components for 2,000 Client

Figure 65 presents the coefficient of variance (CV) of the CPU usages for the three machines used in this experiment. It can be seen that the CV for the tiered deployment is much higher than the MAQ-PROWESS deployment. This signifies that the MAQ-PROWESS deployment uses the processors in a more balanced manner than the tiered deployment reinforcing our claim that MAQ-PROWESS effectively utilizes resources. The outcome is the ability of MAQ-PROWESS to handle more incoming load while preventing a single Node to become the bottleneck as long as possible.

2. Handling Increasing Number of Clients:

Our MAQ-PROWESS algorithm also enables increasing the number of clients handled using the same fixed number of machines compared to a tiered architecture. This can be achieved with a slight variation of Algorithm 5 where the number of nodes are fixed initially to some value. The algorithm terminates as soon as the response time reaches the SLA bound.

Using the result of three nodes obtained in the previous result, additional experiments were conducted. The allocation decisions made by MAQ-PROWESS are used to place the components on the machines and the number of clients is gradually increased till their response times reach a SLA bound of 1 sec. In comparison, the tiered deployment is also used to host the same number of clients.

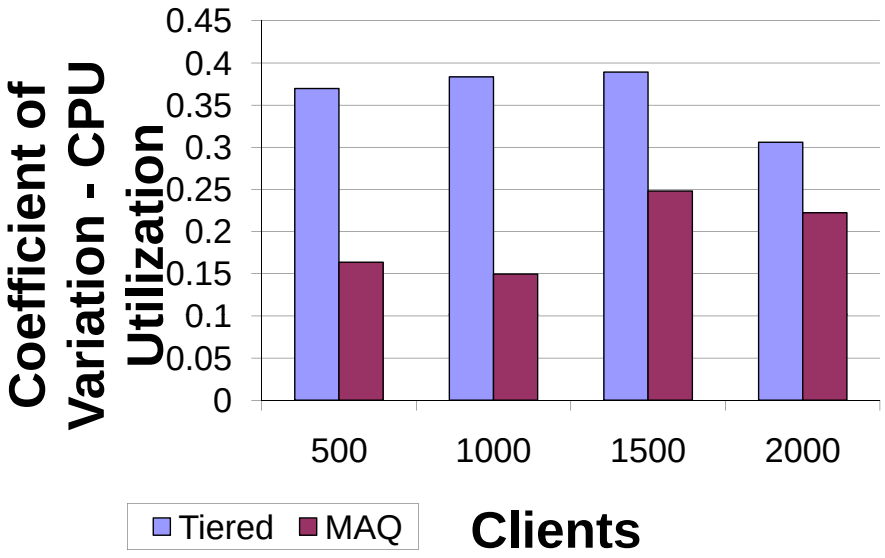


Figure 65: Coefficent of Variation of Node Usage

Figure 66 shows the response time for both the tiered deployment and the MAQ-PROWESS deployment. It can be seen that the tiered deployment reaches a response time of 1 sec at around 1,800 clients while the MAQ-PROWESS deployment reaches a response time of 1 sec at around 2,150 clients. This result shows an improvement of 350 clients or around 20%.

IX.5 Conclusion

This chapter presented a framework composed of a profile based analytical model and component replication and allocation algorithm of a web portal. The framework advocates a profiling method by which typical queuing models of web portals can be made more accurate with the use of load-dependant service demands of individual services on the processor, correction factor for easily estimating multi-processor activity and additional queues to model software contention causing blocking effect for concurrently running inter-dependent queries. It also develops a component replication and allocation algorithm

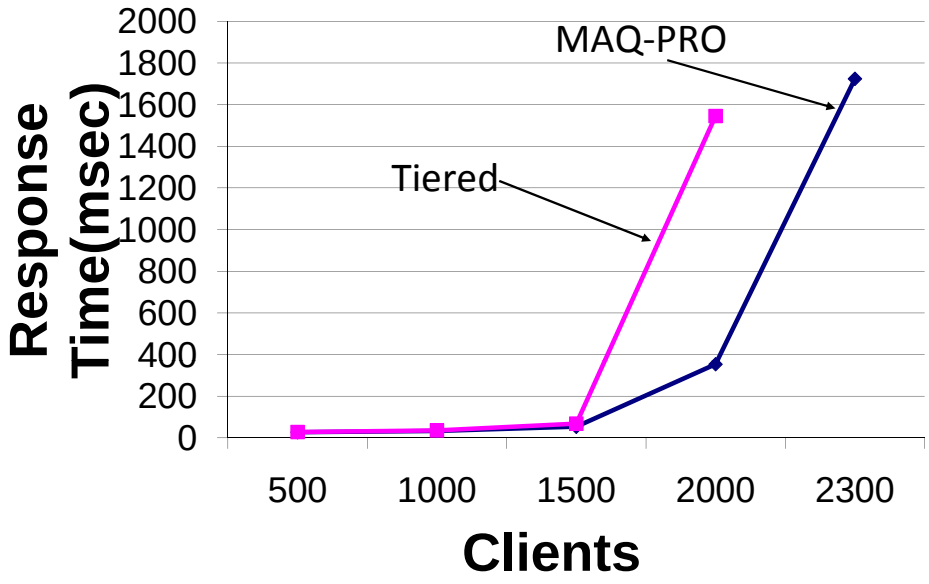


Figure 66: Response Time for Tiered and MAQ-PROWESS

which makes use of the above analytical model in coming up with proper deployment strategy that specifies replication count for each component and placement of the same onto the machines such that performance such as response time is dependable and within SLA bounds while resources used are minimized and client population is maximized. It has been shown that by keeping the resource utilization of each component within a certain threshold such as 30% of processor time, the resources can be utilized better. The framework helps in coming up with a deployment plan that uses lesser machines for the same number of clients compared to a tiered deployment for most situations. It also can accommodate more number of clients for a given set of hardware. The framework, result data and algorithm is available at http://www.dre.vanderbilt.edu/ nilabjar/MAQ

CHAPTER X

MODERN DAY DATA CENTER

The previous parts of the dissertation detailed a method to place components in the various nodes of the given deployment in such a way that the performance of the application can be enhanced. This chapter looks at modern day data center features and looks at how the techniques presented in the earlier chapters can be applied to solve certain problems.

X.1 Introduction

Emerging Trends and Challenges. In todays scenario, data center management has to spend a majority of their budget on maintenance of their ongoing operations. The reason for it is mainly due to factors such as

- *Dedicated server for each application*: Each application requires its own set of system configuration. Moreover it also needs its own version of third party libraries and tools which conflicts with other applications.

- *Design for peak load*: Each application gets a server that is designed to handle its peak load. Even though the frequency of that peak load can be really low, no one wants to take a risk. This is because it is not so easy to migrate applications from one resource to another.

- *Fault Tolerance*: Most of todays applications are not fault tolerant. So standby servers need to be kept which perform little or no work but incur cost.

- *Application Scaling*: Scaling up or down an application in response to load fluctuation is also not easy. Its not possible to sell off a part of a machine while scaling down when workload reduces. Thus when workload is less, most of the resources

are kept idle. When workload increases, the application needs to scale up and new machine need to be bought.

Most of the above factors contribute heavily to low utilization of the machines and also inflexibility in the face of changing demand.

Solution Approach: Judicious use of Virtualization and Cloud Computing. Using recent technologies such as cloud computing and virtualization can help in a long way to solve some of the above problems. Virtualization provides the following

- Virtualization permits a single physical machine to run multiple instances of a virtual machine(VM). These instances are isolated from each other.
- Automated management tools allow for the provisioning of resources across the different virtual machines. Thus it helps to scale up and down depending upon requirements.
- The entire operating system and the environment can be stored on a virtual disk. This helps in duplicating the VM.

Thus virtualization helps to eliminate most of the shortcomings mentioned above. Each virtual machine can be assigned to a separate application with its own environment and configuration and tools/libraries. In this way they can be run on the same machine and average utilization will also be better. In case of peak load, it will be necessary to allocate more resources to the loaded application while the others can still run on less resources.

Virtualization helps in segregating applications and allocating required resources to each application. However, virtualization alone cannot take care of the situation when workload of the applications in an physical host changes with time. For example, if multiple applications face peak load at the same time and the total resource requirement is beyond the capacity of the entire machine, nothing can be done. The other option is to allocate resources to the application such that the peak workload of the applications are always

handled. In this case however, the machine resources will be mostly left unused when the workload is less than the peak. Berkley researchers [47] state that real world server utilization in data centers range only between 5% to 20%. And for most services, peak workload exceeds the average load by factors of 2 to 10. This data clearly indicates that if resources are provisioned based on peak workload there is a large amount of resources which are left idle. Whereas if resources are provisioned based on average workload, there will be large amount of performance degradation at peak workload. So, what is needed is an on-demand resource provisioning system which can help in allocation and releasing resources on the fly whenever workload increases or decreases.

Recently developed, cloud computing has the potential to help provide resources on demand. Cloud computing is basically internet computing where computers, software and information can be leased over the internet. Users can lease a computer when requried, load software, perform computation and then return when done. All of this can also be automated by running scripts and invoking APIs. A computer leased by an user is essentially a virtual machine which is hosted is an actual physical machine. Thus virtualization is essential to cloud computing. Thus an application which wants to scale up resources to cater to an increase in client population can lease the required number of machines for a specific time and release them when the workload lessens.

X.2 Related Research

This section discusses related research in view of the challenges described in the above section. The related work is broadly categorized into three sets. Each of the sets is analyzed individually.

Virtual Machine Placement and Migration Using Heuristics: There have been a lot of work which proposes heuristics to come up with placement decisions of virtual machines

onto physical server. Urgaonkar et. al. [78] has used virtual machines to implement dynamic provisioning of multi-tiered applications. In their work, they define a flexible queuing model to determine how much resources are required to be assigned to each tier. At runtime the actual arrival rate is measured and adjustments are made by increasing or decreasing the provisioned machines to each tier. Each tier of the application is assigned to a VM and they are loaded in every machine. For each physical host only a single VM can be run. Wood et. al. [82] use a similar infrastructure to Urgaonkar et. al. as described above. They mainly concentrate on dynamic migration of virtual machines to apply dynamic provisioning. Here they identify a server which has a hotspot due to overloading. Then they identify the VM which needs more resource. This VM needs to be relocated to some other physical host which is under-loaded. They define a unique metric based on the consumption data of the three resources, cpu, network and memory. This is termed as the volume of a physical server or virtual machine.Cunha et. al. [16] develops a comprehensive queuing model to model virtual servers. They assign each class of jobs in an application onto a virtual machine. They introduce a pricing model which gives rewards for throughput to be within SLA limits and penalty for throughput going above. A constraint on response time with a probabilistic guarantee is imposed. The main system model is built out of a queuing model and non-linear optimization is used to solve the placement problem. Menasce et. al. [43] also comes up with a similar system model comprising of multiple classes. An analytical model which is based on a queuing model has been developed but they modify it to handle prioritized customers. In their work, a beam search technique is used to find the optimal placement of the virtual machines in the different physical hosts which is a combinatorial search technique [68].

[78] and [82] do not relate the placement mechanism to an overall utility value to the data center. They attempt at increasing the throughput of the application only. The applications considered do not have multiple classes which is unrealistic. In the real world, there is the need to do service differentiation by which certain classes of jobs get more

resources. [43] also does not have a nice utility model. The placement search technique may also become complicated. Using heuristics to come up with the placement could be a better option. [16] has a nice utility model of the data center. They also use multiple classes which is realistic. They assume to assign every class onto a single virtual machine. This may not be cost-effective in a situation where an application has numerous classes. The search technique also could be improved.

All of the work above do not account for power usage. This is very important in the recent scenario of large scale data centers. The main motivation is in minimizing the power used and also the number of machines required. The related work in the next section deals with power aware solutions.

Power aware management of virtualized environment: Cardosa et. al. [13] comes up with a heuristic algorithm which minimizes the number of machines in a data center. They propose a utilization function which maps resource allocation to the utility of an application. In the virtualized environment each virtual machine can be set to a minimum, maximum and a fraction of the total cpu percentage that it can use. This is used in this work. It is claimed that this will enable them to come up with better placement which will minimize the number of machines required. Verma et. al. [79] comes up with detailed power cost of running applications on different machines. They consider heterogeneous machines which will have different power efficiency. A simple heuristic is developed based on the power consumption models which will reduce the amount of power used in a data center. They also extend it to the run-time domain where they compute the migration cost of virtual machines and try to migrate virtual machines by minimizing cost.

