Rule-based Automatic Software Performance Diagnosis and Design Improvement

by

JING XU

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

Doctor of Philosophy

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

September 16, 2008

© Copyright 2008, Jing Xu

Library and
Archives Canada

Bibliothèque et
Archives Canada

Published Heritage
Branch

Direction du
Patrimoine de l'édition

395 Wellington Street
Ottawa ON K1A 0N4
Canada

395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence

ISBN: 978-0-494-43917-3

Our file Notre référence

ISBN: 978-0-494-43917-3

NOTICE:

The author has granted a non-exclusive license allowing Library and Archives Canada to reproduce, publish, archive, preserve, conserve, communicate to the public by telecommunication or on the Internet, loan, distribute and sell theses worldwide, for commercial or non-commercial purposes, in microform, paper, electronic and/or any other formats.

The author retains copyright ownership and moral rights in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

AVIS:

L'auteur a accordé une licence non exclusive permettant à la Bibliothèque et Archives Canada de reproduire, publier, archiver, sauvegarder, conserver, transmettre au public par télécommunication ou par l'Internet, prêter, distribuer et vendre des thèses partout dans le monde, à des fins commerciales ou autres, sur support microforme, papier, électronique et/ou autres formats.

L'auteur conserve la propriété du droit d'auteur et des droits moraux qui protègent cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

In compliance with the Canadian Privacy Act some supporting forms may have been removed from this thesis.

Conformément à la loi canadienne sur la protection de la vie privée, quelques formulaires secondaires ont été enlevés de cette thèse.

While these forms may be included in the document page count, their removal does not represent any loss of content from the thesis.

Bien que ces formulaires aient inclus dans la pagination, il n'y aura aucun contenu manquant.

Canada

ABSTRACT

Performance of a software system is the result of many interacting factors, which may require complicated and time-consuming tradeoffs. This thesis describes a rule-based framework, Performance Booster (PB), to identify root causes of performance limits, to untangle the effects of the system configuration (such as favorable thread pool size) from limits imposed by the software design, and to recommend both configuration and design improvements. The framework uses a performance model which represents (and is derived from) a UML design model, and applies transformations to the given performance model to obtain another improved one. The improvements imply configuration and design changes which can be applied to the system. This thesis describes the process of deriving rules from performance expert knowledge, the strategies for the PB framework, and demonstrates feasibility by applying a set of rules to the design of four systems.

ACKNOWLEDGMENTS

This thesis is dedicated to my dear husband and my lovely daughter.

My greatest appreciation goes to my parents, my parent-in-laws, my supervisor Professor Murray Woodside, my good friend Professor Dorina Petriu, my lab colleagues and friends, and all the committee members.

TABLE OF CONTENTS

Chapter 1	Introduction.....	1
1.1	Motivation and Objectives.....	1
1.2	The Approach of the Research.....	3
1.3	Thesis Outline	5
1.4	Contributions.....	5
Chapter 2	Survey on Performance Diagnosis and Mitigation	7
2.1	Performance Models	7
2.2	Performance Diagnosis	9
2.2.1	Existing Tools for Performance Diagnosis Based On Measurements	10
2.2.2	Mechanisms for Bottleneck Identification.....	11
2.3	Mitigation for Performance Problems	12
2.3.1	Configuration Improvement.....	13
2.3.2	Methods for Design Improvement.....	14
2.4	Principles from Connie Smith.....	16
2.5	Performance Expert Systems	20
Chapter 3	Background Knowledge.....	22

3.1	PUMA Project.....	22
3.2	UML and Its Extension Profiles	24
3.2.1	Use Cases	25
3.2.2	Classes and Components.....	26
3.2.3	Deployments	27
3.2.4	Behaviors	28
3.2.5	UML Extension for Annotating Performance Properties	32
3.3	Layered Queueing Network (LQN).....	35
3.3.1	Model Elements and Parameters.....	35
3.3.2	Estimates	38
3.3.3	LQN Solvers	39
3.4	Rule-based Systems and the Jess Rule Engine	40
3.4.1	Rules and Rule-based Systems	41
3.4.2	Architecture of a Rule-based System.....	42
3.4.3	Developing a Rule-based System	43
3.4.4	The Jess Rule Engine and the Jess Language	45
Chapter 4	An Example in Performance Diagnosis and Design Improvement	49
4.1	Design Model and Performance Model for the Building Security System	49
4.2	Performance Diagnosis and Improvement by Human Expert	55

4.2.1	Performance Estimation for Base Design	55
4.2.2	Bottleneck Mitigation by Configurational Change.....	56
4.2.3	Reduce Length of Execution Path by Design Change	58
4.2.4	Performance Improvement and Feedback to Design	61
4.3	What Do We Learn From the Example?.....	62
4.3.1	Two Causes of Performance Problems	63
4.3.2	A Tower Pattern for Root Bottleneck Identification.....	64
4.3.3	Iterative Improvement Process and a Holistic Improvement Principle	65
4.3.4	Some Mitigation Strategies.....	66
4.3.5	What Can Performance Diagnosis Tell.....	67
4.3.6	Potential for Rules.....	68
Chapter 5	Formalization of Performance Expert Knowledge	69
5.1	Understand Performance Requirements	69
5.2	Resource Interaction Model.....	72
5.3	Localization of Problems	74
5.3.1	Bottleneck Identification and Root Bottleneck.....	75
5.3.2	Long Path Identification	79
5.4	Mitigation Techniques for Performance Problems	79
5.4.1	Bottleneck Mitigation	80

5.4.2	Long Path Mitigation	84
5.5	Configuration Changes vs. Design Changes.....	86
5.6	Performance Model Surrogates for Design Properties	88
Chapter 6	Strategies for the Performance Booster (PB) Framework	92
6.1	Overview of PB Search Strategy	92
6.2	PB Framework Architecture	93
6.3	Evaluation of a Design by Its Best Configuration Found by PB	95
6.4	Search for the Best Configuration.....	98
6.4.1	Bottleneck Identification Rules.....	99
6.4.2	Configuration Mitigation	101
6.5	Generate Design Change Options.....	102
6.5.1	Long Path Identification Rules	102
6.5.2	Design Change Rule: <i>ShrinkExec</i>	103
6.5.3	Design Change Rule: <i>MoreAsync</i>	104
6.5.4	Design Change Rules for Bottlenecks	106
6.5.5	Ranking Design Change Options.....	108
6.6	Using Control Parameters to Customize Rules.....	109
6.7	Mapping Feedback to the UML Design.....	111
6.7.1	Resource Multiplicity Change	111

6.7.2	ShrinkExec Change.....	113
6.7.3	MoreAsync Change	114
6.7.4	Batching Change.....	116
6.7.5	Partitioning Change	117
6.7.6	Redeployment Change	119
Chapter 7	Implementation of the Performance Booster (PB).....	120
7.1	Implementation Overview	120
7.2	Virtual Profiling - Fast Path in Configuration Search	123
7.3	How to Extend the PB Rule Base	124
Chapter 8	CASE StudIES and Approach Validation.....	126
8.1	A Tutorial Example.....	126
8.1.1	Base Design of the Web Application (Cases 1 – 4).....	127
8.1.2	First Extended Design (<i>Partitioning</i> Mitigation – Case 5).....	132
8.1.3	Second Extended Design (<i>Batching</i> Mitigation – Case 6).....	135
8.2	BSS Example Revisited	137
8.3	300 Cases with Fixed Model Structure and Different Workload Parameters.	141
8.4	An Industrial Case Study: Image Exchange System.....	144
Chapter 9	Conclusions.....	149
9.1	Achievements.....	149

9.2	Limitations and Assumptions	151
9.3	Future Work	152
	References	154
	Appendix A PB Jess Rule Base	164
	Appendix B LQN models	177
	Building Security System (Base Case)	177
	Building Security System (PB version)	179
	Simple Web Application (Base Design)	182
	Complex Web Service System for 300 cases (Base Case with variables).....	183
	Image Exchange System	186
	Appendix C Diagnosis Log of Image Exchange System Generated by PB	188

List of Figures

Figure 1-1 Rule-based performance diagnosis and improvement	3
Figure 3-1 PUMA project	23
Figure 3-2 Example use case diagram	25
Figure 3-3 Example class	26
Figure 3-4 Example component	26
Figure 3-5 Example deployment	28
Figure 3-6 Example of an activity with input parameter	29
Figure 3-7 Example of state machine	29
Figure 3-8 Example of a sequence diagram	31
Figure 3-9 Example of a scenario annotated with SPT profile	34
Figure 3-10 Example of LQN model	37
Figure 3-11 Response time of forwarding call	39
Figure 3-12 Architecture of a typical rule-based system [Jess in Action]	43
Figure 3-13 Process of building a rule-based system	45
Figure 4-1 Deployment diagram for the BSS	50
Figure 4-2 Annotated sequence diagram for Access Control scenario	51
Figure 4-3 Annotated sequence diagram for Video Store/Acquire scenario	52

Figure 4-4 LQN model for the BSS	54
Figure 4-5 Second phase change on BSS LQN model	59
Figure 4-6 Modified sequence diagram for Acquire/Store scenario.....	62
Figure 4-7 Semantic network of performance problem root cause.....	63
Figure 4-8 Bottleneck tower pattern in BSS example	64
Figure 4-9 Iterative process of performance diagnosis and improvement.....	65
Figure 5-1 Performance requirements	70
Figure 5-2 Semantic network for performance requirements	71
Figure 5-3 Semantic network for resource interaction model	72
Figure 5-4 Semantic network for performance problem root causes.....	74
Figure 5-5 Bottleneck tower pattern	75
Figure 5-6 Multiple tower pattern with one root bottleneck.....	76
Figure 5-7 Multiple root bottlenecks	77
Figure 5-8 Performance problem mitigations	80
Figure 5-9 Configuration improvement sub-problem.....	86
Figure 5-10 Design change recommended by automatic diagnosis: <i>moreAsync</i> operation on DB1 for treating App1	91
Figure 6-1 PB - Rule-based knowledge system framework	94
Figure 6-2 Alternating configure and design change process.....	95
Figure 6-3 Tree structure of a diagnosis-mitigation progress.....	97

Figure 6-4 Process of searching for best configuration	99
Figure 6-5 Resource multiplicity in LQN and UML	111
Figure 6-6 Resource replication in LQN and UML.....	113
Figure 6-7 <i>ShrinkExec</i> in LQN and UML.....	114
Figure 6-8 <i>MoreAsync</i> in LQN and UML.....	115
Figure 6-9 Batching in LQN and UML	117
Figure 6-10 Partitioning in LQN and UML.....	118
Figure 6-11 Redeployment in LQN and UML	119
Figure 7-1 Class diagram of PB implementation.....	121
Figure 8-1 UML sequence diagram for a web application	126
Figure 8-2 Derived LQN model for the web application system.....	127
Figure 8-3 PB diagnosis tree (showing only the rounds).....	129
Figure 8-4 Performance result vs. cost weight for case 1 and 2	131
Figure 8-5 Performance vs. cost weight for case 3 and 4	131
Figure 8-6 Derived LQN model for extended design of the web application system with multiple entries on App2 (case 5)	133
Figure 8-7 Performance improvement for case with partitioning (case 5)	134
Figure 8-8 Improved LQN model for case 5	135
Figure 8-9 Derived LQN model for extended design of the web application system with task Net (case 6).....	136