The utilization model given in [13] is mapping resource utilization with application utility. Finding such mapping is difficult [32]. Instead it is more straightforward to relate throughput with utility and then map resource allocation with throughput. This will need a robust analytical model for relating throughput with resource allocation. [79] provides nice power models which can be used in any algorithm for virtual machine placement. But their

solution need to be used with analytical performance models which will relate resource allocation to throughput and to overall utility. It is also important to consider multiple classes of jobs within an application.

Autonomic management of virtual computing environment using control theoretic approaches: Padala et. al. [53] provide a control theoretic solution. They run each tier of the application on each virtual machine and carry out extensive black box profiling of the applications and builds a approximated model which relates performance attributes such as response time with the fraction of processor allocated to the virtual machine running the application. The controller is a two level controller comprising of a utilization controller for every virtual machine and a arbiter controller for overall control. The controller handles proper cpu allocation upon workload change on the application. They also implement QoS differentiation between applications under overload conditions. Wang et. al. [80] also has a two-level control architecture for virtualized environment. The load balancing controller ensures that the virtual machines are all load-balanced and the response time of the applications in all the virtual machines are the same. The response time controller then controls the cpu frequency of the machines to minimize it so that power is saved. The response time controller also ensures that the SLA bound for the response time is met. They assume that an application will only have a single job class. This is unrealistic in a real world situation where every application provides a number of services. The presence of multiple classes will influence the models of the application which are used to implement the controller. This will bring in inaccuracies in the controller behavior.

The above sets mostly discusses related work which use virtualization. Such work is mostly useful for cloud computing infrastructure providers. Providers of cloud infrastructure need to migrate and allocate various virtual machine instances over the actual physical hosts present. But from an user perspective, its more important to allocate and release machines on the fly. Thus judicious ways to use the auto-scaling features of cloud computing is important.

Moreno et. al. [49] recommends a nice architecture for elastic manegement of cluster based services. It consists of virtualized infrastructure layer that works with a VM manager and a cloud service provider. This extra virtual layer abstracts away the user from the low level details of the actual cloud provider. This helps in autoscaling resources with the least amount of disturbance to the user.

Waheed et. al. [26] proposes a reactive algorithm to allocate extra resources to a cluster farm when workload increases. This work monitors the response time of the various clusters and as soon as response time is violated in any of the clusters extra machines are allocated to the cluster.

Yang et. al [29] propose a profile based approach to the problem of just-in-time scalability in a cloud environment. In this approach, profiles capture expertsŠ knowledge on scaling applications dynamically. A profile consists of the different components of the software and how they should scale when workload changes. Guided by profiles, profile driver automates the setup and scaling of execution environments, which ensure just-in-time scalability of cloud applications.

None of the above work concentrates on figuring out the right amount of resources that are required over time for a given workload. In a cloud environment, with inherent ability to increase and decrease the resources available to the application, it is important to figure out the right amount of resources to provision and to increase the utility of the application in terms of SLA conformance and resource usage. It is also important to consider the relative cost of SLA violation, machine costs and costs of reallocation or reconfiguration of application in a cloud environment.

X.3 Solution Approach: On Demand Resource Provisioning Using Look-Ahead Optimization

A simple way to allocate resources in response to the workload pattern described in Figure 67 would be in a reactive manner. For example, the incoming workload rate may be

monitored at all times and whenever the rate changes, the resource requirement is computed from the new value of workload using a performance model. The required resources are then allocated/released from the cloud infrastructure. A problem with the above strategy is that there is some amount of finite time in setting up a new server, deploy the application component instances and redirect client invocations. In many cases, this can take up minutes. The performance of the system will then suffer for that entire length of time. What is more desirable is that the workload changes be predicted from before and servers and computers be allocated before time such that when the actual workload increases, the system is ready to handle it.

To carry out this strategy, this dissertation leverages the work done by Sherif. et. al. [1, 2] in developing a predictive control approach to design self-managing computing systems. In such a strategy, the actions are selected by optimizing the system behavior over a limited prediction horizon. The workload to the application is forecasted for the future periods and also the system behavior is estimated from the predicted workload using a performance model. The optimization of the system behavior is carried on by minimizing the cost incurred to the application. The cost of the application will be a combination of various factors such as cost of SLA violations, leasing cost of resource and a cost associated with the changes to the configuration. The advantage to such a method is that it can be applied to various performance management problems from systems with simple linear dynamics to complex ones. The performance model can also be varied and corrected with system dynamics as conditions in the environment change like workload variation or faults in the system. The rest of the section gives details of the solution in terms of a web portal like Ebay or Amazon and uses the case study of RUBiS like in previous chapters.

Typical workload to large on-line systems such as Amazon or Ebay is highly varying in nature. Figure 67 shows the number of client arrivals each hour to the website for soccer world cup 1998. The figure clearly shows that the number of clients is highly variable with a cyclical trend in it which is affected by the time of the day. Resource allocation for such

systems is highly challenging. There will be different amount of resources required for handling the various values of the client population. If resources are allocated for the peak workload, then most of the time the resources will remain idle. Again, if resources are allocated for the average workload, then performance such as response time will suffer when there is peak workload. The ideal situation is when resources can be allocated when there is high load and released in times of low load. Cloud computing provides an infrastructure for implementing such a policy. As discussed above, the challenges to the development of such a policy are discussed below:

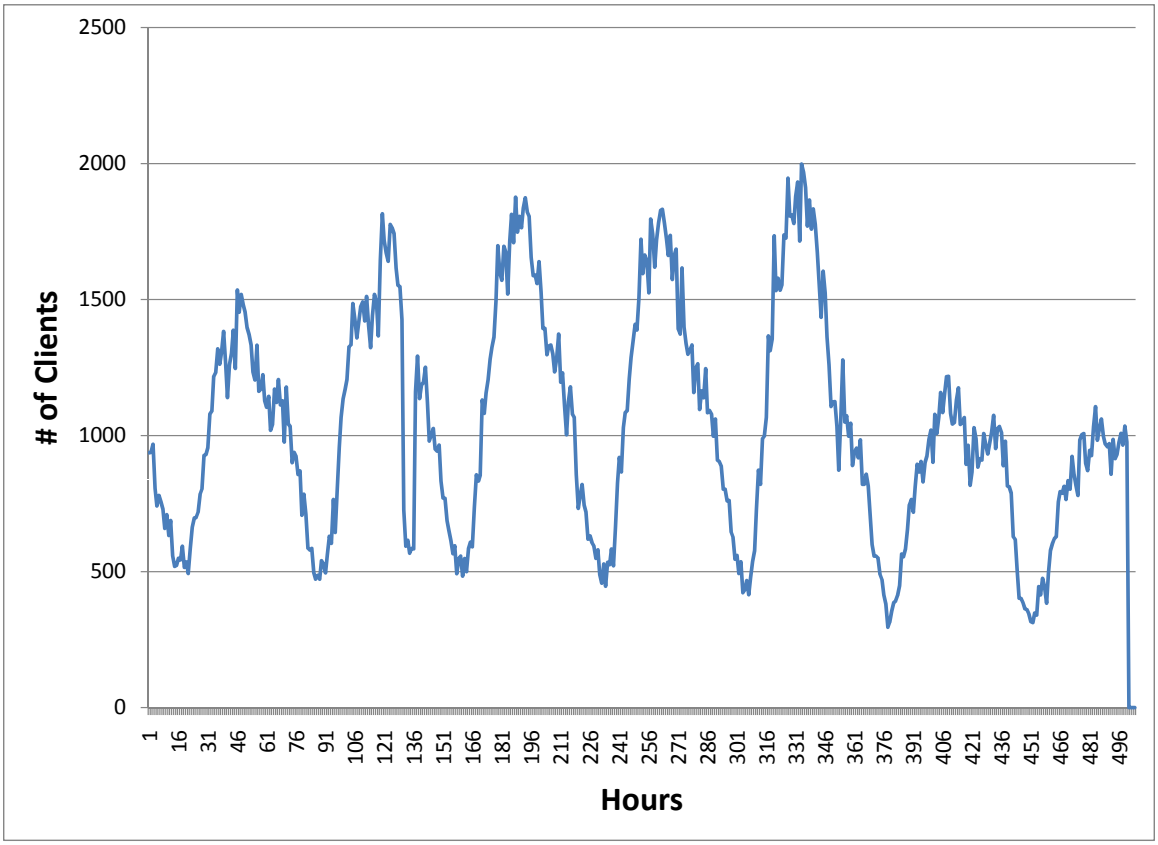


Figure 67: Client Population With Time

- **Workload Prediction** is an important step for on demand resource scaling. The installation of new resources and bringing them on-line will require some amount of

time and effort. It is not instantaneous. Thus its important that workload is predicted for some time in the future. If it can be done with some amount of correctness, it will help in anticipating the incoming load and allocate enough resources for that. Workload prediction can be implemented using previous historical data and identifying patterns within that.

- **Identifying resource requirement from given workload**. Required resources to handle a particular number of client population with desired response time needs to be figured out. Proper identification of resource requirements is important since resources need to be provisioned for a successful handling of incoming workload. This entire identification needs to be done statically. Previous chapters in this dissertation discusses how an accurate performance model can estimate the response time of an application given the amount of resources, the application specific profile data and the workload. Such a model can be used to extract the amount of resources required for given response time and workload.

- **Optimizing resource provisioning.** To optimize resource usage or minimize idle resources, the best way would be define an time interval and change resources as many times as possible as workload changes. In the limit this interval could be made infinitesimally small and resources are changed continuously. This will obviously ensure that the optimum amount of resources are always used. Obviously this is not possible since, changing resources is not spontaneous. It involves a number of action steps such as booting up new machines, redirecting clients to new servers, state updates, component connections, etc. All of these will take time and may cause performance degradation. Thus scaling up or down resources also involves cost and needs to be optimized.

The next section goes in detail into each of the challenges identified above.

X.4 Solution Details

This section goes into the solution for each of the challenges discussed above and comes up with potential solution for each.

X.4.1 Workload Prediction

Workload prediction needs to estimate the incoming workload of the application for future time periods. In this work, the idea presented in [2] is leveraged to estimate the future workload. In [2], an autoregressive moving average method (ARMA) has been proposed which is expressed in the following way:

$$\hat{\lambda}(k+1) = \beta\bar{\lambda} + (1-\beta)\lambda(k) \tag{X.1}$$

The equation X.1 represents a first order equation with a single constant β. For the workload shown in Figure 67 a second order ARMA filter gives good result as shown in [42]. The equation for the filter used is given by

$$\lambda(t+1) = \beta \times \lambda(t) + \gamma \times \lambda(t-1) + (1-(\beta+\gamma))(\lambda(t-2)) \tag{X.2}$$

The value for the variables β and γ are given by the values 0.8 and 0.15 respectively as mentioned in [42]

X.4.2 Identifying resource requirement

Given the workload incoming to the application, there is the need to find the resources required to handle the workload such that the SLA bounds are maintained. In the above look-ahead framework, there is the need to find out the response time of the application under various hardware configurations. For this, the analytical models discussed in the previous chapters need to be used. Here an algorithm is presented which makes use of the analytical algorithms presented in 6. The algorithm accepts the amount of workload given

Algorithm 6: Response Time Analysis

Input:
Ld Workload
Hw Total Machines
SD Service Demand for the job classes
Z Think Time

Output:
Response Time $R \leftarrow$ Vector of response times for all job classes

begin
// Use 2 machines with each tier in each machine
M = 2;
while $M <= Hw$ **do**
// Get response time and server utilization by running MVA
[R, U] = MVA (SD, Ld, Z);
i = maxUtil (U); // Get bottleneck tier number
// Add a machine and replicate tier i on it
M = M + 1
$M \leftarrow i$
end
end

by a vector of client populations, each member representing the number of clients in each job class. The number of machines provided ,the service demand of the components and the think time for clients is also given as input. Algorithm 6 initially creates a default placement strategy whereby it places each tier of the application onto a particular machine and then enters into an iterative stage. On each iteration, the algorithm makes a call onto the MVA algorithm which is the analytical model described in Chapter V. The MVA returns the utilization of each tier which can be used to find the bottleneck machine (the machine with the highest utilization). The tier present in that machine is then replicated and placed in a new machine which is introduced in that iteration. In this manner, the iteration continues until the total number of machines equal the given maximum machines.

X.4.3 Optimizing Resource Provisioning

This section details the resource provisioning method employed in this dissertation. This essentially works on the principles of look-ahead optimization detailed in Sherif et.

al. [2]. For this method to work, there is the need for a finite number if configurations available at all times. In the problem of just in time resource provisioning the number of machines can be increased, decreased or kept the same. If the number of machines that can be increased/decreased is kept within a particular finite value then the whole set of available configurations become finite.