Figure 8-10 Performance improvement for case with batching (case 6)	137
Figure 8-11 Batching change in the LQN model.....	137
Figure 8-12 PB diagnosis tree of BSS model	139
Figure 8-13 LQN model structure for random parameter study case	141
Figure 8-14 Number of steps taken for 300 cases.....	143
Figure 8-15 Improvement on throughputs of 300 cases	143
Figure 8-16 Sequence diagram of the image transfer scenario of IES	146
Figure 8-17 LQN model for the image transfer scenario of IES	147
Figure 8-18 PB diagnosis tree for IES	147

List of Rules

Rule 5-1 Bottleneck Existence Rule	77
Rule 5-2 Software Client-Server Rule	77
Rule 5-3 S-H Client-Server Rule	78
Rule 5-4 Client-Server Transitive Rule	78
Rule 5-5 Root Bottleneck Rule	78
Rule 5-6 Heavy Demand Rule	79
Rule 5-7 Bottleneck Multiplicity Rule.....	81
Rule 5-8 Processor Upgrade Rule.....	82
Rule 5-9 Software Bottleneck Bypass Rule.....	82
Rule 5-10 Redeploy Rule.....	82
Rule 5-11 Software Bottleneck Long Path Rule.....	83
Rule 5-12 Hardware Bottleneck Long Path Rule	83
Rule 5-13 Path Reduction Tryout Rule.....	85
Rule 5-14 Configuration Priority Rule	87
Rule 6-1 Hardware Bottleneck Identification Rule	100
Rule 6-2 Software Bottleneck Identification Rule.....	100
Rule 6-3 Configuration Mitigation Rule.....	101
Rule 6-4 Pure Long Path Rule	102

Rule 6-5 Soft Bottleneck Long Path Rule..... 103

Rule 6-6 Hard to Soft Bottleneck Rule 103

Rule 6-7 ShrinkExec Rule 104

Rule 6-8 MoreAsync Rule 105

Rule 6-9 Redeployment Rule..... 106

Rule 6-10 Partitioning Rule 107

Rule 6-11 Batching Rule..... 107

List of Scripts

Script 3-1 Red Light Rule	41
Script 3-2 Rules for Identifying Client-Server Relationship	44
Script 3-3 Root Bottleneck Identification Rule	44
Script 3-4 Bottleneck Identification Rule-Set.....	48

List of Tables

Table 4-1 Simulation results for BSS base design.....	55
Table 4-2 BSS LQN results for using multiple Buffers (40 cameras).....	57
Table 4-3 BSS LQN results for multi-threading StoreProc (40 cameras, 4 Buffers) .	57
Table 4-4 BSS LQN results for using multiple processors for ApplicationCPU	58
Table 4-5 BSS LQN results for with more concurrency case (100 cameras, 10 Buffers)	60
Table 4-6 Significant changes and performance results for BSS system	61
Table 5-1 Software Design Properties represented in the Performance Model.....	89
Table 8-1 Thread multiplicity constraints.....	128
Table 8-2 Performance analysis and improvement results	130
Table 8-3 Performance results for case with partitioning (case 5)	133
Table 8-4 Performance results for case with batching (case 6)	136
Table 8-5 Iterative generated values for four variables in 300 cases.....	142
Table 8-6 Diagnosis and improvement summary on the 300 cases.....	142
Table 8-7 Resource capacity limit for IES.....	145

List of Acronyms

BSS: Building Security System
CSM: Core Scenario Model
DA: Diagnosis Assistant, a component in PB
EG: Execution Graphs
EJB: Enterprise Java Bean
EQN: Extended Queueing Network
EXPERFORM: EXpert system for PERFORMance
GSPN: Generalized Stochastic Petri Nets
IE: Inference Engine
IES: Image Exchange System
LQN: Layered Queueing Network
LQNS: LQN Solver, the analytic LQN model solver
LQSIM: LQN simulation solver
MARTE: Modeling and Analysis of Real-Time and Embedded systems
MS: Model Solver
OMG: Object Management Group
PB: Performance Booster
PN: Petri Net
PUMA: Performance by Unified Model Analysis
QN: Queueing Network
QoS: Quality of Service
RB: Rule Base
SPA: System Performance Advisor
SPE: Software Performance Engineer
SPT: the UML profile for Schedulability, Performance and Time
UML: Unified Modeling Language

CHAPTER 1 INTRODUCTION

This chapter gives the motivation and objectives of this research, as well as an outline for the rest of the thesis.

1.1 Motivation and Objectives

Almost all software systems have performance requirements. Some are hard deadlines, as in safety-critical control systems, but soft requirements on the average response time, or a percentage of response delays, are more common (e.g., in telecommunications, and in service systems of all kinds). Performance, in the sense of response delay at a stated (possibly heavy) rate of requests, is a central feature of quality of service (QoS) specifications. The application domain for this work is any system describable by a Layered Queuing Networks (LQN) model; this includes all kinds of client-server architecture and many others, as will be shown in this thesis.

There are tradeoffs between performance and other quality requirements including maintainability, scalability, reliability, flexibility, security, etc. The software industry also requires good reusability and short time-to-market for profit purposes. Advanced software technologies that address these requirements often sacrifice performance. For example, CORBA (the Common Object Request Broker Architecture) provides interoperability across heterogeneous environments but introduces overheads for message encoding. In SAP applications written in the ABAP (Advanced Business Application Programming) language, proper use of batching of database operations can improve performance dramatically, but takes longer to implement and increases the maintenance effort. Therefore, it is helpful to have quantitative estimates of performance in order to know how much the performance will be sacrificed or gained before the tradeoff is made.

If performance analysis is begun in the early stages of design, such as architectural design, these tradeoffs can be analyzed and resolved before most of the development work is done. This is the purpose of the Software Performance Engineering (SPE) methodology introduced twenty years ago by Connie Smith [Smith86], and is the starting point of this research. This thesis will focus on a particular part of the SPE problem, which is the systematic use of design evaluation to suggest *design improvements*, and the identification of rules and strategies which can be applied automatically or semi-automatically to create *design change suggestions*. Performance principles from [Smith90] will be used as starting points. Since some of the principles are not targeted at changing the design and some are difficult for systematic use, new rules are needed. Aspects of performance problems, such as bottleneck patterns and latencies of synchronized executions could be used as input for rules. Design improvement suggestions could be changing interaction patterns, restructuring components, etc. More potential design improvements will be summarized in section 2.3.2. Their use will be discussed in section 5.4.

Performance evaluation and diagnosis are usually time consuming and require a thorough understanding of performance, which is one of the reasons why a fix-it-later approach is commonly taken in industry in regard to performance issues. In this research, the performance evaluation will be carried out using tools developed by other students in the *Performance by Unified Model Analysis* (PUMA) project [Woodside05] [PUMA08], so this work can concentrate on diagnosis and improvement.

This research is a part of the PUMA project and is aimed at providing meaningful feedback from performance evaluation to software design. Together with the previously mentioned tools, this enables a round trip between the software design world and the performance world. The Layered Queuing Networks (LQN) models created by other PUMA tools will be taken as inputs. Insight about the system performance can be gained through diagnosis based on the performance estimations. Bottleneck information and

suggested solutions can then be fed back into the PUMA tool chain, or directly to the user through UML specifications.

1.2 The Approach of the Research

This research focuses on two aspects of software performance engineering: diagnosis, which is to identify the root causes of performance problems in a software system; and improvement, which is to propose changes that can be made to the software design to improve its performance.

This research uses a rule-based approach to automate the above described diagnosis and improvement process, which is to formalize performance expert knowledge into rules, and to utilize an inference engine to interpret the rules on performance data and to generate design improvement suggestions. A rule-based system, named *Performance Booster* (PB) has been built to implement the approach.

Figure 1-1 Rule-based performance diagnosis and improvement

Figure 1-1 shows the information flow of PB. It is intended to be integrated into the PUMA toolset [PUMA08], which provides automated extraction of layered performance models from UML design models (step 1, described in [Woodside05][PetriuDB07][PetriuDC06][Gu02]). In this thesis the term *model* will always denote a performance model, and *design model* will denote a UML [OMG08a] software model. Performance model definition exploits annotations made using standard UML profiles (e.g. the SPT or Schedulability, Performance and Time profile [OMG02] and its successor MARTE [OMG08b]) and the automated extractions capture these annotated values as model parameters.

PB executes the loop of steps 2-3-4-5:

Step 2: solve the model using the Layered Queueing tools described in [Franks00];

Step 3: extract performance measures and identify rules to be applied;

Step 4: find changes to the model that will improve performance;

Step 5: apply the changes to the model.

After step 5 the changes can be manually mapped to the UML design (steps 6 and 7), using relationships between design features and the corresponding properties of the performance model.

The rules for improvement incorporate published guidelines and principles, including the principles and antipatterns described by Smith and Williams [Smith02][Smith00] and the bottleneck analysis described in [Franks06][Neilson95]. The relationship between a software design and its execution environment addressed in [Woodside02] underlies the relationship between design and configuration optimization, in this work. The present work captures and extends these concepts in general rules which can be applied for automated rule-based reasoning. Rules are applied within the rule-based system Jess [Friedman-Hill03][Friedman-Hill08].

1.3 Thesis Outline

The rest of the thesis is organized as following:

Chapter 2 describes approaches and research works in performance diagnosis and design improvement that were done by other researchers. Chapter 3 describes how this research fits in the big picture of performance engineering as part of the PUMA project and provides background knowledge on the techniques that will be used in this research, including design languages, performance modeling techniques and rule-based systems. Chapter 4 presents a Building Security System (BSS) example that has been improved by a human performance expert, which motivates the use of the present rule-based approach. Chapter 5 describes how performance expert knowledge is formalized into abstract rules, which is an intermediate step for building a rule-based system. Chapter 6 provides details on the implementation of the rule-based framework for automatic performance diagnosis and design improvement. Chapter 7 shows several study cases by using the rule-base framework, including a tutorial example on a web application, revisiting the BSS study case, and 300 randomly generated cases. Chapter 8 summarizes the contributions, and the potential directions that this research could lead to.