Another factor is the choice of the look-ahead period. Having a too small look-ahead period may not get the optimum result, while a very large period will increase complexity. A very large look-ahead period also does not make much sense since the future estimations will have more errors and thus formulating a decision based on such data will make things more erroneous. Thus the number of look-ahead periods need to balance out the various trade-offs.

Algorithm 7 describes the algorithm for look ahead optimization. This algorithm is invoked at every time step to make a decision about the resource allocation in the next step. It accepts the number of look ahead steps, K as input and workload and application data. The workload data contains the number of client population in the previous time steps. The application data includes service demands for various classes of jobs and think times of clients. The algorithm also accepts the set of configurations that needs to be checked. This will typically include the number of machines that can be increased or decreased.

The algorithm iterates over the number of look ahead steps and calculates the cumulative costs. For every future time step, it computes the cost of selecting each possible resource allocation. To compute the cost of a particular allocation, it uses the algorithm 6 to compute the estimated response time for the particular machine configuration. Once that is calculated, it is used to calculate the cost of the allocation which is a combination of how far the estimated response time is from the SLA bounds, cost of leasing additional machines and also a cost of re-configuration. The cost of reconfiguration is computed based on the number of machines that needs to updated. Obviously re-configuration will incur some costs and thus the algorithm will try to reduce the amount of reconfiguration. Each of

Algorithm 7: Look Ahead Optimization

Input:
K look ahead steps
W Workload data
R_star SLA response time bound
H set of configurations
SD Service Demand for the job classes
Z Think Time

Output:
Deployment Plan $DP \leftarrow$ Map of each tier onto nodes
Machines $M \leftarrow$ Number of nodes
Cost $\leftarrow$ Total cost

```
1  begin
2    for i ← 0 to K do
3        // Estimate future workload
4        W_i = β × W_{i−1} + γ × W_{i−2} + (1 − β − γ) × W_{i−3}
5        forall H do
6            // get the response time from the estimated workload
7            R = Rsp_Time(W_i, SD, Z)
8            // compute cost based on response time and machine usage
9            Cost(R, R_star, H)
10       end
11   end
12 end
```

these cost components will have weights attached to them which may be varied depending on the type of application and its requirements.

X.5 Experimental Evaluation

This section presents the results of the look-ahead algorithm. Initially, it is shown how the algorithm decides on the resources to be allocated in a just-in-time manner so that the cost is minimized. Next, the effects of various different cost weightage is studied. This is important since different applications will have different weightage combinations. Such a study shows interesting resource allocation trends that will be required to provide the best value for a particular application.

X.5.1 Just in time resource allocation

This section shows how the look-ahead algorithm in Algorithm 7 prescribes just-in-time resource allocation as workload changes. The workload for this experiment is as given in Figure 67 which is the workload for the 1998 FIFA World Cup website. The cost function has the three components as discussed above. The weights on each component is the same.

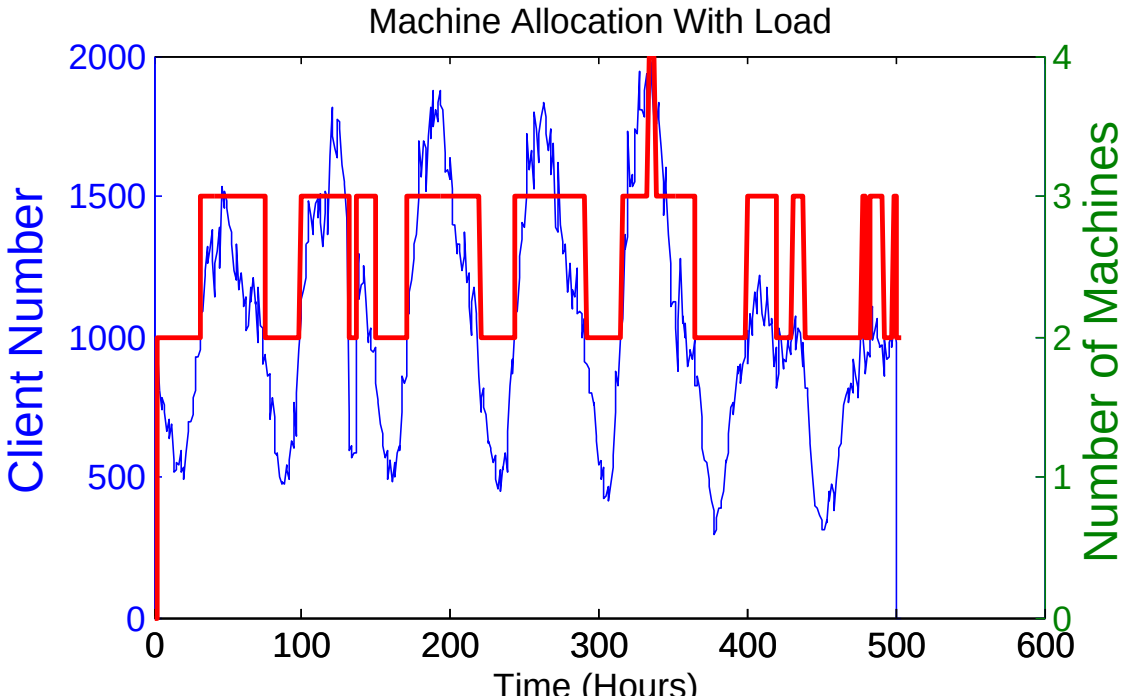


Figure 68: Just in time resource allocation with load

Figure 68 shows how the look-ahead algorithm prescribes the changes in the resources required as the incoming load changes. The computation is done on the basis of predicted workload which is done with the help of the ARMA filter which is given in X.2. Figure 68 clearly shows that the base resources required are 2 machines and it increases to 3 or 4 when the load is increased. The prediction of the look-ahead algorithm closely matches the incoming load. It prescribes resource increase whenever there is high load and less

resource when there is less load. Thus Figure 68 shows the effectiveness of the look-ahead algorithm and how it can save cost while also assuring that the performance of the application is assured.

X.5.2 Resource usage under various cost priorities

This section looks at the resources required for different priorities in the cost function. As mentioned before in Algorithm 7 there is a Cost function used to determine the cost for various configurations. This cost function is actually comprised of three components, penalty for violation of SLA bounds, cost of lease of machine and cost of reconfiguring the application when machines are either leased or released. Each of these components has a weight attached to it and the system can be made to always minimize a certain component by increasing the attached weight to it to an arbitrary high value.

$$Cost = W_r \times (R_{sla} - R) + W_c \times M_k + W_f \times \|(M_k - M_{k-1})\| \tag{X.3}$$

Equation X.3 shows a sample cost function which is used in this work. Table 26 describes the components of the cost function. There are three parameters to the cost which are the three weights of each component. By varying the various weights the priority of the application can be expressed.

Component	Description	Unit
W_r	Penalty for SLA violation	\$/sec
W_c	Cost of Leasing a Machine per hour	\$/machine
W_f	Cost of reconfiguring application	\$/machine
R_{sla}	SLA given response time	sec
R	Maximum response time of application	sec
M_k	Number of machines used in the k^{th} interval	Numeric
M_{k-1}	Number of machines used in the $k-1^{th}$ interval	Numeric

Table 26: Components of Cost Function

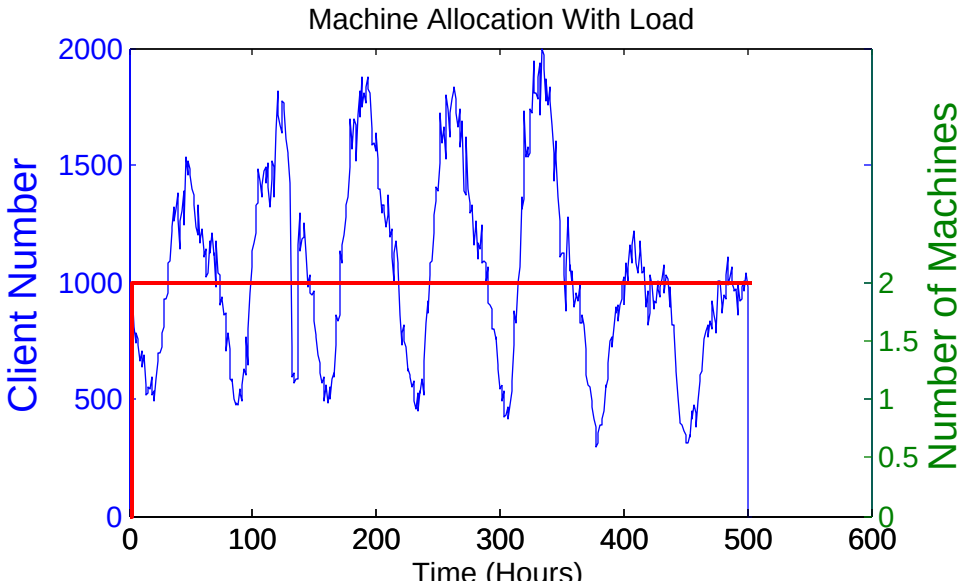


Figure 69: Resource Allocation for Low SLA Violation Cost and High Machine Cost

The resources allocated in the various time intervals will depend upon the weights assigned to the various components of the cost function. The rest of the section studies the different trends of resource allocation and how they are influenced by the varying weights of the cost function.

X.5.2.1 SLA violation against resource cost

This section studies the effect of SLA violation against cost of resources. In this study, the ratio of the cost of SLA violation against the cost of machines are varied while the application reconfiguration cost is assumed to be zero. Thus it is assumed that the application can be easily reconfigured with varying machines. The ratio of SLA penalty to machine cost is varied from $4:1$ to $1:13$. It is seen that there is significant difference in resource allocation between the different configurations. An application with high SLA violation penalty has stronger performance assurance whereas one with low SLA penalty has lesser performance assurance.

Figures 69,70 and 71 shows how the resources are allocated every hour over the entire