1.4 Contributions

The contributions of this thesis are:

1. A systematic approach of formalizing performance engineering knowledge into rules. The results of this approach are:
 - A set of rules which captures expert knowledge in the performance engineering field to the extent of being able to identify performance issues

in complex computer systems and to provide basic improvement feedback to designers.

- A set of surrogates of design properties in performance model
2. An extensible rule-based framework which automates performance diagnosis and a improvement suggestion procedure.
 3. A prototype implementation which can be used for proof of concept.
 4. A set of case studies, including one based on an industrial example.

CHAPTER 2 SURVEY ON PERFORMANCE DIAGNOSIS AND MITIGATION

This section describes previous work done by other researchers in performance diagnosis and design improvements.

2.1 Performance Models

Balsamo et al. give a good survey on model-based performance prediction in [Balsamo04]. Different performance models are used for performance evaluation of software systems, including Queueing Network and its extension -- Extended QN (EQN) or Layered QN (LQN), Petri Net (PN), Process Algebra, Simulation models, and Stochastic Process.

Queueing Networks (QN) has been widely used to model and analyze the performance of complex systems. QN are defined as a set of interconnected nodes. Each node has a waiting queue to hold tokens for service requests. Smith and Williams in [Smith02][Williams98] proposed a SPE-based methodology, which uses a software execution model represented by Execution Graphs (EG) and a system execution model based on QN models. Software behaviors are first specified in UML sequence diagrams and then transformed into the software execution model with information from class diagrams and deployment diagrams for completion. A QN model is generated by combining the software execution models and resource requirements acquired from user and performance specialists. Performance estimations of response time and device utilization can be obtained by solving the QN model. Cortellessa and Mirandola in [Cortellessa00] proposed the PRIMA-UML methodology which extended Smith's

approach by utilizing information from different UML diagrams to incrementally generate a performance model.

Menasce and Goma used CLISSPE (CLient/Server Software Performance Evaluation) specification for building QN models from a code-like specification [Menasce98][Menasce00]. CLISSPE is both a language and a tool. It can generate QN models from UML diagrams. It can also automatically compute service demands and estimate CPU and I/O costs which have important impacts on software performance and the decision on design choices. Balbo et al. combined queueing networks with generalized stochastic Petri Nets to model complex system behavior typical of software systems, including synchronized operations and processor sharing [Balbo88].

Generalized Stochastic Petri Nets (GSPN) is a low level performance modeling language. There are plenty of solving tools for these and other timed Petri Nets, such as ALPHA/Sim, GreatSPN, CPN Tools, HPSim [Petri04]. An important advantage of Petri Nets is that it can represent nearly all kinds of system architecture and design mechanisms. In [Merseguer01] and [Bernardi02], the authors proposed a systematic translation of Statecharts and Sequence Diagrams into GSPNs. Separate labeled GSPN subsystems were first generated for each unit in Statecharts and Sequence Diagrams. Then such subsystems were composed into one GSPN model by merging elements with the same labels. A significant disadvantage of PN is state explosion which makes it not suitable for analysis of complex systems.

Layered Queuing Network (LQN) [Woodside95a][Franks96][Franks98][Rolia95] is an extension to QN. It models software architecture and behavior in an intuitive way. Petriu and her students study the systematic generation of LQN models from UML specifications [Gu02][PetriuDC99][PetriuDC02]. UML activity diagrams annotated with UML performance profile [OMG02] expressed in Extensible Markup Language (XML) notation are taken as inputs. The tools can be integrated with commercial UML tools such as Rational Rose and ArgoUML that provide XML Metadata Interchange (XMI)

export. Dorin Petriu and Woodside presented an approach which derives LQN performance models from system scenarios described by means of Use Case Maps (UCM) in [PetriuDB02a]. A UCM2LQN tool was created for automatic model transformation and has been integrated into a general framework called UCM Navigator [PetriuDB03].

2.2 Performance Diagnosis

Performance problems in software systems can be caused by two types of reasons: one is an execution path which takes too long; the other is a bottleneck. A long execution path can be caused by function assignment; i.e., bad decomposition, too much overhead, bad coding, or lack of parallelism, etc. Symptoms for long path problems are usually long response time or high possibility of missing deadlines, while no obvious bottleneck can be identified. A bottleneck is the limiting factor on increasing the throughput of a software application or a component. It can be caused by a long delay in the holding time of a resource, such as: waiting time for synchronous operations (fork and join), lack of resources such as processor or memory, blocking at critical sections, and also by large execution demands. A symptom of a bottleneck is usually high utilization of software components or hardware resources. Long paths and bottlenecks may exist at the same time in a system or one may even cause the other. The question of how to navigate through the results of a software model and to identify the source of performance problems was discussed in a general way in [Woodside95b]. In a broader context of computer and network systems (not just software designs), various kinds of reasoning aids for performance diagnosis have been described.

Previous work on systematic performance diagnosis was mostly performed on software implementations rather than software designs. The Paradyn Parallel Performance Tool and Pablo are two well known tools for performance diagnosis from measurements as introduced in section 2.2.1. Model based diagnosis is more ad-hoc and lacking tools. Mechanisms used for identifying bottlenecks by measurement based tools and other research are described in section 2.2.2.

2.2.1 Existing Tools for Performance Diagnosis Based On Measurements

The Paradyn project is led by Professor Barton Miller and Professor Jeff Hollingsworth [Miller95]. The primary product of this project is a performance measurement tool for parallel and distributed programs. It instruments unmodified executable files by inserting performance instrumentations at execution time. It can provide precise performance data down to the procedure and statement level. Paradyn also automates much of the search for performance bottlenecks. In a bottleneck diagnosis, it usually starts from top level hypothesizes in three categories, excessive synchronous waiting time, excessive IO blocking time and CPU bound. Then it refines the hypotheses that tested as true (with performance problems) and eliminates those that tested as false (with no performance problems) until it reaches an appropriate granularity.

[Karavanic99] enhanced the Paradyn Parallel Performance Tool to use historical performance data gathered in previous executions of an application to increase the effectiveness of automated performance diagnosis. [Helm95] presents Poirot, an architecture to support automatic and adaptable performance diagnosis, by using the goals and methods that are similar to the hypothesis and refinement approach in Paradyn.

Pablo [Reed92] is a performance analysis environment designed to provide performance data capture, analysis and presentation across a wide variety of scalable parallel systems. The project is led by Professor Daniel A. Reed from The University of Illinois, Urbana-Champaign. It performs the diagnosis also based on instrumentation of executable code. The Pablo environment includes software performance instrumentation, graphical performance data reduction and analysis and support for mapping performance data to both graphics and sound. It is now headed to integration with data parallel Fortran compilers, based on the emerging High Performance Fortran (HPF) standard.

2.2.2 Mechanisms for Bottleneck Identification

A bottleneck is a limiting factor on increasing the capacity of a software application. Bottlenecks fall in two categories according to where they occur. A hardware bottleneck means hardware resources cannot provide enough power or capacity, thus slow down the software execution. A software bottleneck means one or more software components are saturated while their underlying hardware resources are not fully utilized. Identifiers for different bottlenecks of different causes or in different structures may be different. There are mainly two types of identifiers, waiting time and utilization.

Paradyn [Miller95] uses waiting time metrics as bottleneck identifiers. Excessive synchronous waiting time, excessive IO blocking time and CPU bound are the three top-level hypotheses. The problem with using waiting time is that it is an absolute value, which cannot intuitively identify a bottleneck. Long execution time in one component may not be a bottleneck if it is functionally necessary. Only when long waiting time caused by it occurred somewhere else, this component may become a bottleneck.

Utilization is the other type of bottleneck identifier that is popularly used. High utilization of a hardware device can identify a hardware bottleneck. But for software, saturation at one point may be caused by problems in the underlying components. Therefore, absolute

utilization of a software component may not be able to identify a software bottleneck. Neilson et al. gave an in-depth description of this phenomenon in [Neilson95] and proposed *bottleneck strength* as a metric for bottleneck in client/server systems and rendezvous networks. Bottleneck strength is defined as the ratio of a component's utilization to that of its most highly utilized server either directly or indirectly. A simpler but more efficient calculation was also given to be used when system measurements or simulation data were not available. It is more efficient in identifying a bottleneck than using absolute utilizations. It allows potential bottleneck to be detected without pushing the system to saturation. Peter Tregunno in [Tregunno03] used the bottleneck strength algorithm for bottleneck detection and diagnosis in layered software. A set of strategies for bottleneck mitigation were also described, which will be introduced in section 2.3.2.

2.3 Mitigation for Performance Problems

This section looks at how previous researchers have improved the performance of their studied systems.

Many previous research achieved performance mitigation by adjusting the system configuration, such as threading level, cache size, deployment, and scheduling policies, etc. A few also attempt design changes, such as introducing concurrent or asynchronous operations in an ad-hoc way.

Configuration adjustment is straightforward and easier in implementation. Most configuration mitigation strategies can be adopted at runtime or deployment time, without recoding the application. The performance improvement can be limited.

Design changes require deep understanding of the application structure and business logic. Therefore, most of the previous attempts for design changes are done in ad-hoc ways. The improvements are significant. Recoding is inevitable if the application has been already built.

2.3.1 Configuration Improvement

Changes made to the system configuration have an impact on performance. Research shows considerable interest in using models for system configuration. Configuration solves problems of allocation, replication of servers, routing, scheduling, etc. Examples of configuration activities include allocating software components to hardware nodes, adjusting buffer sizes, setting multithreading levels of software components, tuning priorities for execution, caching database connections, etc. A change in configuration sometimes can incur design changes. For instance, changing the concurrency level of a software component may require a change in the design to support multithreading. This is not the primary topic of this research, but it has to be considered, because a change in system configuration is usually considered before a design change in practice. Designers are in favor of configuration changes for its low cost and also because a change in design can only show its full effect under the best possible configuration and deployment alternatives [Xu03].

In [Menasce00], Menasce and Goma used the QN-based modeling tool CLISSPE to evaluate a database system under different CPU configurations and database configurations, and obtained suggestions for system capacity planning and database configuration on table caching and indices setting. Zheng and Woodside, in [Zheng03], described an approach to find optimal deployment and scheduling priorities for tasks in a class of distributed real-time systems by applying a heuristic search strategy to solutions of LQN models. Risse et al., in [Risse04], presented a systematic method to find a

satisfactory configuration with feasible effort, based on a two-step approach. First, a hardware configuration is determined based on a queuing network analysis and then a software configuration is incrementally optimized by simulating LQN models.