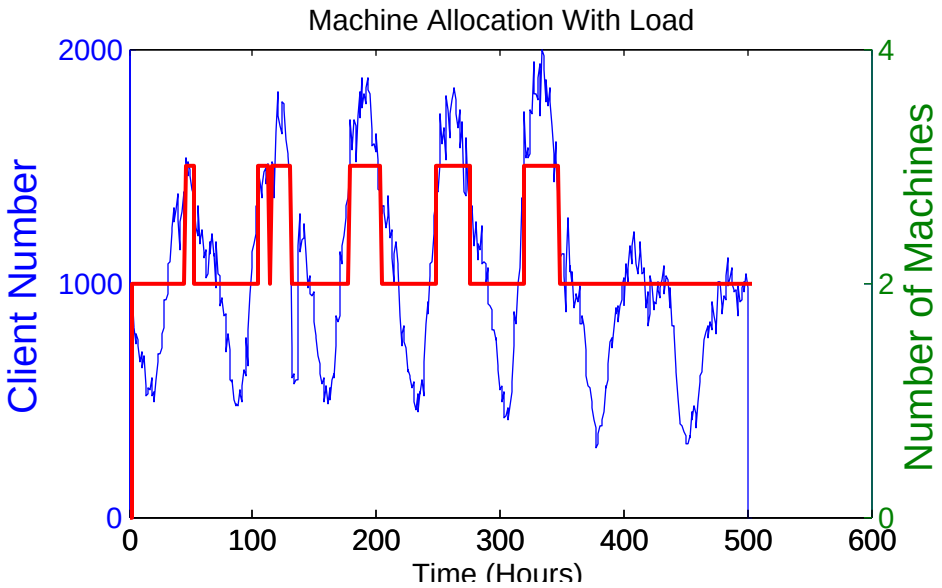


Figure 70: Resource Allocation for Medium SLA Violation Cost

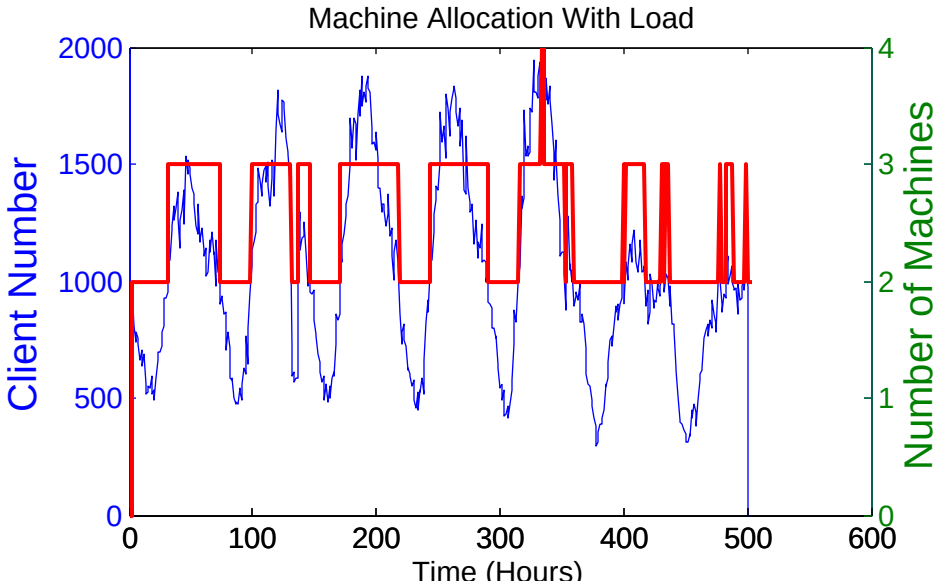


Figure 71: Resource Allocation for High SLA Violation Cost

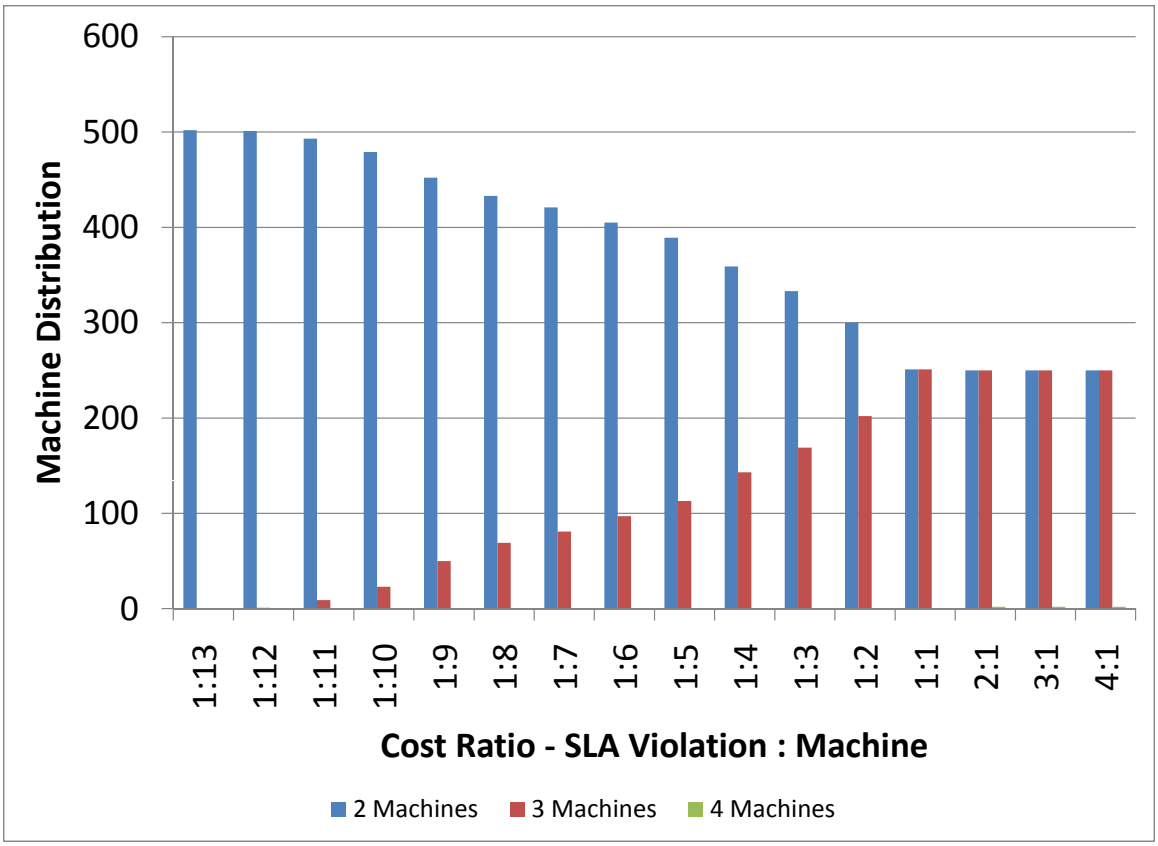


Figure 72: Resource Allocation for variety of systems

time period. The priorities of the application determine the difference in resource allocation. For a low performance assurance and high machine cost, the number of machines used is only 2 over the entire time interval. The cost of machines exceeds the cost of SLA violations and such a configuration will have to tolerate a number of SLA violations. Contrary to this configuration, for a medium performance assured system, Figure 70 shows how there are many intervals in which 3 machines are used. This balances the cost of machines and cost of SLA violations. For the highly assured application of Figure 71, there is much variation in resource usages with a number of intervals having 3 machines and also some having 4 machines. Here the priority is in assuring performance and the cost of machines is much lower.

Finally, figure 72 shows the distribution of number of machines required for a variety of systems ranging from highly assured systems (ratio of SLA violation penalty to machine

cost being > 4:1) to very weakly assured systems (ratio of SLA violation penalty to machine cost being > 1:13). In this figure, each point on the x-axis is a ratio of cost of SLA violation to cost of machine. The y-axis plots the number of intervals in which each type of machine is used. For example, for the point corresponding to cost ratio of 1:4, 359 intervals use 2 machines and the other 143 intervals use 3 machines. The ratio of SLA violation cost to machine cost increases as we move further down the x-axis. The figure shows the use of more 3 machines than 2 machines as we move to the right. This is because the relative cost of machines decreases to the right and the penalty of SLA violation increases.

The above study nicely shows how such a look-ahead algorithm can nicely compute resource allocation in a dynamic situation where both the workload into the system as well as the resources available are highly dynamic.

X.5.2.2 Effect of reconfiguration cost

The previous section did not account for the cost of reconfiguration of resources. Reconfiguring the application is necessary whenever there is a change in the number of machines used to host the application. It assumed that reconfiguration was easy and spontaneous. In a real-world scenario, this is not so. There is always a cost involved with reconfiguration. As mentioned previously, reconfiguration involves a number of tasks such as booting up new machines, redirecting clients to new servers, state updates, component connections, etc. All of this might end up stopping or suspending the application for some period of time which will cause performance degradation and will cause in penalty for lack of service to end users. The exact amount of penalty will depend upon what time the reconfiguration is done and on the number of incoming clients coming in. This penalty will be varying and can be minimized by choosing to do the reconfiguration at a time when the customer incoming rate is the minimum. Anyway coming up with the actual cost of reconfiguration is beyond the scope of this work. This work assumes this cost as a ratio to

the other costs of SLA violation and machine costs and evaluates the given system under high, medium and low values for this ratio.

The previous section illustrated results which showed how the resource allocation varied as the ratio of SLA violation cost to the cost of machines changed. In this section, the same resource allocation trends are studied in the presence of cost of reconfiguration. Three levels of the cost of reconfiguration is considered, low, medium and high. This cost is relative to the cost of both SLA violation and machine cost.

X.5.2.3 High SLA violation cost

Figure 71 shows how resource allocation is done when there is high SLA violation cost compared to machine cost. Here in every interval, the mean response time is below the SLA bound. And machines are allocated whenever it is needed. It is released again since there is cost of machine but only making sure that the SLA is maintained. When there is a cost of reconfiguration introduced, it will resist the changing of resources. It can be compared somewhat like inertia. Inertia in physical bodies resists changes to its current physical condition such as a body in rest resists movement while a body in motion resists slowing down. Thus the cost of reconfiguration will similarly resist the dynamic nature of resource allocation. Higher this cost, higher will be its resistance to the changes. This cost is expressed as the third component of Equation X.3. The weight W_f represents the level of inertia and it is multiplied by the change level which is the number of machines allocated or released. Initially when a small amount of reconfiguration cost is introduced, it does not effect much as shown in Figure 73. The resource allocation is similar to figure 71. There are small deviations, where the spikes in resource changes are a little wider in figure 73 than in figure 71. This is due to the inertia in change introduced due to some cost associated with change.

The effect of the cost of reconfiguration is more pronounced when it is made a little higher. Figure 74 shows a distinct change in resource allocation over the hourly intervals

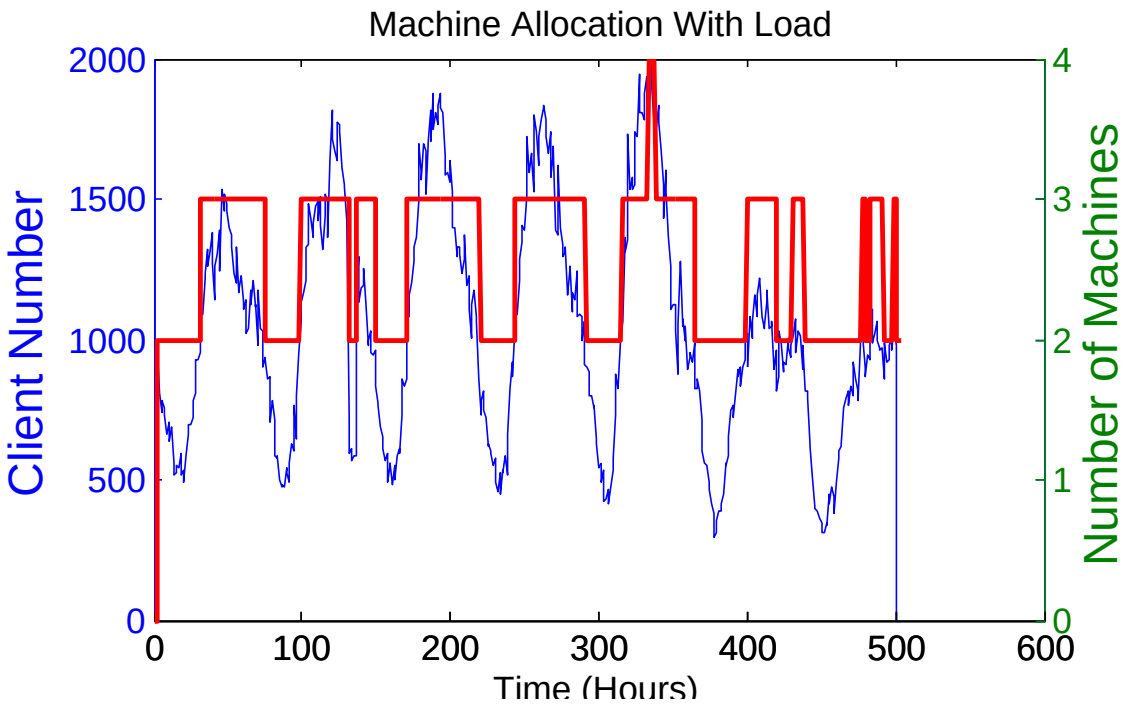


Figure 73: Resource Allocation for High SLA violation with low reconfiguration cost

compared to figures 71 or 73. In Figure 74, the number of machines increases to 3 at around the 40^{th} hour and remains steady. Somewhere around the 350^{th} hour it increases to 4 machines since the workload increased at that time. Subsequent to that, the workload decreased but the machines were never released since the cost of reconfiguration is much higher compared to the cost of machines. The changes of the machines around 40 and 350 hours was warranted because of the high SLA violation cost and the machines were never released even though the workload lessened since the cost reconfiguration was higher.

This behavior of resisting change is further pronounced in Figure 75 where there is even higher cost of reconfiguration. Here again there is an increase of machines to 3 at around the 40 hour mark and the machine is never released. The change to 4 machines which was seen in Figure 74 does not occur here because the cost of reconfiguration is much more higher that the cost of SLA violation. Thus even though there is SLA violation, it is only of a short duration (the peak workload around 350 hour) and is of lesser cost than the cost of changing resources. That the SLA violation near 300 hour was of a short

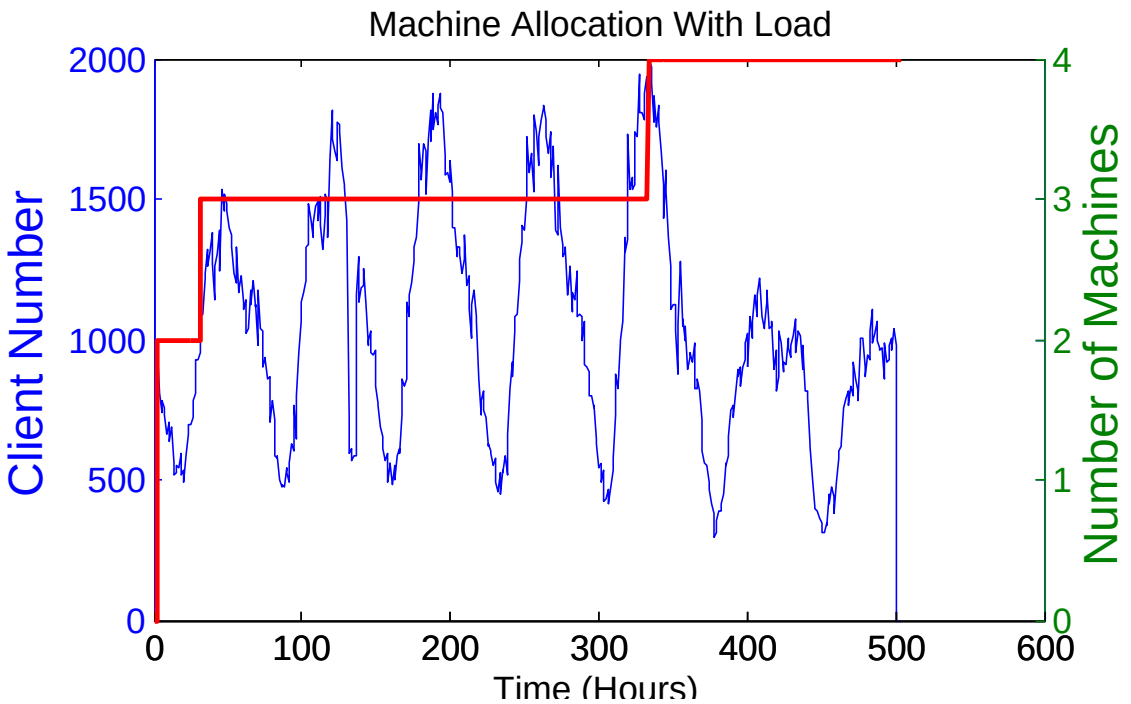


Figure 74: Resource Allocation for High SLA violation with medium reconfiguration cost

duration can be understood from the Figure 73 where there is a very short spike of machine allocation to 4 around that time. When the cost of reconfiguration becomes high the look ahead algorithm decides not to expend in the extra cost of reconfiguration to cover up that short SLA violation.

X.5.2.4 Medium SLA violation cost

In this section the effect of cost of reconfiguration is studied when the cost of SLA violation and the cost of machine is more or less the same. The figure 76 show the resource allocation trend when the cost of all three components are similar. The trend of resource allocation is very similar to Figure 73 with the exception of the short spike of allocating 4 machines around the 100^{th} hour mark. In the configuration of 1_1_1 there is no such allocation. This is because the cost of SLA violation is now lesser and the resistance to change due to reconfiguration cost does not allow such a short spike of new resource allocation.

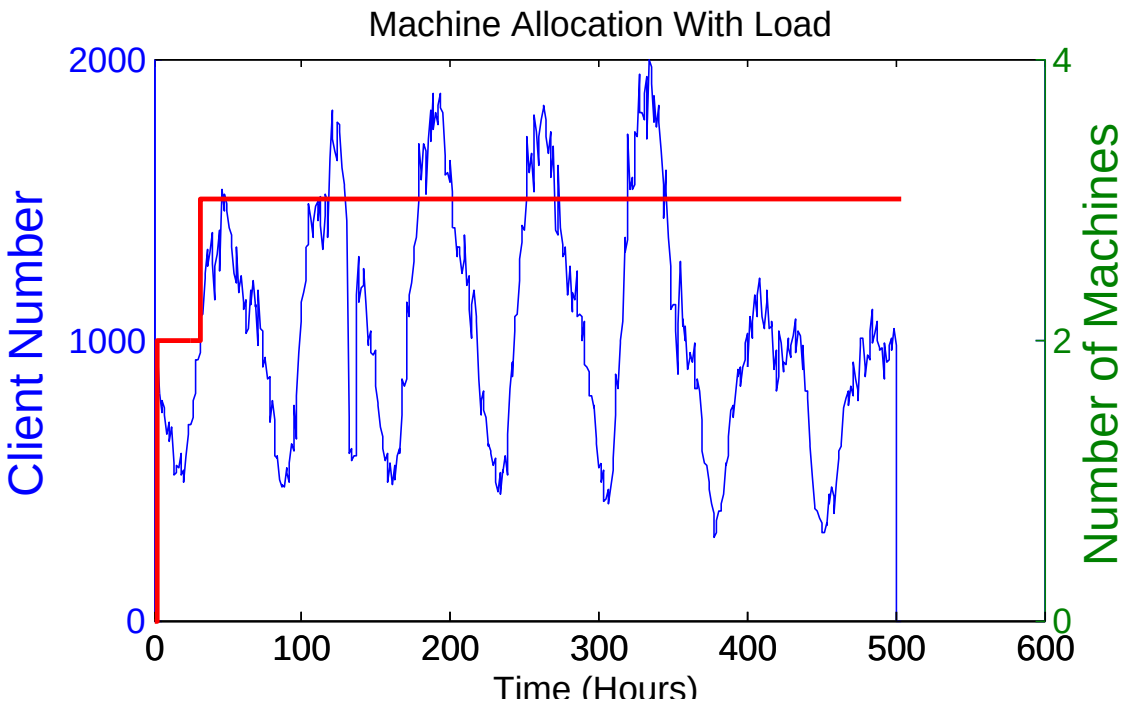


Figure 75: Resource Allocation for High SLA violation with high reconfiguration cost

Thus the application loses some money due to SLA violation but makes it up by saving more by avoiding reconfiguration.

The configuration 1_1_4 and 1_1_10 behave similar to each other but much differently from configuration 1_1_1. It can be seen in Figures 77 and 78. They are also very similar to configuration 4_1_10 shown in figure 75. In both these cases, the cost of reconfiguration is much higher than either of SLA violation or machine leasing cost. Thus the resistance to change is higher and thus once it reaches 3 machines around the 40 hour mark it does not release the machine nor does it try to increase it. Here SLA is mostly covered but it loses some money on machine cost but is offset by avoiding excessive reconfiguration. However there is a slight difference between the allocation of 1_1_4 and 1_1_10. This is shown in Figures 77 and 78. The 4^{th} machine is allocated in 1_1_10 a little later than 1_1_4. This is because the former has even higher reconfiguration cost and waits for the workload to become even higher to incur higher SLA violation cost.

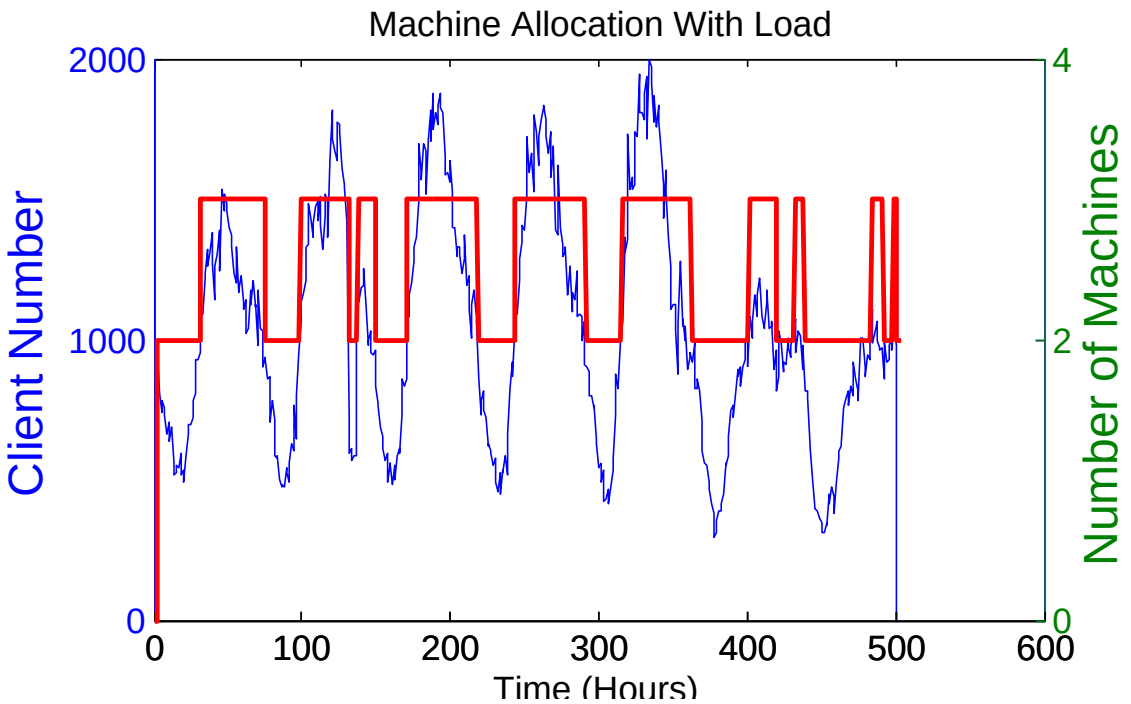


Figure 76: Resource Allocation for Medium SLA violation with low reconfiguration cost

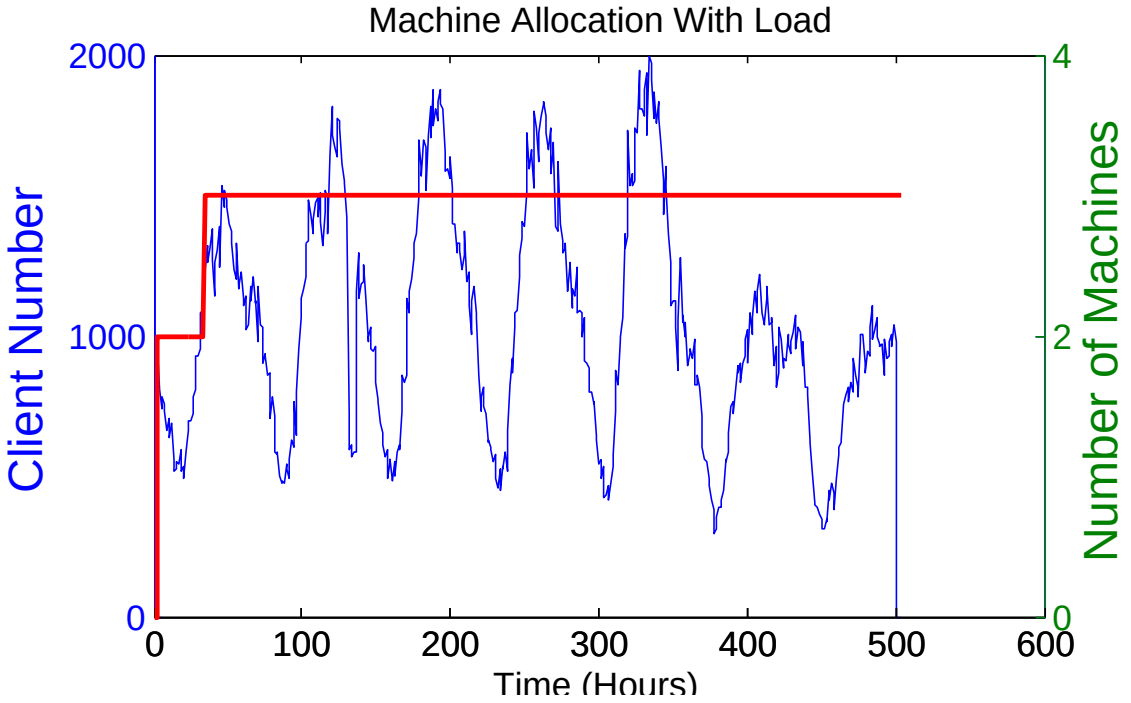


Figure 77: Resource Allocation for Medium SLA violation with medium reconfiguration cost