2.3.2 Methods for Design Improvement

Although many papers point out the problem spots incurring performance failures through performance model evaluation, few of them provide explicit feedback on solutions. Some papers do show opportunities for design improvement. Dorin Petriu and Woodside in [PetriuDB02b] showed considerable performance improvement for an online book store application. After finding out that the performance limitation was caused by a long waiting time at a database, the authors suggest separating out a read-only database for frequently accessed data or using an “optimistic” path which gives early response to customers before processing being completed. Gorton et al. in [Liu04] [Gorton03] evaluated an EJB based application known as Stock-Online using different EJB patterns and demonstrated that the prediction can be used for providing guidance on early stage design decisions by comparing the prediction results for different architectures. This author’s previous paper [Xu03] showed obvious performance improvement on a Building Security System by introducing second phase execution and moving heavy workload to second phase. The design change was achieved after systematic steps of tuning on software multithreading level and, finally, a long path problem surfaced.

Peter Tregunno proposed four software bottleneck mitigation techniques in [Tregunno03]:

- 1) increasing threading levels at the bottleneck task,
- 2) replication of the bottleneck task and its processor,

- 3) the reduction of phase one processor demand of bottleneck task and
- 4) the decreasing of the number of interactions that the bottleneck task has with lower layers.

The first and second techniques both try to add more resources to the system, either hardware or software resources. The third one, reducing phase one service demand is also a solution for path length reduction. It is especially suitable for the situation in which bottleneck problem and long path problem are mutually related. The example in Chapter 4 also shows this approach. Reducing requests through batching is one of the examples for the forth technique. Early binding also falls in this technique category. They are discussed in Smith's *Processing Versus Frequency Tradeoff Principle* and *Fixing-Point Principle* in the next section. Tregunno's set of four approaches seems to be incomplete; they do not address diverting load away from the bottleneck component by reducing the requests to it.

All of the above approaches on design improvement are performed in an ad-hoc way. Most of them take the following steps. First, performance models are created and solved. Then by comparing the results to performance requirements, improvement goals are set. Possible problem causes are identified by inspection of results. Potential improvement strategies are invented and the models then are modified and resolved. Alternatives for improvement suggestions can be compared through model results. Diagnosis and improvement activities are repeated until performance requirements are satisfied or the requirements have to be reexamined. An example will be shown in Chapter 4.

Until now, how to provide systematic guidance to software designers on design change is still an open question, which is also the purpose of this research. There are some general ideas about design change that are followed by designers that will be listed in the rest of this section. The nature of design change in regard to performance consideration is to introduce more concurrency or to shorten total execution demand.

There are two ways to introduce more concurrency. One is to add more resources, such as using multithreading [Tregunno03] [Xu03] and replicas [Pan96]. This solution is widely used now and is usually the first choice of solution for performance improvement. Adding more threads or replicas itself is not a design change. However, in order to implement multithreading and control on load balance among replicas, additional logic has to be designed. The other way of introducing more concurrency is to reduce blocking time. Interaction patterns between components have great impact on the blocking time. Therefore, changing the interaction pattern usually helps with minimizing the blocking time. For instance, using an asynchronous call instead of a synchronous call and using a forwarding call instead of a synchronous call both can efficiently reduce blocking time. There is a concept of second phase execution in the Layered Queueing Network (LQN) model. Research shows that moving first phase workload (both demand and outgoing service request) to second phase can greatly improve concurrency level [Xu03]. This will be shown in Chapter 4 as well. The balance of work along a pipeline can also affect performance.

Execution demand can be reduced in two ways. One is to reduce overhead occurring in the scenario, such as batching small operations, pooling resources, catching data, etc. Another is to optimize the functional execution demands.

2.4 Principles from Connie Smith

Connie Smith proposed several general principles for developing high performance software systems in her books [Smith90] and [Smith02]. The use of a model to improve a software design is the focus of these “performance principles” and of her work with

Williams on antipatterns [Smith00]. These principles are fundamental to the approach implemented in this research.

The principles are grouped into three categories: independent, synergistic and control principles. They are geared towards early stage of design decisions and are mostly focused on handling long path problems. Independent principles improve the performance of a task by improving the task's own performance parameters, such as reducing its resource requirements. Principles of this type include the fixing-point principle, processing vs. frequency tradeoff principle and centering principle. Synergistic principles improve the overall system performance through cooperation. They can reduce the average waiting time for inter-process synchronization. Shared resources principle and parallel processing principle belong to this type. The locality design principle is both independent and synergistic. Smith also makes instrumentation a principle which belongs to the third type of performance principles, i.e., control principles. It does not directly improve software performance, but it is vital to managing performance. In the rest of this section, the above mentioned principles will be introduced in more detail.

Fixing-Point Principle: For responsiveness, fixing should establish connections at the earliest feasible point in time, such that retaining the connection is cost-effective. This principle is also known as early binding. Fixing or binding means connecting the desired functions to the instructions that accomplish the action, or connecting the desired result to the data used to produce it. Early binding may save the overhead of retaining the connection at execution time. However, it may lose flexibility or portability. For example, a software system written in C is compiled into machine code executable on an operating system, i.e. the system is bound to an operating system at compile time. A system written in Java, on the other hand, is platform independent. It is translated into machine code by the Java Virtual Machine (JVM). Therefore, to fulfill the same functions, a system in C usually outperforms the same system in Java at the cost of losing platform independence. Finding the earliest feasible binding time is important in applying this principle.

Locality-Design Principle: Create actions, functions, and results that are “close” to physical computer resources or to other operations. “Close” here means being near in time, space, effect or degree. Examples of physical computer resources include instructions, files, database, processors, and storage devices. This principle provides a guide for function and data allocation in the design, i.e. deployment design. It encourages using local memory data and special purpose processors which correspond to space locality and effect locality. Degree locality can be explained as designing functions with workloads fit for processor’s capacity. Batching operations on a processor is used as an example for temporal locality, which could also be classified into the processing vs. frequency tradeoff principle.

Processing Versus Frequency Tradeoff Principle: Minimize the product of processing times frequency. This principle addresses the amount of work done per processing request and its impact on the number of requests made. The “tradeoff” looks for opportunities to reduce request frequency by doing more work per request, and vice versa. The total workload is the product of processing time per request and the number of requests. So minimizing this product guarantees minimal processing demand in total.

Shared-Resource Principle: Share resources when possible. When exclusive access is required, minimize the sum of the holding time and the scheduling time. Resource sharing is the prerequisite for parallel computing which brings dramatic improvements in performance. However, some resources have to be exclusively used, such as a writing lock. Exclusive use affects performance by introducing additional processing overhead to schedule the resource and the possible contention delay to gain access to the resource. The contention delay depends on how many processes request exclusive use and the length of time they hold them. Therefore, minimizing the sum of the overhead for scheduling and the holding time decreases the overall average wait time to gain access to the resource. There are four ways to minimize the holding time: minimize the processing time, hold only while needed, request smaller resource unit, and decompose one long request into multiple shorter requests. The latter two may introduce additional overhead

or waiting time for regaining the resource though. The processing vs. frequency principle may be applied to check whether it is worth it or not.

Parallel Processing Principle: Execute processing in parallel (only) when the processing speedup offsets communication overhead and resource contention delays. Part of this principle has been discussed in the shared-resource principle. In addition to resource contention delays, parallel processing also introduces communication overhead. Remote Procedure Call (RPC) is a technology for communication between different processes on the same or different processors. It brings overhead for marshalling and un-marshalling messages to be passed. Another factor that affects performance in parallel processing is synchronization delay. Sometimes one process has to wait for the result of another process in order to continue. Remedies similar to those for minimizing holding time of shared resources can be taken here.

Centering Principle: Identify the dominant workload functions and minimize their processing. Centering is based on the folkloric “80-20” rule for the execution of code within programs, which claims that less than 20 percent of a program’s code accounts for greater than 80 percent of its computer resource usage. Therefore, find and focus on optimizing this 20 percent code will give efficient performance improvement. To generalize it, less than 20 percent of the system functions that are provided will be requested by system users for more than 80 percent of time. These frequently requested functions are dominant workload functions. Minimizing their processing time will greatly affect the overall system performance.

Instrumenting Principle: Instrument systems as you build them to enable measurement and analysis of workload scenarios, resource requirements, and performance goal achievement. This is the foundation of our performance analysis and improvement.

2.5 Performance Expert Systems

[Hoogenboom92] described a knowledge system called the System Performance Advisor (SPA), which can provide advice to UNIX system administrators in order to assure that the system is reaching its potential performance capacity. SPA collects system data at run time by using UNIX utilities (such as `ruptime`, `ps`, `vmstat`) and can store historical data. The data are analyzed by forward-chaining rules and results are changes and advise to system administrator or commands that can take curative action (such as `mail`, `kill`, `rm`) directly on the target system. The approach was validated by artificially introducing problems while SPA is running and reporting on the success rate of problem identification. SPA focuses more on problem identification. Its therapy actions require human intelligence and expert knowledge to apply.

[Asogawa90] described an expert system named EXPERFORM (EXpert system for PERFORMance), which is specialized for diagnosing and proposing counter plans for performance problems on NEC's ACOS-4 series. It was targeted for performance issues raised by hardware resource insufficiency, inappropriate resource allocation and unfavorable scheduling strategies. The inference procedure took a suspect – verify – counter plan approach. The system was field tested and was evaluated as being comparable to a human expert with 3 to 5 years of experience. Development of EXPERFORM itself took 3 years (7 man-years).

Both systems do performance diagnosis based on data collected from running computer systems. Both systems can identify some performance problems and provide advices for therapy or counter plan. Both of them are mainly targeting hardware problems and configuration issues. Neither of them touches the system design.

Parsons and Murphy built a rule based performance diagnosis tool named Performance Antipattern Detection (PAD) [Parsons08] by using the Jess rule engine. The rules were targeting high level design patterns and Enterprise Java Bean (EJB) applications. PAD is

also based on monitoring data from running systems, similar to the above two expert systems. It extracts the run-time system design from monitoring data and detects performance antipatterns by applying rules to it. A system's run-time design was defined as an instantiation (or instantiations) of systems design decisions which have been made during development. Examples of antipatterns are: the Sessions-A-Plenty Antipattern (i.e. the overzealous use of session beans, even when they are not required); the Customers-In-The-Kitchen Antipattern where web tier components directly access persistent objects (e.g. Entity Beans); and the Fine Grained Remote Calls or Face Off Antipattern. The antipatterns are grouped into a number of categories, including antipatterns across or within run-time paths, inter-component relationship antipatterns, antipatterns related to component communication patterns, data racking antipatterns, pooling antipatterns, and intra-component antipatterns.

Compare to this work, PAD does not use the design before it is implemented. Instead, it uses architectural patterns (antipatterns) to identify potential problem rather than performance evaluation.