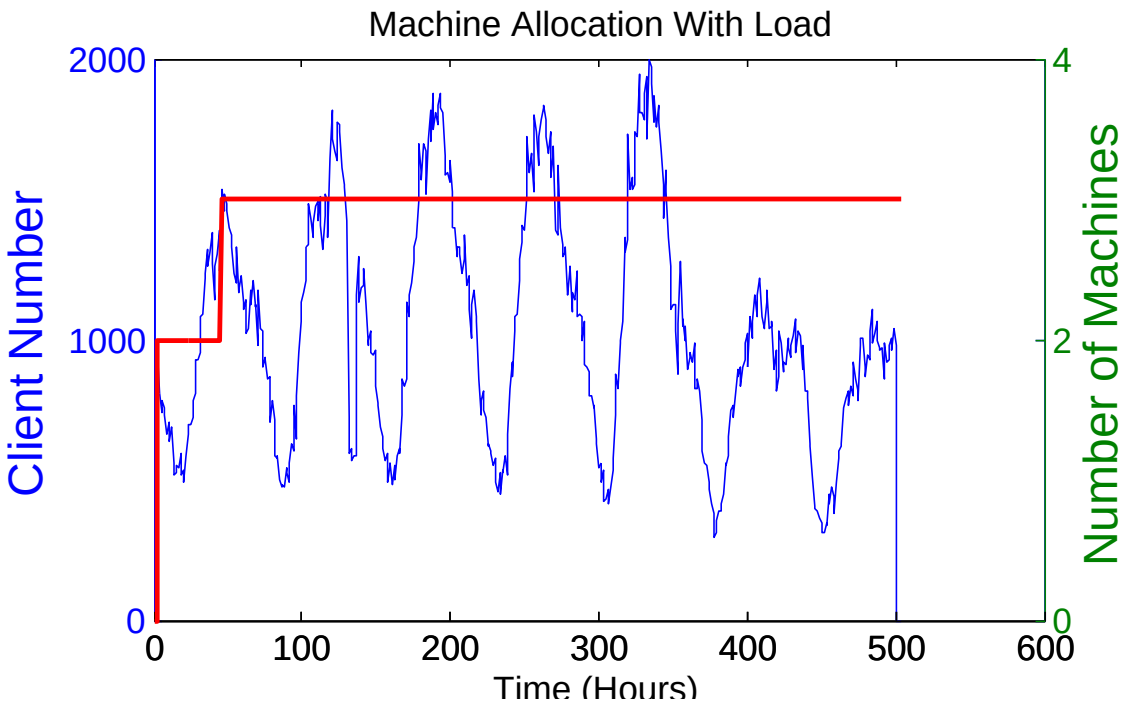


Figure 78: Resource Allocation for Medium SLA violation with high reconfiguration cost

X.5.2.5 Low SLA violation cost

Here the effect of reconfiguration cost on a system with low SLA violation cost compared to machine cost is studied. The ratio of SLA violation cost to machine cost considered is 1:5 i.e. the cost of machines is 5 times the penalty of violating SLA. Thus the concentration will be more on using lesser machine. The configuration 1_5_15 has very high reconfiguration cost (3 times machine cost and 15 times SLA violation cost). Thus the tendency is to resist changes. The resource allocation is seen in Figure 82. It can be seen that the number of machines remain at 2 throughout even though there are high workload coming in. Only when the workload reaches very high peak of 2000, the 3_{rd} machines get allocated. Due to the cost of the machine being 5 times higher than SLA violation, 4 machines are not allocated which would ensure no SLA is violated. Another point to be noted is that once it reaches 3, the machine is not released even though the workload decreases since the cost of reconfiguration is much higher than machine cost.

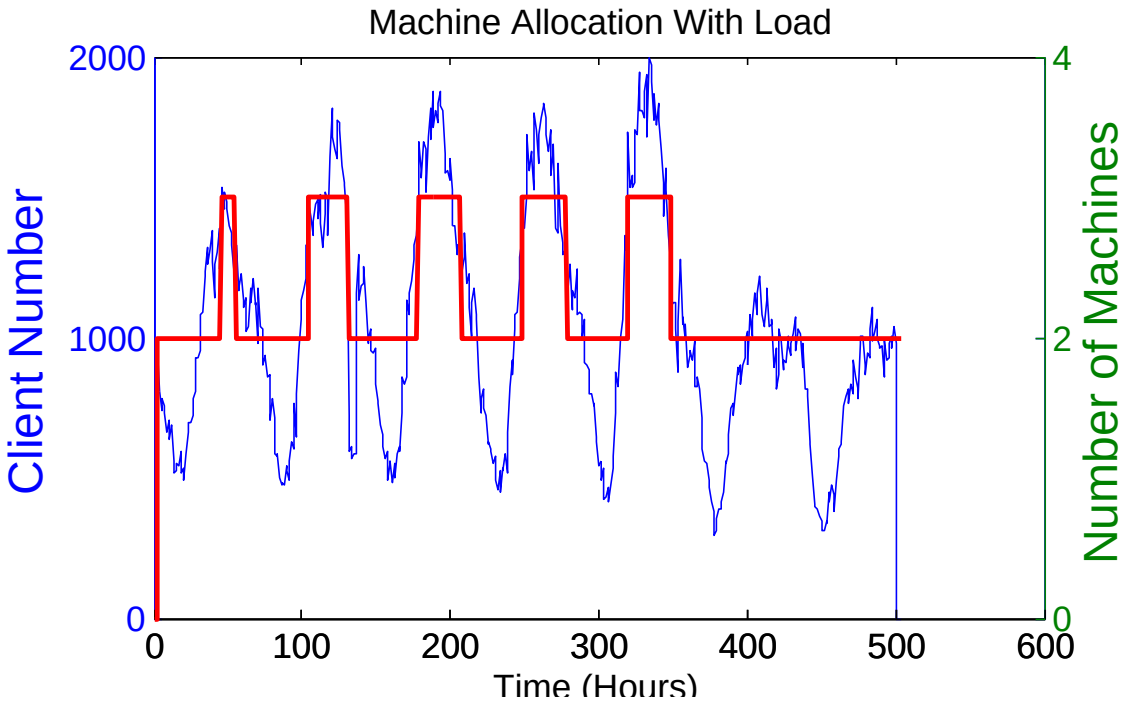


Figure 79: Low SLA violation, Medium Machine Cost and low reconfiguration cost

The other configurations with lesser reconfiguration costs behave similarly with subtle differences among each other. For example, configuration 1_5_1 and 1_5_4(Figures 79 and 80) are very similar except that due to higher reconfiguration costs, the latter configuration tends to delay the release of the 3^{rd} machine. Configuration 1_5_10, shown in Figure 81 delays the allocation of the 3^{rd} machine until the 150_{th} hour mark compared to hour 60 in 1_5_1 and 1_5_4. All of the above configurations also do not use 4 machines at all and endure some amount of SLA violations. This is understandable since the machine cost and reconfiguration cost are much higher.

X.6 Conclusion

This chapter presented challenges in a modern day data center scenario. The main challenge is in proper utilization of resource in the face of SLA driven performance assurance. Recent technologies such as virtualization and cloud computing bring about promises to

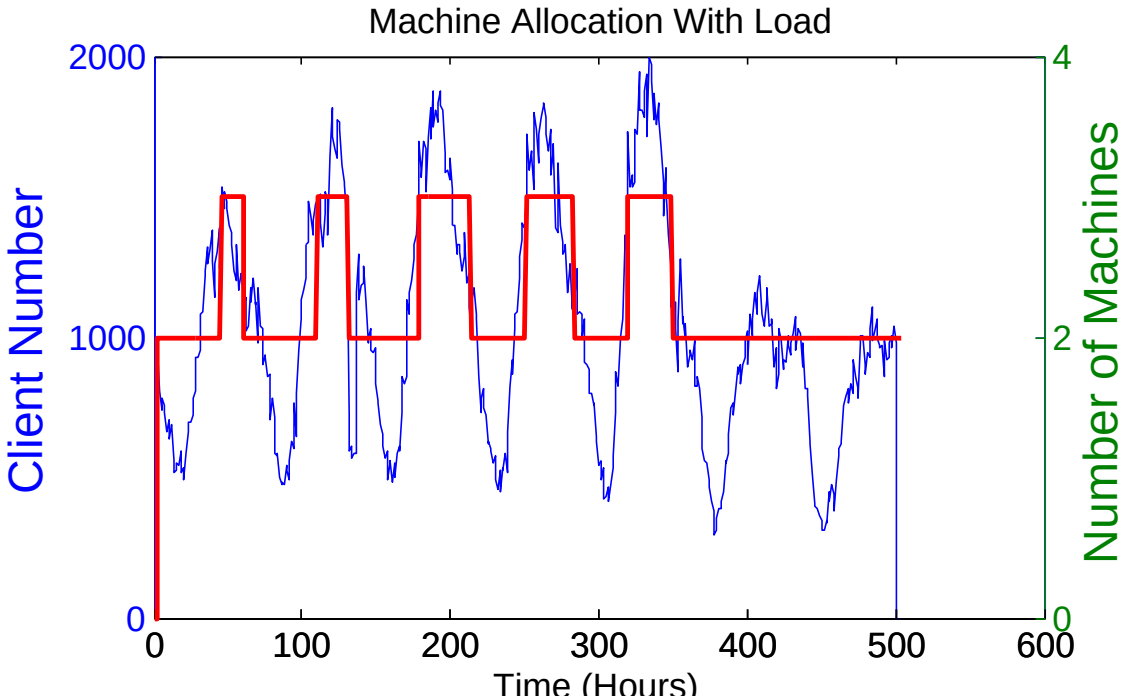


Figure 80: Low SLA violation, Medium Machine Cost and medium reconfiguration cost

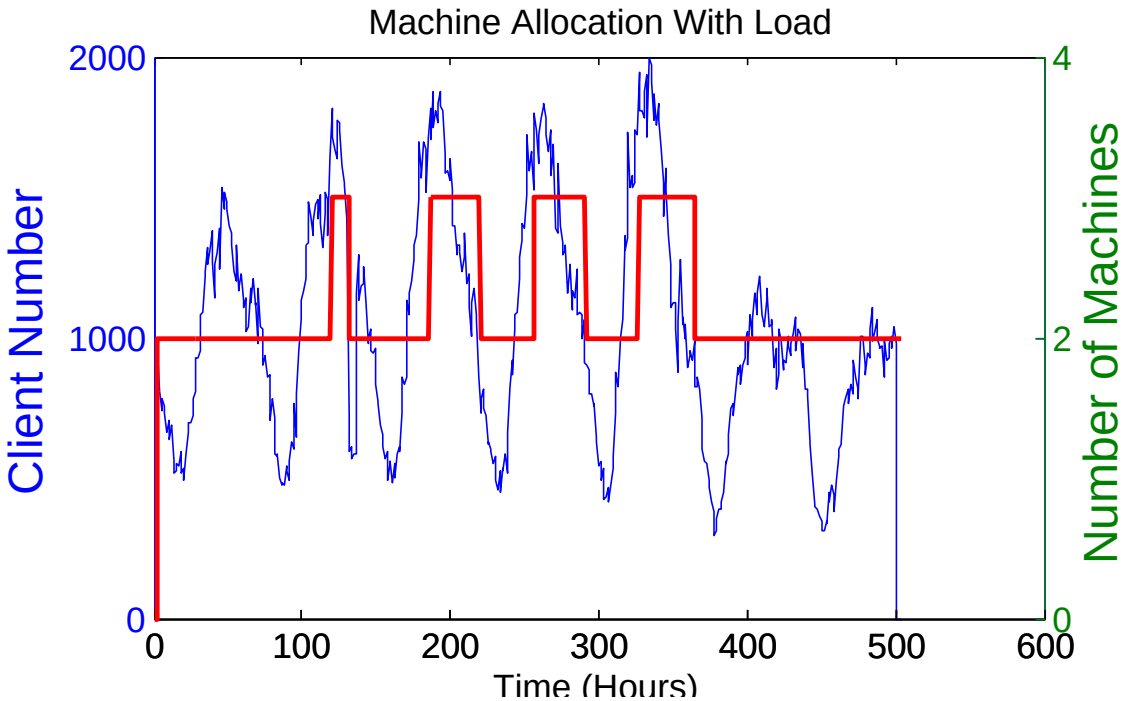


Figure 81: Low SLA violation, Medium Machine Cost and high reconfiguration cost

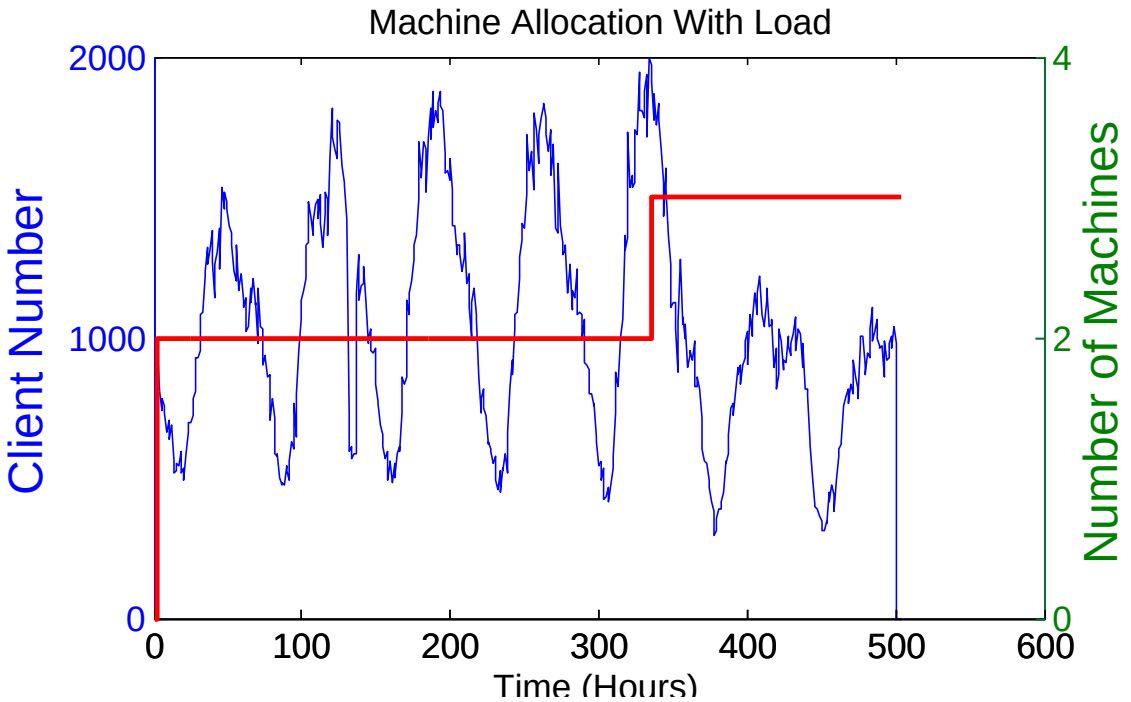


Figure 82: Low SLA violation, Medium Machine Cost and very high reconfiguration cost

solve some of the challenges. A look ahead resource allocation algorithm has been proposed in this thesis. This algorithm predicts the incoming workload for certain time steps in the future and allocates resources based on the optimizing a cost function. It can be seen that the resource allocation nicely follows the workload pattern. It has also been shown that the resource allocation trend varies depending upon the nature of the cost function.

CHAPTER XI

CONCLUDING REMARKS

This thesis works towards a solution for the problem of QoS assurance in large scale enterprise distributed component based applications. Such application form the core of many different types of systems like distributed real time and embedded systems such as shipboard computing or large scale shared data centers which acts as host to multiple applications or proprietary multi-tiered internet applications. A common objective across all such systems is to maximize the utility of the system. The utility of the system is proportional to the throughput of the application or to the amount of clients handled. The cost of the system is expressed by the procurement cost of the resources/machines and the power cost of running them. In this thesis, a static allocation strategy has been proposed which aims at placing the different components of the application in an intelligent way such that the total resources are minimized. The strategy also takes care that the response time or Quality of Service is always bounded by SLA limit. The solution proposed consists of a three-level strategy wherein component resource requirement is profiled in the first step. In the second step an accurate model of the application is prepared. In the third step a heuristic algorithm is used which uses the model and the resource profiles to come up with a component placement strategy that will ensure the use of lesser resource.

In the end a look ahead algorithm for resource allocation in modern day data center planning has been proposed. This algorithm can be used in conjunction with modern day technologies such as cloud computing to provide optimized resource allocation. The actual nature of resource allocation will depend upon a cost function which will give priority to the different aspects of the application such as SLA violation, machine cost, reconfiguration cost etc. This approach predicts the incoming load for the next time intervals and depending upon that it searches for the best configuration in the subsequent intervals which

Table 27: Summary Of Research Contributions

Category	Contributions
Component Resource Requirement Identification	TargetManager: design and implementation of (1) distributed profiling framework, (2) implementation of profiling techniques for component resource profiling, and (3) customer behavior modeling for overall component resource requirement
Performance Estimation of Software Components	(1) Queuing theoretic models for large scale multi-tiered internet applications, (2) more accurate analytic models for high utilization, software contention and multiple processors/cores, (3) simulation modeling of multi-threaded application with software contentions and (4) markov chain modeling of soft real time systems
Application Component Placement	(1) Detailed comparative study of muliple bin-packing heuristics, (2) design and development of component placement heuristic based on worst-fit bin packing and (3) development of component replication and placement heuristic based on worst fit bin packing.

will minimize the cost for the application. The configuration for the first interval is then used and applied and the rest are ignored. In the next interval, again the search is made and the best solution is chosen.

Table 28: Summary of Publications

Category	Publications
Profile Driven Identification of Component Resource Requirement	1. Bulls-Eye: A Resource Provisioning Service for Enterprise Distributed Real-time and Embedded Systems , Proceedings of the International Symposium on Distributed Objects and Applications (DOA), Montpellier, France, Oct 30th - Nov 1st, 2006. 2. Dynamic Analysis and Profiling of Multi-threaded Systems, Designing Software-Intensive Systems: Methods and Principles, Edited by Dr. Pierre F. Tiako, Langston University, OK, April, 2008. 3. Design and Performance Evaluation of an Adaptive Resource Management Framework for Distributed Real-time and Embedded Systems, EURASIP Journal on Embedded Systems (EURASIP JES): Special issue on Operating System Support for Embedded Real-Time Applications, Edited by Alfons Crespo, Ismael Ripoll, Michael Gonzalez Harbour, and Giuseppe Lipari, 2008, pgs. 47-66.
Performance Estimation of Component Based Software Applications and application placement	4. A Component Assignment Framework for Improved Capacity and Assured Performance in Web Portals, Proceedings of the 11th International Symposium on Distributed Objects, Middleware, and Applications (DOA'09) Vilamoura, Algarve-Portugal, Nov 01 - 03, 2009. 5. Modeling Software Contention using Colored Petri Nets, Proceedings of the 16th Annual Meeting of the IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS), September 8-10, Baltimore, MD. 6. The Impact of Variability on Soft Real-Time System Scheduling, Proceedings of the 15th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA 2009), Beijing, China, August 24-26, 2009. 7. A Capacity Planning Process for Performance Assurance of Component-Based Distributed Systems, Proceedings of the International Conference on Performance Engineering to be held in March 2011, Karlsruhe, Germany. 8 Impediments to Analytical Modeling of Multi-Tiered Web Applications, Poster paper in the 18th Annual Meeting of the IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS), August 17-19, Miami, FL. 9. Model-Driven Performance Evaluation of Web Application Portals, Model-Driven Domain Analysis and Software Development: Architectures and Functions, a book edited by Janis Osis and Erika Asnina, In submission. 10. Toward Effective Multi-capacity Resource Allocation in Distributed Real-time and Embedded Systems, Proceedings of the 11th IEEE International Symposium on Object/Component/Service-oriented Real-time Distributed Computing (ISORC 08), Orlando, Florida, May 5-7, 2008.

REFERENCES

[1] S. Abdelwahed, Jia Bai, Rong Su, and N. Kandasamy. On the application of predictive control techniques for adaptive performance management of computing systems. *Network and Service Management, IEEE Transactions on*, 6(4):212 –225, dec. 2009.

[2] Sherif Abdelwahed, Nagarajan Kandasamy, and Sandeep Neema. A control-based framework for self-managing distributed computing systems. In *WOSS '04: Proceedings of the 1st ACM SIGSOFT workshop on Self-managed systems*, pages 3–7, New York, NY, USA, 2004. ACM.

[3] C. Amza, A. Ch, A.L. Cox, S. Elnikety, R. Gil, K. Rajamani, and W. Zwaenepoel. Specification and Implementation of Dynamic Web Site Benchmarks. In *5th IEEE Workshop on Workload Characterization*, pages 3–13, 2002.

[4] Ronald E. Barkley and T. Paul Lee. A Heap-based Callout Implementation to Meet Real-time Needs. In *Proceedings of the USENIX Summer Conference*, pages 213–222. USENIX Association, June 1988.

[5] W. Binder and J. Hulaas. A portable CPU-management framework for Java. *IEEE Internet Computing*, pages 74–83, 2004.

[6] W. Binder, J.G. Hulaas, and A. Villazón. Portable resource control in Java. In *Proceedings of the 16th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 139–155. ACM New York, NY, USA, 2001.

[7] Walter Binder. Portable, Efficient, and Accurate Sampling Profiling for Java-based Middleware. In *Proceedings of the 5^{th} International Workshop on Software Engineering and Middleware (SEM '05)*, pages 46–53, 2005.

[8] J. Bonér. AspectwerkzŮdynamic AOP for java. In *Invited talk at 3rd International Conference on Aspect-Oriented Software Development (AOSD)*. Citeseer, 2004.

[9] M. Broberg, L. Lundberg, and H. Grahn. Visualization and performance prediction of multithreaded Solaris programs by tracing kernel threads. In *Proceedings of the 13th International Parallel Processing Symposium*, pages 407–413, 1999.

[10] G. Brose. Jacorb: Implementation and design of a java orb. In *Distributed applications and interoperable systems: IFIP TC6 WG6. 1 International Working Conference on Distributed Applications and Interoperable Systems (DAIS'97), 30th September-2nd October 1997, Cottbus, Germany*, page 143. Chapman & Hall, 1997.

[11] A. Budhiraja and A.P. Ghosh. A large deviations approach to asymptotically optimal control of crisscross network in heavy traffic. *Annals of Applied Probability*, 15(3):1887–1935, 2005.

[12] BM Cantrill, TW Doeppner Jr, S.S. Inc, and M. View. Threadmon: a tool for monitoring multithreaded program performance. In *System Sciences, 1997, Proceedings of the Thirtieth Hawaii International Conference on*, volume 1, 1997.

[13] Michael Cardosa, Madhukar R. Korupolu, and Aameek Singh. Shares and utilities based power consolidation in virtualized server environments. In *IM'09: Proceedings of the 11th IFIP/IEEE international conference on Symposium on Integrated Network Management*, pages 327–334, Piscataway, NJ, USA, 2009. IEEE Press.

[14] D. Carrera, M. Steinder, I. Whalley, J. Torres, and E. Ayguade. Utility-based placement of dynamic web applications with fairness goals. In *Network Operations and Management Symposium, 2008. NOMS 2008. IEEE*, pages 9–16, April 2008.

[15] EG Coffman Jr, MR Garey, and DS Johnson. Approximation algorithms for bin packing: a survey. 1996.

[16] S. Cunha, Jussara M. Almeida, Virgilio Almeida, and Marcos Santos. Self-adaptive capacity management for multi-tier virtualized environments. In *Integrated Network Management*, pages 129–138, 2007.

[17] J. Davies, N. Huismans, R. Slaney, S. Whiting, M. Webster, and R. Berry. An aspect oriented performance analysis environment. In *International Conference on Aspect Oriented Software Development*. Citeseer, 2003.

[18] José Luis Díaz, Daniel F. García, Kanghee Kim, Chang-Gun Lee, Lucia Lo Bello, José María López, Sang Lyul Min, and Orazio Mirabella. Stochastic analysis of periodic real-time systems. In *RTSS '02: Proceedings of the 23rd IEEE Real-Time Systems Symposium (RTSS'02)*, page 289, Washington, DC, USA, 2002. IEEE Computer Society.

[19] M. Dmitriev. Application of the hotswap technology to advanced profiling. In *Proceedings of the First Workshop on Unanticipated Software Evolution, held at ECOOP 2002 International Conference*. Citeseer.

[20] M. Dmitriev. Safe Class and Data Evolution in Large and Long-Lived Java [tm] Applications, Sun Microsystems. *Inc., Mountain View, CA*, 2001.

[21] M. Dmitriev. Profiling Java applications using code hotswapping and dynamic call graph revelation. In *Proceedings of the 4th international workshop on Software and performance*, pages 139–150. ACM New York, NY, USA, 2004.

[22] W.W. Eckerson et al. Three Tier Client/Server Architecture: Achieving Scalability, Performance and Efficiency in Client Server Applications. *Open Information Systems*, 10(1), 1995.

[23] D. Schmidt *et al*. *Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects, Volume 2*. Wiley & Sons, New York, 2000.

[24] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, MA, 1995.

[25] J. Gosling. Java intermediate bytecodes. *ACM Sigplan Notices*, 30(3):111–118, 1993.

[26] Waheed Iqbal, Matthew Dailey, and David Carrera. Sla-driven adaptive resource management for web applications on a heterogeneous compute cloud. In Martin Jaatun, Gansen Zhao, and Chunming Rong, editors, *Cloud Computing*, volume 5931 of *Lecture Notes in Computer Science*, pages 243–253. Springer Berlin / Heidelberg, 2009.

[27] K. Jensen. *Coloured Petri nets: basic concepts, analysis methods and practical use: volume 1*. Springer-Verlag London, UK, 1996.

[28] Kurt Jensen, Lars Michael Kristensen, and Lisa Wells. Coloured petri nets and cpn tools for modelling and validation of concurrent systems. *Int. J. Softw. Tools Technol. Transf.*, 9(3):213–254, 2007.

[29] Yang Jie, Qiu Jie, and Li Ying. A profile-based approach to just-in-time scalability for cloud applications. pages 9 –16, sep. 2009.

[30] A. Karve, T. Kimbrel, G. Pacifici, M. Spreitzer, M. Steinder, M. Sviridenko, and A. Tantawi. Dynamic placement for clustered web applications. In *Proceedings of the 15th international conference on World Wide Web*, pages 595–604. ACM New York, NY, USA, 2006.