CHAPTER 3 BACKGROUND KNOWLEDGE

This chapter introduces the background knowledge and techniques that will be used for this research so that we could have a specific context for discussion. Section 3.1 provides an overview of the Performance Unified Modeling Analysis (PUMA) project, which is aimed at bridging the gap between software design world and performance world by proving round trip transformation and analysis between the two. This research is a part of the PUMA project, developing on the feedback path from performance analysis to software design. Section 3.2 introduces UML and its extension profiles which are important modeling languages used in the design world, and are used as design languages in the PUMA project. Section 3.3 introduces the Layered Queuing Network (LQN) and its estimates. LQN is an extended Queueing Network modeling language and has advantages for modeling software behavior. It is chosen as the target performance modeling language in the PUMA project. Section 3.4 provides introduction on rules, rule-based systems and the Jess rule engine which is used for inference in this research.

3.1 PUMA Project

The PUMA project [Woodside05][PUMA08] provides an approach to enable an early stage of performance analysis by automatic transformation from a UML design model to a LQN performance model “Do it now and fix it later” is commonly seen in industry, since the time to market usually takes precedence over system performance. Performance diagnosis is time consuming and needs special knowledge. Therefore, system performance is usually ignored at the design stage. This may result in great damage in performance. If an inappropriate software architecture was used, changing it after the

implementation will be even more expensive. The PUMA project is motivated to address this issue.

Figure 3-1 shows an overview of PUMA project. The software design is first represented in UML specifications. Then a core scenario model is derived from the design and is further translated into a performance model. A Core Scenario Model (CSM) language [PetriuDB04] is defined to simplify the transformation from UML model to LQN model, while supporting other design languages and performance modeling languages. [Woodside05] describes the translation from CSM to labeled GSPN (Generalized Stochastic Petri Nets).

Figure 3-1 PUMA project

Platform and environment information can be added into the performance model by utilizing corresponding submodels in order to study the performance impact of different platforms and to guide design choices on platforms. Templates for building submodels of

Enterprise Java Beans (EJBs) have been developed to assist modeling EJB applications [Xu05a] [Xu05b].

By solving these performance models and analyzing the results, insight about performance aspects of the design can be obtained. Performance is satisfactory if it meets the performance requirements in the design documents. Bottlenecks can be found by diagnosis based on the detailed model results. Multiple experiments are usually needed for this diagnosis. Therefore plans for experiments are necessary for efficient diagnosis.

Suggestions for design changes can be derived from the analysis of model results. Changes may first be made in performance models and then be fed back to core scenario models or UML design models if the performance is improved after the changes. Not all structures or scenarios can be presented precisely in the performance model. There is usually no one-to-one mapping from performance model changes to UML design changes. Adjustment and selection of the change suggestions are needed. Section 6.7 provides suggested mapping between performance model changes and UML design changes for the mitigation strategies investigated in this thesis.

The backward path in the PUMA project, starting from the experiment plan to the result diagnosis until suggestions for design changes, is the main focus of this thesis.

3.2 UML and Its Extension Profiles

The Unified Modeling Language (UML) [OMG08a] is a general-purpose notational language for specifying and visualizing complex software, especially large, object-oriented projects. It is now widely used in software design (e.g. [Gomaa00a]). This section describes UML 2.0. Examples using the previous version UML 1.4 will be seen in Chapter 4.

There are many sub models or diagram types in UML. This section focuses on the concepts that are related to this research, including use cases, classes, components, deployments and system behaviors. For the system behavior, this introduction focuses on the interaction behaviors which are represented by sequence diagrams.

3.2.1 Use Cases

Use cases represent the functional requirements of a system. They capture what a system is supposed to do. The key concepts associated with use cases are actors, use cases and the subject. In a use-case diagram, the subject is the system under consideration, denoted by a square frame which contains the use cases. It illustrates the boundary between what is inside and outside of the system. A use case, presented by an oval, represents a unit of functionality provided by the system. Actors represent human beings or external devices or systems that interact with the system.

Figure 3-2 Example use case diagram

A use-case diagram also shows the relationships between actors and use cases, and the relationships among different use cases. Connectors from an actor to use cases inside the system illustrate the needs of the actor from the system. There are two kinds of relationships among use cases, the extension relationship and the including relationship.

Figure 3-2 shows the use case diagram for a Building Security System. The system controls user access to a building and logs all access information. It also manages the video surveillance of the building, storing video images captured by video cameras installed in the building. A more detailed introduction to the building security system will be given in Chapter 4.

3.2.2 Classes and Components

Classes and components are like the building blocks of a software system.

A class describes a set of objects that share the same features, including attributes and operations. A Class diagram shows the static structure of a system, especially the entities and their relationships within the system. Figure 3-3 shows a class definition of the CardInfo entities. The relationships among classes include inheritance, associations, aggregation, and composition.

Figure 3-3 Example class

Figure 3-4 Example component

A Component represents a modular part of a system that encapsulates its contents and whose manifestation is replaceable within its environment. A component defines its behavior in terms of provided and required interfaces. A component is usually implemented by a group of classes. A component may have different implementations. For different implementations, the internal class structure of the component may be different. Figure 3-4 shows the `AccessControl` component with provided interface, Access Request, and requested interface for Database. The high-level view of a software design is usually represented by component diagrams.

3.2.3 Deployments

The deployment of a software application is an important factor that impacts the system performance. It determines the contention at underlying hardware devices or other computation resources. In UML 2.0, a deployment is the allocation of an artifact or artifact instance to a deployment target.

An artifact is the specification of a physical piece of information that is used or produced by a software development process, or by deployment and operation of a system. Examples of artifacts include model files, source files, scripts, and binary executable files, a table in a database system, a development deliverable, or a word-processing document, and a mail message.

A deployment target is the location for a deployed artifact to reside. It can be represented by a computation node, a device, or an execution environment.

Figure 3-5 shows an example deployment. The implementation file of the `AccessControl` component is deployed on the `ApplicationCPU` node.

Figure 3-5 Example deployment

3.2.4 Behaviors

Our performance analysis is based on system behaviors, which can be described by activity diagrams, state machines, and sequence diagrams.

Activity Diagrams

Activity diagrams are commonly used to describe lower-level behaviors. An activity diagram describes the execution of a series of actions. Data or messages transferred between actions can also be annotated.

Figure 3-6 shows the Access Control activity diagram of a Building Security System. Activity diagrams can also be organized in swim lanes to show the deployment of activities to objects in the system. In UML 2.0, swim lanes are evolved to partitions.

Figure 3-6 Example of an activity with input parameter

State Machines

In UML 2.0, State Machines can be classified into behavior state machines and protocol state machines. Behavior state machines are used for modeling the discrete behavior of various elements in a system. Protocol state machines are used for expressing the usage protocol of part of a system. Figure 3-7 shows the state machine of a DoorLock entity in the building security system.

Figure 3-7 Example of state machine

Sequence Diagram

The sequence diagram is the most common kind of interaction diagram. It is the foundation of this performance analysis. The performance models in this work are systematically derived from sequence diagrams. A sequence diagram potentially models the competition for software resources.

A sequence diagram describes the message interchange between a number of lifelines in a system design with their ordering. It is suitable for both high level designs and detailed designs.

A lifeline represents the role taken by an individual participant in the interaction. It is usually an instance of an element in the system design, such as a class, a device, a human being or an external system. A rectangular bar above alone a life line represents a single thread of control occurred on this life line.

A sequence diagram is constructed by interaction fragments or combined interaction fragments. A combined interaction fragment defines an expression of interaction fragments. There are some important combined interaction fragments that we are interested in, including Alternative, Option, Parallel, Critical Region, and Loop. In sequence diagrams, they are denoted by rectangular boxes containing interaction fragments with interaction operators (3-letter abbreviations except *critical* and *loop*) in the upper-left corner. Sometimes, there is a dashed line that separates the interactions within the combined interaction fragment into two or more parts.

- **Alternative:** the interaction operator *alt* designates a choice of behavior. There are two or more parts of interaction fragments within the rectangular. Only one of them will be executed.

- **Option:** with the interaction operator *opt*, the interaction fragment within the rectangular will either be executed or not.
- **Parallel:** the interaction operator *par* designates that the parts of interaction fragments can be executed in parallel and interleaved. Within each part, the ordering of interactions is still preserved.
- **Critical Region:** the interaction operator *critical* designates that the enclosed interaction fragments can only be executed in a single thread of control. Any part within it cannot be interleaved with other executions on the same lifeline.
- **Loop:** the interaction operator *loop* designates that the enclosed interaction fragments will be repeated for a number of times. Maximum and minimum numbers of iterations can be defined.

Figure 3-8 Example of a sequence diagram

Figure 3-8 shows the sequence diagram of an Access Control scenario in the Building Security System with an Option combined interaction fragment in it. The scenario is simplified. In a sequence diagram, half open arrows represent asynchronous messages. Solid arrows represent synchronous operations. Open arrows with dashed lines represent the reply messages of synchronous operations.

As we will see in Chapter 4, the expression of alternatives in UML 1.4 is different.

3.2.5 UML Extension for Annotating Performance Properties

UML is an extensible language. In order to enhance the ability of modeling different aspects of software systems, there are several profiles defined for UML, such as the UML profile for CORBA, UML profile for Enterprise Application Integration (EAI), UML profile for Enterprise Distributed Object Computing (EDOC), UML Profile for QoS and Fault Tolerance, and UML Profile for Schedulability, Performance and Time (SPT) [OMG02]. In this research, our interest is in the UML SPT profile which is currently being upgraded to the UML profile for Modeling and Analysis of Real Time Embedded Systems (MARTE) [OMG08b].

UML SPT profile was adopted by OMG in 2003. It provides a way for designers to input time-, schedulability- and performance-related aspects in software design, thus enables deriving performance models that could be used to make quantitative predictions regarding these characteristics and, in turn, to facilitate optimization in software design.

Important extensions in stereotypes of UML SPT profile include *PAcontext*, *PAClosedLoad*, *PAopenLoad*, *PAhost*, *PAresource*, *PAstep*, *GRMacquire* and *GRMrelease*. There are tags associated with these stereotypes, annotating important values to be used for performance evaluation.

PAcontext is the stereotype for performance context. A performance context specifies one or more scenarios that are used to explore various dynamic situations involving a specific set of resources. For example, a sequence diagram can be a performance context.

A workload specifies the intensity of demand for execution of a specific scenario as well as the required or estimated response times for that workload. There are two types of workload, closed workload and open workload, annotated by *PAClosedLoad* and *PAopenLoad* stereotypes separately in UML SPT profile. Tag *PApopulation* for *PAClosedLoad* defines the number of clients in the system. Tag *PAoccurrencePattern* for *PAopenLoad* defines the distribution pattern of the request arrivals. Tag *PArespTime* can be defined for both *PAClosedLoad* and *PAopenLoad* to describe the response time observed by a client.

The stereotype *PAresource* is used to annotate passive resources that are accessed during the execution of an operation. A passive resource may be shared by multiple concurrent resource operations. A *PAresource* can be a physical device or a logical protected-access entity. An important tag associated with *PAresource* is the *PAcapacity*, which specifies the multiplicity of the resource.

The stereotype *PAhost* is used to annotate processing resources, such as a processor, interface device or storage device. It usually appears in the deployment diagram.

The stereotype *PAstep* is used to annotate steps (or operations) in a scenario that takes a finite time to be completed. A step itself can be a scenario which is constructed by multiple smaller steps. The steps are tagged with a demand value for processing time (tag *PAdemand*) which is the CPU demand for the step.

The stereotype *GRMacquire* is used to annotate the actions that acquire protected resources such as semaphores, locks or buffers. Once the resource is no longer needed, the *GRMrelease* action will be performed.

Figure 3-9 Example of a scenario annotated with SPT profile

The values for time related tags *PAdemand* and *PArespTime* are represented as string arrays in the format of: “(“ <source-modifier> “,” <type-modifier> “,” <time-value> “)”. The <source-modifier> defines the source of the value. Possible content is ‘req’, ‘asmd’, ‘pred’, or ‘msr’, meaning respectively: required, assumed, predicted, or measured. The <type-modifier> used in this thesis includes ‘mean’, meaning average value, and ‘percentile’, meaning the percentile values for the variable. Other possible values for <type-modifier> are defined in SPT profile [OMG02]. <time-value> is a time value described by a numeric value with a time unit such as millisecond (ms).

Figure 3-9 shows the same scenario as in Figure 3-8 annotated with performance data using the UML SPT profile.

3.3 Layered Queueing Network (LQN)

Layered Queueing Network (LQN) [Woodside95a][Franks96][Franks98][Rolia95] is chosen as the performance modeling language used in this research. It models software architecture and behavior in an intuitive way. There are powerful solving tools including an analytical solver and a simulation solver, which can get abundant results on every software components and hardware devices. LQN solvers scale well. LQN analytical solver can solve very complex models with tens of nodes efficiently. LQN simulation solver solves relatively slower than the analytical one, but it gives more accurate results. [Franks98], [Maly00] and [Dilley98] show applications of LQN.

The rules in Chapter 5 reference an abstract Resource Interaction Model which could be interpreted in any performance formalism. However the implementation and examples in this research use the LQN model.

3.3.1 Model Elements and Parameters

In a LQN model, each process is represented by a “task” shown in diagrams as a rectangle with one or more “entry” rectangles attached to its left. A “task” models an active object, process, thread or any other logical resource that requires mutual exclusion (such as a buffer pool). An “entry” models the operation which processes a distinct class of messages received by the task, For example, if a “task” models an object, an “entry” models a method.

The top level tasks in a model are usually reference tasks that model the workload generators. There are two types of workload, closed workloads and open workloads. A closed workload is modeled by a reference task with optional multiplicity and an entry

with thinking time in its second phase. Each copy of the reference task repeatedly generates a new request to the underlying services of the system after waiting for a period of thinking time. An open workload is modeled by a reference task with an entry that has a request arrival rate. It can also have multiplicity and thinking time defined. The poisson distribution is used for arrival rates.

Arrows to other entries indicate requests made by an operation to other components. A solid arrowhead shows a synchronous call (where the caller expects a reply, and is blocked until receiving it); it may be shown as a call and its corresponding return in the sequence diagram. An open arrowhead shows an asynchronous message, and a dashed arrow shows a synchronous request which is forwarded to another task.

A server task can carry out part of its work after replying to its client; this is termed a “second phase” of service, and may have its own workload. For each entry the host demand is represented by $[s1, s2]$ for first and second phase CPU demand in time units. For each request arc the mean numbers of calls in the two phases are represented by $(y1, y2)$; the second phase value is optional. The general concept of second phase workload is further discussed in section 6.7.3 and its representation in a UML design is shown in Figure 6-8 in that section. A request arc in the model can have a mean number of calls per entry invocation, or a deterministic integer number.

Task multiplicities represent the number of identical replicated processes (or threads) that work in parallel serving requests from the same queue, or the number of logical resources of the same type (e.g., buffers). They are usually corresponding to the *PACapacity* tag values in the UML models.

Processing resources, including CPUs or other types of devices, are represented in circles. They are always at the bottom layer of LQN models. All tasks must be assigned and executed on one of the processing resources.

Replication is another modeling mechanism representing multiplicity of resources, including tasks and processing resources. Different from simple task multiplicity, each replica has its own queue. When a server task is replicated, the client task sends out requests randomly to one of the queues before the replicas. Therefore, even if a replica is idle at any given moment, there may be requests waiting for the same type of resource in the queue of another replica.

Figure 3-10 Example of LQN model

Figure 3-10 shows the LQN model derived from the scenario in Figure 3-9. Each lifeline is modeled as a task with entries for the operations on the lifeline. Additional tasks for Users and Database are added corresponding to the messages across the scenario boundary.

Although a LQN model is powerful for modeling various kinds of software structures, it does have limitations when modeling some features, such as a reading lock. To model those special cases, approximations can be made to simulate the behaviors.

3.3.2 Estimates

Estimates, including utilizations, service time, waiting time, throughput, deadline and missing probability, are used for performance evaluation and bottleneck identification in LQN models.

Utilization: the mean number of resources being busy during the execution. The value represents the number of busy threads for a software resource (task) or the number of busy processors. When a resource is executing or being blocked by underlying services, the resource is busy. For a single-unit resource this is a fraction, otherwise it may exceed one.

Normalized utilization: Utilization divided by the units of resource configured. A resource with a normalized utilization of 100% is fully saturated.

Service time: the time spent on executing its own processor demand and being blocked by embedded services. The blocking time for an underlying service is the response time of that service.

Response time: the time starting from the moment that a client issues a service request to the moment that a response corresponding to the required service returns to the same client. The server that receives the request from the client and the server that sends the response to the client may not be the same one. For example, in Figure 3-11, server A is a forwarding server which receives a request from the client. The actual response is sent from server B to the client. Here the response time includes waiting time for the server A task, waiting time for processor by server A task, server A task execution time, waiting time for server B task, waiting time for processor by server B task and server B task execution time.

Figure 3-11 Response time of forwarding call

Waiting time: The time spent on waiting for service of software component or hardware device, not including the service itself.

Throughput: average number of service requests fulfilled per unit time.

Deadline and miss probability: A service may have a requirement for its maximum service time, which is called a deadline. Then the miss probability, the probability that the deadline is missed, indicates the performance satisfaction of the system.

3.3.3 LQN Solvers

The LQN model can be solved analytically or by simulation.

The analytic solver, *LQNS*, calculates the estimations of a LQN model by using mathematical theory and approximation. It transforms the LQN model into a set of queueing network models and then solves these models following complex algorithms [Franks00]. It can solve models quickly. However, the results are sometimes rough, especially when the model involves complex communication patterns such as forwarding

calls. Another limitation of the analytical solver is that it only gives mean values and, in some cases, variances of the estimations.

The simulation solver, *LQSIM*, simulates the behavior of a LQN model. It gives more accurate results for all kinds of model structure and communication patterns. However, it is time consuming. The more accuracy it requires, the longer it takes to solve the model. An advantage of the simulation solver is that, besides giving mean value of estimations, it also gives the 95% and 99% confidence intervals of all estimations.

There is another useful tool, *SPEX*, which can be used to batch multiple solutions of a LQN model in different configurations, and extract the results in a table. It allows the user to define input and output variables in a LQN input file. Input values for assigned variables can be defined by enumeration or by using a simple linear increasing formula. Estimations of output variables corresponding to each input value will be extracted and tabled in an output file. Each solution of the LQN model with a specific input variable value still has its own full output file. *SPEX* itself is not a solver. It calls the analytical solver or the simulation solver as required.

3.4 Rule-based Systems and the Jess Rule Engine

This research uses a rule-based approach to solve software performance issues. We follow a lot of rules in our day-to-day life. A human being itself is a highly intelligent rule-based system. To leverage the power of computer systems, expert systems or knowledge-based systems use rules to capture human expert knowledge in certain fields, such as medical diagnosis, product configuration, etc. This section introduces some fundamental concepts in rule-based systems, and the Jess rule engine which is used for implementation in this research.

3.4.1 Rules and Rule-based Systems

A *rule* can be defined as one or more actions to be taken under specified conditions. For example, we follow the rule “stop at a red light” when driving. From the definition and the example, we can see that a rule includes two essential parts: a condition, e.g. “red light”, and an action, e.g. “stop”.

Rules can be written more formally as *IF-THEN* statements in pseudo code. For instance, the above mentioned “stop at a red light” rule can be written as following:

```
IF light is red
THEN stop
END
```

Script 3-1 Red Light Rule

In this form, the *IF* part of a rule represents the conditions for triggering the rule. In rule-based systems it is called the rule’s *left-hand side (LHS)*, *predicates* or *premises*. The *THEN* part describes the actions to be taken when the rule is triggered, and is called the *right-hand side (RHS)*, *actions* or *conclusions*.

A *rule-based system* uses *rules* to derive *conclusions* from *premises* [Friedman-Hill03]. Early rule-based systems were designed to address issues of certain fields and were also known as expert systems or knowledge systems. Over time, other fields also require expert knowledge to be captured and reused when human experts are absent. Researchers realized the need for general purpose rule-based systems which separate the functionality of recognizing and interpreting rules from rules that represent domain knowledge. Such general purpose systems are named *expert system shells* or *rule engines*.

EMYCIN (Empty MYCIN) [Melle79] was the first expert system shell built by developers of MYCIN [Davis77], which is an expert system for diagnosing bacterial infections of the blood. CLIPS and Jess are two currently popular rule engines. By using

these rule engines, modern rule-based system developers are released from complex algorithms and tedious coding for rules and facts, and rather focus on developing rules by formalizing domain expert knowledge.

3.4.2 Architecture of a Rule-based System

Figure 3-12 shows the architecture of a typical rule-based system. It includes an *inference engine*, a *working memory* and a *rule base*. The *inference engine* is the brain of a rule-based system, which mimics human thinking activities to interpret and execute rules. It controls the process of applying rules to data and generating output. Data are usually represented by facts stored in *working memory* for the inference engine to work on. They represent a status of the system. Rules are deployed in the *rule base*. Rather than simply storing rules in text strings, the *rule base* usually utilizes a rule compiler which processes and reorganizes rules into forms that can be accessed by the *inference engine* more efficiently.

The *inference engine* works in discrete cycles and has three main parts: a *pattern matcher*, an *agenda* and an *execution engine*. A cycle starts with the *pattern matcher* looking for facts in *working memory* that can satisfy premises of rules from the *rule base*. Satisfied rules are marked as active and are put into the *agenda*. Multiple rules may be matched at the same time. Figure 3-12 shows an example in which two rules, r1 and r2, both have their premises satisfied and thus are both being put in the *agenda*. Multiple rules in the *agenda* at the same time form a conflict set, since all of them are ready for firing. The inference engine orders the active rules in the *agenda* by a process called conflict resolution (the algorithm for conflict resolution is complex and is beyond the scope of this thesis). The first rule after ordering will be picked up by *execution engine* to fire. Firing of one rule may result in modification of facts in *working memory*. Then the cycle is repeated again until no more rules can be fired.