[31] Krishna M. Kavi, Alireza Moshtaghi, and Deng-Jyi Chen. Modeling multithreaded applications using petri nets. *Int. J. Parallel Program.*, 30(5):353–371, 2002.

[32] G. Khanna, K. Beaty, G. Kar, and A. Kochut. Application performance management

in virtualized server environments. In *10th IEEE/IFIP Network Operations and Management Symposium, 2006 (NOMS 2006)*, pages 373–381, 2006.

[33] Kanghee Kim, Jose Luis Diaz, and Jose Maria Lopez. An exact stochastic analysis of priority-driven periodic real-time systems and its approximations. *IEEE Trans. Comput.*, 54(11):1460–1466, 2005. Member-Lucia Lo Bello and Member-Chang-Gun Lee and Member-Sang Lyul Min.

[34] T. Kimbrel, M. Steinder, M. Sviridenko, and A. Tantawi. Dynamic Application Placement Under Service and Memory Constraints. In *Experimental And Efficient Algorithms: 4th International Workshop, WEA 2005, Santorini Island, Greece, May 10-13, 2005: Proceedings*, page 391. Springer, 2005.

[35] Fabio Kon, Tomonori Yamane, Christopher K. Hess, Roy H. Campbell, and M. Dennis Mickunas. Dynamic resource management and automatic configuration of distributed component systems. In *COOTS'01: Proceedings of the 6th conference on USENIX Conference on Object-Oriented Technologies and Systems*, pages 2–2, Berkeley, CA, USA, 2001. USENIX Association.

[36] Samuel Kounev. Performance Modeling and Evaluation of Distributed Component-Based Systems Using Queueing Petri Nets. *IEEE Transactions on Software Engineering*, 32(7):486–502, 2006.

[37] W.T.C. Kramer and C. Ryan. Performance variability of highly parallel architectures. *Proceedings of the International Conference on Computational Science (ICCS 2003*, 2659:560–569.

[38] S. Kumar and PR Kumar. Closed Queueing Networks in Heavy Traffic: Fluid Limits and Efficiency. *Stochastic networks: stability and rare events*, page 41, 1996.

[39] G. Latouche and V. Ramaswami. *Introduction to Matrix Analytic Methods in Stochastic Modeling*. Society for Industrial Mathematics, 1999.

[40] J. Li and H.P.L.P. Alto. Monitoring of component-based systems. *HP Tech Reports*, 2003.

[41] Sorin Manolache, Petru Eles, and Zebo Peng. Schedulability analysis of applications with stochastic task execution times. *Trans. on Embedded Computing Sys.*, 3(4):706–735, 2004.

[42] Rajat Mehrotra, Abhishek Dubey, Sherif Abdelwahed, and Asser Tantawi. Integrated monitoring and control for performance management of distributed enterprise systems. *Modeling, Analysis, and Simulation of Computer Systems, International Symposium on*, 0:424–426, 2010.

[43] D.A. Menasce and M.N. Bennani. Autonomic virtualized environments. In *Proceedings of the International Conference on Autonomic and Autonomous Systems*, volume 28. Citeseer, 2006.

[44] Daniel A Menasce and V. A. F. Almeida. *Capacity Planning for Web Services*. Prentice Hall, Upper Saddle, NJ, 2002.

[45] Daniel A. Menascé and Mohamed N. Bennani. Analytic performance models for single class and multiple class multithreaded software servers. In *Int. CMG Conference*, pages 475–482. Computer Measurement Group, 2006.

[46] Daniel A. Menasce, Lawrence W. Dowdy, and Virgilio A. F. Almeida. *Performance by Design: Computer Capacity Planning By Example*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2004.

[47] A. Michael, F. Armando, G. Rean, DJ Anthony, K. Randy, K. Andy, L. Gunho,

P. David, R. Ariel, S. Ion, et al. Above the clouds: A berkeley view of cloud computing. *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2009-28*, 2009.

[48] Robin Milner, Mads Tofte, and Robert Harper. *The definition of Standard ML.* MIT Press, Cambridge, MA, USA, 1990.

[49] Rafael Moreno-Vozmediano, Ruben S. Montero, and Ignacio M. Llorente. Elastic management of cluster-based services in the cloud. In *ACDC '09: Proceedings of the 1st workshop on Automated control for datacenters and clouds*, pages 19–24, New York, NY, USA, 2009. ACM.

[50] J. Murayama. Performance profiling using TNF. *Sun Developer Network, July 2001*, 2001.

[51] Object Management Group. *Deployment and Configuration Adopted Submission*, OMG Document mars/03-05-08 edition, July 2003.

[52] G. Pacifici, W. Segmuller, M. Spreitzer, M. Steinder, A. Tantawi, and A. Youssef. Managing the response time for multi-tiered web applications. *IBM TJ Watson Research Center, Yorktown, NY, Tech. Rep. RC23651*, 2005.

[53] P. Padala, K.G. Shin, X. Zhu, M. Uysal, Z. Wang, S. Singhal, A. Merchant, and K. Salem. Adaptive control of virtualized resources in utility computing environments. *ACM SIGOPS Operating Systems Review*, 41(3):302, 2007.

[54] E.A. Pek

"oz and J. Blanchet. Heavy Traffic Limits Via Brownian Embeddings. *Probability in the Engineering and Informational Sciences*, 20(04):595–598, 2006.

[55] Fabrizio Petrini, Darren J. Kerbyson, and Scott Pakin. The case of the missing supercomputer performance: Achieving optimal performance on the 8,192 processors of asci q. In *SC '03: Proceedings of the 2003 ACM/IEEE conference on Supercomputing*, page 55, Washington, DC, USA, 2003. IEEE Computer Society.

[56] M.L. Puterman. *Markov decision processes: discrete stochastic dynamic programming*. John Wiley & Sons, Inc. New York, NY, USA, 1994.

[57] S.P. Reiss. Visualizing Java in action. In *Proceedings of the 2003 ACM symposium on Software visualization*. ACM, 2003.

[58] S.P. Reiss. Efficient monitoring and display of thread state in Java. *IWPC 2005*, pages 247–256, 2005.

[59] Christian P. Robert and George Casella. *Monte Carlo Statistical Methods (Springer Texts in Statistics)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2005.

[60] Nilabja Roy, Nathan Hamm, Manish Madhukar, Douglas C. Schmidt, and Larry Dowdy. A Uniform Modeling Methodology for Soft Real-time Systems. Technical Report ISIS-09-104, Institute For Software Integrated Systems, Vanderbilt University, April 2009.

[61] Nilabja Roy, Yuan Xue, Aniruddha Gokhale, Larry Dowdy, and Douglas C. Schmidt. A Component Assignment Framework for Improved Capacity and Assured Performance in Web Portals. In *Proceedings of the 11th International Symposium on Distributed Objects, Middleware, and Applications (DOA'09)*, pages 671–689, November 2009.

[62] Prasan Roy, S. Seshadri, S. Sudarshan, and Siddhesh Bhobe. Efficient and extensible algorithms for multi query optimization. *SIGMOD Rec.*, 29(2):249–260, 2000.

[63] A.A. Safaeei, M. Kamali, MS Haghjoo, and K. Izadi. Caching Intermediate Results for Multiple-Query Optimization. In *IEEE/ACS International Conference on Computer Systems and Applications, 2007. AICCSA'07*, pages 412–415, 2007.

[64] Douglas C. Schmidt, Bala Natarajan, Aniruddha Gokhale, Nanbor Wang, and Christopher Gill. TAO: A Pattern-Oriented Object Request Broker for Distributed Real-time and Embedded Systems. *IEEE Distributed Systems Online*, 3(2), February 2002.

[65] Douglas C. Schmidt, Michael Stal, Hans Rohnert, and Frank Buschmann. *Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects, Volume 2*. Wiley & Sons, New York, 2000.

[66] A. Schrijver. *Theory of linear and integer programming*. Wiley, 1986.

[67] C. Smith and D. Henry. High-performance Linux cluster monitoring using Java. In *Proceedings of the 3rd Linux Cluster International Conference*, 2002.

[68] R. Smith, IH Osman, CR Reeves, and GD Smith. Modern heuristic search methods. *Ed. John Wiley & Sons Ltd*, 1996.

[69] Anand Srivastav and Peter Stangier. Tight approximations for resource constrained scheduling and bin packing. In *Proceedings of the 4th Twente Workshop on Graphs and Combinatorial Optimization*, pages 223–245, New York, NY, USA, 1997. Elsevier North-Holland, Inc.

[70] C. Stewart and K. Shen. Performance modeling and system management for multi-component online services. In *Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation-Volume 2 table of contents*, pages 71–84. USENIX Association Berkeley, CA, USA, 2005.

[71] K. Subramaniam and M. Thazhuthaveetil. Effectiveness of sampling based software

profilers. In *1st International Conference on Reliability and Quality Assurance*, pages 1–5. Citeseer, 1994.

[72] R. Suri, S. Sahu, and M. Vernon. Approximate Mean Value Analysis for Closed Queuing Networks with Multiple-Server Stations. In *Proceedings of the 2007 Industrial Engineering Research Conference*. Citeseer, 2007.

[73] H. Sutter. The free lunch is over: A fundamental turn towards concurrency in software. *Dr. Dobbs Journal*, 30(3), 2005.

[74] B. Urgaonkar, A.L. Rosenberg, P. Shenoy, and A. Zomaya. Application Placement on a Cluster of Servers. *International Journal of Foundations of Computer Science*, 18(5):1023–1041, 2007.

[75] B. Urgaonkar, P. Shenoy, A. Chandra, and P. Goyal. Dynamic provisioning of multi-tier internet applications. In *Autonomic Computing, 2005. ICAC 2005. Proceedings. Second International Conference on*, pages 217–228, 2005.

[76] B. Urgaonkar, P. Shenoy, and T. Roscoe. Resource overbooking and application profiling in a shared Internet hosting platform. *ACM Transactions on Internet Technologies (TOIT)*, 9(1):1–45, 2009.

[77] Bhuvan Urgaonkar, Giovanni Pacifici, Prashant Shenoy, Mike Spreitzer, and Asser Tantawi. An Analytical Model for Multi-tier Internet Services and its Applications. *SIGMETRICS Perform. Eval. Rev.*, 33(1):291–302, 2005.

[78] Bhuvan Urgaonkar, Prashant Shenoy, Abhishek Chandra, Pawan Goyal, and Timothy Wood. Agile dynamic provisioning of multi-tier internet applications. *ACM Trans. Auton. Adapt. Syst.*, 3(1):1–39, 2008.

[79] Akshat Verma, Puneet Ahuja, and Anindya Neogi. pmapper: power and migration

cost aware application placement in virtualized systems. In *Middleware '08: Proceedings of the 9th ACM/IFIP/USENIX International Conference on Middleware*, pages 243–264, New York, NY, USA, 2008. Springer-Verlag New York, Inc.

[80] Yefu Wang, Xiaorui Wang, Ming Chen, and Xiaoyun Zhu. Power-efficient response time guarantees for virtualized enterprise servers. In *RTSS '08: Proceedings of the 2008 Real-Time Systems Symposium*, pages 303–312, Washington, DC, USA, 2008. IEEE Computer Society.

[81] Lisa Wells, S"ren Christensen, Lars M. Kristensen, and Kjeld H. Mortensen. Simulation based performance analysis of web servers. In *PNPM '01: Proceedings of the 9th international Workshop on Petri Nets and Performance Models (PNPM'01)*, page 59, Washington, DC, USA, 2001. IEEE Computer Society.

[82] Timothy Wood, Prashant J. Shenoy, Arun Venkataramani, and Mazin S. Yousif. Black-box and gray-box strategies for virtual machine migration. In *NSDI*, 2007.

[83] Q. Zhang, L. Cherkasova, and E. Smirni. A regression-based analytic model for dynamic resource provisioning of multi-tier applications. In *Proceedings of the ICAC*, volume 7. Citeseer, 2007.

[84] Qi Zhang, Ludmila Cherkasova, Guy Mathews, Wayne Greene, and Evgenia Smirni. R-capriccio: a capacity planning and anomaly detection tool for enterprise services with live workloads. In *Middleware '07: Proceedings of the ACM/IFIP/USENIX 2007 International Conference on Middleware*, pages 244–265, New York, NY, USA, 2007. Springer-Verlag New York, Inc.

[85] Qi Zhang, Ludmila Cherkasova, Ningfang Mi, and Evgenia Smirni. A regression-based analytic model for capacity planning of multi-tier applications. *Cluster Computing*, 11(3):197–211, 2008.