Figure 3-12 Architecture of a typical rule-based system [Jess in Action]

3.4.3 Developing a Rule-based System

As mentioned in section 3.4.2, using a general purpose rule engine enables a researcher to focus on defining rules rather than worrying about a complex algorithm for manipulating data and rules. Most of the effort in building a rule-based system is usually spent on gathering, organizing, structuring and encoding expert knowledge.

In contrast with normal programs which are mostly *procedural*, in the sense that the program executes a limited number of steps in more or less a certain order as defined in advance, rule-based programs are written *declaratively*, which means the developer only defines what action is to be taken or what conclusion is to be drawn under a set of certain conditions instead of a step-by-step execution procedure. Rules are discrete. The rule engine will take care of control on execution flow and status update.

For example, with only the two rules shown in Script 3-2, we can decide the client-server relationship between any two resources (either hardware or software) in a layered structured system. The process of traversing each resource in the system along the calls is automatically executed by the rule engine. The rules are presented here in a high-level pseudo-code. They will be expanded into detailed rules which treat hardware resources and software resources differently in Chapter 5 and Chapter 6.

```
IF resource A calls resource B
THEN resource B is a server of resource A
END
```

```
IF resource B is a server of resource A
AND resource C is a server of resource B
THEN resource C is a server of resource A
END
```

Script 3-2 Rules for Identifying Client-Server Relationship

This approach makes it easier for researchers to divide a complex problem into smaller sub-problems, and shift some of the burden for dealing with tangled problems to the rule engine. For example, by adding one more rule as shown in script 3-3 to the above small rule set for clients and servers, we obtain a rule base that can identify “root bottlenecks” in a multi layer system. A root bottleneck is the root cause for saturations in a software system. It is a saturated resource that sits in the bottom layered of a saturation chain in the system.

```
IF resource A is saturated
AND no server of resource A is saturated
THEN resource A is a root bottleneck
END
```

Script 3-3 Root Bottleneck Identification Rule

Figure 3-13 shows the standard process for building a rule-based knowledge system [Jackson98].

Figure 3-13 Process of building a rule-based system

The first step is to gather expert knowledge in the targeted field presented in natural language. In the analysis stage, the knowledge is organized into more formal representations, such as a semantic network. A semantic network [Sowa92] is a directed or undirected graph consisting of vertices, which represent concepts, and edges, which represent semantic relations between the concepts. Examples will be seen in Chapter 5 of this thesis, when our performance expert knowledge is formalized into semantic networks.

Then the knowledge is further encoded into decision tables and rules in order to be used by the inference engine of the knowledge system. This research follows this process in building the rule-based software performance expert system that automates software performance diagnosis and providing suggestions on design improvements.

3.4.4 The Jess Rule Engine and the Jess Language

There are off-the-shelf rule engines available, such as CLIPS [Riley08], Jess, JBoss Rules, and Drools. Jess is a Java-based, open source rule engine [Friedman-Hill08][Friedman-Hill03], which was developed at Sandia National Laboratories in Livermore, California in the late 1990s. Being written in Java itself, its ability to access Java utilities makes it an ideal tool for adding rules technology to Java-based software systems. Therefore, choosing it for implementation in this research becomes straightforward since we have a LQN core model and a GUI editor program JLQNEditor already written in Java.

Jess uses an enhanced version of the Rete algorithm [Forgy82] to process rules effectively. Rete is an efficient pattern matching algorithm which was designed by Dr. Charles L. Forgy of Carnegie Mellon University and first published in 1974. Rete organizes rules and their matching facts into a network of nodes, which has a structure of a generalized trie. A trie is also called a prefix tree, which is an ordered tree data structure with keys being stored in arcs instead of in tree nodes [Trie08]. Rete is designed to sacrifice memory for speed. Considering this factor, some of the rules in this research, which requires serial steps of operations, such as complex LQN model manipulations, are coded in Java procedures instead of Jess rules for memory efficiency.

Jess is not only a rule engine, but also a scripting environment, which provides the capability of scripting declarative rules. Jess was originally inspired by CLIPS [Riley08], which is an open source rule engine written in C with a longer history than Jess, so the Jess language is quite similar to the CLIPS language in syntax.

A Jess language unit is a list of tokens enclosed in a set of parentheses. Lists can be embedded. A token starting with a question mark “?” is a variable and can hold any type of data object. The keyword `defrule` defines a Jess rule. LHS and RHS in a rule are separated by symbol `=>`. Multiple conditions listed in the LHS of a rule are combined in an AND relationship. *Connective constraints* including `&` (*and*), `|` (*or*), and `~` (*not*) can also be used to define complex logical expressions. Multiple actions listed in the RHS of a rule will be performed in sequence when the rule is fired.

Complex constraints or actions can also be defined as functions in Jess. Functions can be written either in Java or in the Jess language. Jess uses a prefix notation for function calls, i.e. the head of a list for a function call is the name of the function being called. A *Predicate function* (a Boolean function) can be used in the LHS of a rule to define constraints by being preceded with a colon (:).

Script 3-4 shows a Jess language version of the three rules for root bottleneck identification that were introduced in section 3.4.3. Three rules are defined in Jess language: `direct-cs-rule`, `transitive-cs-rule` and `root-bottleneck-rule`. The `direct-cs-rule` says, if a fact `call` exists with caller `?a` and callee `?b`, a `client-server` fact will be asserted with client name `?a` and server name `?b`. `?a` and `?b` are variables used for temporary value storage.

The `transitive-cs-rule` has two constraints with an AND relationship. It says if two `client-server` facts exist at the same time with the first one's server being the second one's client, a new `client-server` fact will be asserted with the first one's client and the second one's server.

The `root-bottleneck-rule` uses a predicate function `is-saturated` in defining LHS constraints. It says if a resource `?a` is saturated, and no server of `?a` is saturated, a root-bottleneck is identified at `?a`.

A data structure for facts such as `call`, `client-server`, `resource`, and `root-bottleneck` can be defined by keyword `deftemplate` in Jess. The definitions for these structures are omitted here.

```
(defrule direct-cs-rule
  (call (caller ?a) (callee ?b))
  =>
  (assert (client-server (client ?a) (server ?b))))

(defrule transitive-cs-rule
  (client-server (client ?a) (server ?b))
  (client-server (client ?b) (server ?c))
  =>
  (assert (client-server (client ?a) (server ?c))))
```

```

(defrule root-bottleneck-rule
  (resource (name ?a&:(is-saturated ?a)))
  (not (client-server (client ?a)
                      (server ?b&:(is-saturated ?b))))
  =>
  assert (root-bottleneck (name ?a)))

```

Script 3-4 Bottleneck Identification Rule-Set

A Jess rule base can be partitioned into manageable modules by using the `defmodule` keyword. Jess engine allows one *activated* module at a time in runtime. Only the rules from the activated module can be fired on current facts. This feature enables partitioning of the working process into stages and controlling of the stage flow. In PB, the rule base is partitioned into 5 modules which will be introduced in section 7.1.

CHAPTER 4 AN EXAMPLE IN PERFORMANCE DIAGNOSIS AND DESIGN IMPROVEMENT

This chapter presents an example of a model, and design improvement previously reported in [Xu03]. The purpose of this example is to illustrate how substantial improvements can be made to a performance model systematically by a human expert, changing both the configuration and the design of the software. Analysis of the resolution process revealed rules that could be followed in similar situations, which motivated the rule-based approach. The example is expressed in UML 1.4 notation, as it was defined before UML 2 was available.

4.1 Design Model and Performance Model for the Building Security System

The example used is a Building Security System (BSS) which is a distributed application designed to control access and to monitor activity in a building like a hotel or a university laboratory. Scenarios derived from two Use Cases will be considered, related to control of door locks by access cards, and to video surveillance. Figure 4-1 shows the planned deployment of the BSS. This description is taken from [Xu03].

In the Access Control scenario a card is inserted into a door-side reader, read and transmitted to a server, which checks the access rights associated with the card in a data base of access rights, and then either triggers the lock to open the door, or denies access. In the Acquire/Store Video scenario, video frames are captured periodically from a number of web cameras located around the building, and stored in the database. Figure

4-2 and Figure 4-3 show the sequence diagrams for the two scenarios annotated with performance parameters using a UML SPT profile.

Figure 4-1 Deployment diagram for the BSS

Figure 4-2 shows the detailed specification of the Access Control scenario. The User provides an open workload, meaning a given arrival process. The tagged values define it as a Poisson process with a mean interarrival time of 0.5 seconds, and state a percentile requirement on the response time (95% of responses under 1 second). They also define a variable name \$UserR for the resulting 95th percentile value, to be estimated. Each step is defined as a focus of control for some component, and the stereotype can be applied to the focus of control or to the message that initiates it; it can also be defined in a note. The steps are tagged with a demand value for processing time (tag PADemand) which is the CPU demand for the step. The request goes from the card reader to the Access Controller software task, to the database and its disk, and then back to execute the

check logic and either allow the entry or not. openDoor is a conditional step which can be tagged with a probability (PAProb) which here is set to unity.

Figure 4-2 Annotated sequence diagram for Access Control scenario

The devices are stereotyped as `<<PResource>>`, as in the deployment diagram, and so are the software tasks AccessController and Database; this is because a task has a queue and acts as a server to its messages. A resource can be tagged as having multiple copies, as in a multiprocessor or a multithreaded task. The Database process is tagged with 10 threads, by {PACapacity = 10}, and its disk subsystem is tagged as having two disks.

Figure 4-3 Annotated sequence diagram for Video Store/Acquire scenario

The specification for the Video Store/Acquire scenario is shown in Figure 4-3. There is a single VideoController task which commands the acquisition of video frames from \$N cameras in turn, by a process AcquireProc. The initial step is the focus of control of VideoController which is stereotyped as a closed workload source with one instance, with a required cycle time having 95% of cycles below 1 second, and a predicted value \$Cycle to represent the model result. AcquireProc is a concurrent

process (<<PAresource>>). It acquires a Buffer resource by a step `allocBuf` which is also stereotyped as <<GRMacquire>>, indicating a resource acquisition. Buffer is a passive resource shown in the deployment diagram with a multiplicity `$Nbuf`, managed by `BufferManager`. In the sequence diagram the use of Buffer is indicated by a note and by the stereotype <<GRMacquire>>. In the base case, `$Nbuf` is set to 1. Once a buffer is acquired, `AcquireProc` requests the image from the camera, receives it and passes the full buffer to a separate process `StoreProc`, which stores the frame in the database and releases the buffer. The `writeImg` operation on the Database has a tag `PAextOp` to indicate that it calls (`$B` times) for an operation `writeBlock`, which is not defined in the diagram. This operation can be filled in, in the performance model, by a suitable operation to write one block of data to disk.

The system must implement other Use Cases as well, such as operations for administration of the access rights, for sending an alarm after multiple access failures, or for viewing the video frames, but for simplicity we assume that the main performance concerns relate to the two Use Cases described above.

Both scenarios have delay requirements. The access control scenario has a target completion time of one second, and the surveillance cycle has a target of one second or less between consecutive polls of a given camera. In both cases we will suppose that 95% of responses to users, or of polling cycles, should meet the target delay. Further, it is desired to initially handle access requests at about 1 per 2 second on average, and to deploy about 50 cameras. Additional camera capacity would be desirable, and a practical plan for scaling up the system to larger buildings and higher loads is to be created.

Figure 4-4 LQN model for the BSS

Figure 4-4 shows the LQN model systematically derived from the two sequence diagrams shown above. As mentioned above, the performance requirements are to meet a 1-second-deadline for both the Access Control scenario and the Acquire/Store Video scenario, with a 95% confidence level. In the LQN model, these requirements are translated to requiring the service time of the Video Controller task (also called its cycle time) and the response time of the User task to be less than 1 second with probability of 95%.

4.2 Performance Diagnosis and Improvement by Human Expert

This section will show how a human performance expert might diagnose and resolve the performance issues in this BSS application. Observations on this procedure will lead to our rule-based approach.

4.2.1 Performance Estimation for Base Design

The base design of the BSS system has a configuration of a single thread for all the tasks except Database, Disk and networks, whose thread multiplicities are specified as a requirement in the design models. Table 4-1 shows the performance estimation results obtained by solving the derived LQN model using the LQN simulation solver for 10 to 40 cameras. From the results we can see that the response time for the Access Control scenario (i.e. the column named User in the table) satisfies its performance requires and meets the 1 second deadline with less than 5% miss probability. Increasing the number of cameras has a negligible impact on this delay. However, the Video Acquire/Store scenario can only support 20 cameras to meet its deadline with 95 % probability.

Table 4-1 Simulation results for BSS base design

Ncam	Average Response Time		Normalized Utilizations				Prob of Missing Deadline	
	Cycle (sec)	User (sec)	AcqProc	Buffer	StoreProc	ApplicationCPU	Cycle	RUser
10	0.327	0.127	0.960	0.9998	0.582	0.549	0	0.031
20	0.655	0.138	0.963	0.9999	0.582	0.545	0.0007	0.036
30	0.983	0.133	0.964	0.9999	0.582	0.544	0.4196	0.038
40	1.310	0.129	0.965	0.9999	0.582	0.544	0.9962	0.034

The estimations of normalized utilization show that both AcquireProc and Buffer are saturated, with utilization higher than 95%. From our experience, we can easily

identify that `Buffer` is the root cause of the saturation, because it is an indirect server of `AcquireProc`. As a result of the client-server relationship, `AcquireProc` is saturated because of being blocked waiting to acquire a buffer.

In layered systems with synchronous (blocking) relationships between clients and servers, a saturated server causes its clients to wait, and tends to saturate them in turn. Thus the cause of saturation is at the lower layer, transmitted by pushback to the higher layer. This pushback may be transmitted through several layers, and always the lowest saturated task is the “root” or causal bottleneck. This may be at the bottom layer (a processor or device) but it may equally be at a higher layer, in the case of a task which distributes its waiting among several servers all of which are not blocked. A bottleneck at a higher layer is a software bottleneck. Remember that “server” in this discussion includes processors and also pseudo-tasks like the one representing the buffer pool in the BSS example.

4.2.2 Bottleneck Mitigation by Configurational Change

Adding resources to bottlenecks is the most straightforward mitigation. It is named *cloning* in [Tregunno03], and is interpreted as increasing task or processor multiplicity in the LQN model. In design models, this will be mapped to multithreading for software resources and multiprocessors or multicore processors for hardware resources. Addressing the root bottleneck will result in the best benefit for performance. For example, in this case, only increasing the multiplicity of task `Buffer` will solve the bottleneck. Increasing `AcquireProc` without adding buffers will not gain much benefit.

Table 4-2 shows the performance estimation for the system with 40 cameras under different configurations of `Buffers`. The performance improvement due to multiple buffers is obvious. The probability of missing the cycle time deadline drops greatly, from 99% for 1 buffer to 9.35% for 10 buffers, but the requirement of a 5% probability for

missing the deadline is still not achieved. We can see that now the root bottleneck is pushed to a lower layer task `StoreProc`.

Table 4-2 BSS LQN results for using multiple Buffers (40 cameras)

NBuf	Average Response Time		Normalized Utilizations				Prob of Missing Deadline	
	Cycle (sec)	User (sec)	AcqProc	Buffer	StoreProc	ApplicationCPU	Cycle	RUser
1	1.309	0.137	0.965	0.9999	0.583	0.544	0.9961	0.034
2	1.016	0.132	0.975	0.8762	0.800	0.702	0.5503	0.032
3	0.941	0.132	0.980	0.8235	0.893	0.756	0.2506	0.036
4	0.911	0.131	0.983	0.8042	0.936	0.782	0.1597	0.032
7	0.879	0.132	0.986	0.8136	0.984	0.810	0.0948	0.033
10	0.872	0.129	0.987	0.8437	0.995	0.817	0.0935	0.034

Therefore, the second resolution step is to clone the `StoreProc` task. Table 4-3 shows the results for the case of 40 cameras with 4 buffers. We can see that with 2 `StoreProc` threads, the probability of missing the deadline has dropped to a satisfactory level. The system capacity is now above 40 cameras, about double that for the base design.

Table 4-3 BSS LQN results for multi-threading StoreProc (40 cameras, 4 Buffers)

No. of Store Proc	Average Response Time		Normalized Utilizations				Prob of Missing Deadline	
	Cycle (sec)	User (sec)	AcqProc	Buffer	StoreProc	ApplicationCPU	Cycle	RUser
1	0.911	0.131	0.983	0.8042	0.936	0.782	0.1597	0.032
2	0.756	0.137	0.946	0.5805	0.616	0.940	0.0022	0.035
3	0.743	0.139	0.932	0.5484	0.441	0.956	0.0015	0.039

Following the same root bottleneck identification pattern, a new root bottleneck is identified as `ApplicationCPU`, which is a processor instead of a task. So far, the root bottleneck has been pushed from software resources to hardware resources. This bottleneck can be mitigated by using a multi-processor, giving the results in Table 4-4.

The simulation results show that 2 ApplicationCPUs are enough for solving the hardware bottleneck here. Using a dual processor is a typical configuration strategy. The system capacity is now 50 cameras, with 4 Buffers, 2 StoreProc threads and 2 ApplicationCPUs. This is 2.5 times higher than the base case and achieves our initial goal of system capacity.

**Table 4-4 BSS LQN results for using multiple processors for ApplicationCPU
(40 cameras, 4 Buffers, 2 StoreProc threads)**

No. of CPU	Average Response Time		Normalized Utilizations				Prob of Missing Deadline	
	Cycle (sec)	User (sec)	AcqProc	Buffer	StoreProc	ApplicationCPU	Cycle	RUser
1	0.756	0.137	0.946	0.5805	0.616	0.94	0.0022	0.035
2	0.648	0.127	0.995	0.6111	0.653	0.549	0	0.035
3	0.644	0.128	0.997	0.6105	0.652	0.368	0	0.033

The point has been reached where, except for the reference task VideoController, there is only one saturated resource in the system, namely AcquireProc. We may consider it as the root bottleneck. However, LQN results show that cloning it gives no improvement. In fact, its queue contains at most one request at any time and never grows. The performance is not limited by a lack of resources now, but by design limitations.

4.2.3 Reduce Length of Execution Path by Design Change

As mentioned before, there are two saturated tasks in the model, the reference task VideoController, and AcquireProc. A reference task in a closed model drives the system by generating workload, and usually represents the behaviour of an external client. Its normalized utilization is always 1, because it is always blocked by all of the services in the scenario.

Here VideoController is similar to an external client, although it is a part of the system. The VideoController has to wait for the message returned from AcquireProc before generating the next polling call. The call from the VideoController to AcquireProc is synchronous, and all of the work of AcquireProc is finished in its first phase. Therefore, only one instance of AcquireProc can be activated at any time. The system suffers from too much serialization, and the system capacity is limited by the duration of the scenario which polls one video camera.

To solve this problem, a change in the system design is required. A key point is to enable concurrent activations of the AcquireProc task by multi-threading the process. A solution is to move the calls made by AcquireProc for allocating and using the buffer into its second phase, and making an early reply to VideoController. Then VideoController can generate its next polling call earlier. This change introduces more asynchronous behavior into the system. Figure 4-5 shows the change presented in the BSS LQN model. The invocation frequency between task AcquireProc and BufferManager is changed from (1, 0) to (0, 1).

Figure 4-5 Second phase change on BSS LQN model

After introducing more asynchronous behavior into the system, the software and hardware bottlenecks appear again. Following the same routine of identifying root bottleneck and cloning mitigation repeatedly as before, the system performance can be improved dramatically. This process is formalized into a procedure called Configuration

Mitigation or Configuration Tuning later in this thesis. During the tuning, the bottleneck moves around within the system. For example, the bottleneck may move to task `AcquireProc` or `StoreProc` as well as `ApplicationCPU`.

Table 4-5 shows some system performance results under different configurations. Here we aim for a capacity of 100 cameras and increase the number of threads for `Buffer`, `AcquireProc`, `StoreProc` and the number of `ApplicationCPU` step by step. Finally, with 3 `AcquireProc`, 6 `StoreProc` threads and 3 `ApplicationCPU`, we manage to achieve the 1-second-deadline for the cycle time for the case with 100 cameras with a probability of 99.95%. This capacity is 5 times higher than the base design, and twice the capacity before changing the design.

Table 4-5 BSS LQN results for with more concurrency case (100 cameras, 10 Buffers)

No. of Acquire Proc	No. of Buffer	No. of Store Proc	No. of Application CPU	Average Response Time		Normalized Utilizations				Prob of Missing Deadline	
				Cycle (sec)	User (ms)	Acquire Proc	Buffer	Store Proc	Applica CPU	Cycle	RUser
2	4	2	2	1.250	0.133	0.988	0.9233	0.886	0.710	0.9995	0.0332
2	10	6	3	0.837	0.132	0.988	0.6892	0.751	0.707	0.0057	0.0307
3	10	6	3	0.768	0.134	0.983	0.8948	0.910	0.769	0.0005	0.0352

The results also show that `AcquireProc` and `StoreProc` tasks are saturated again. Therefore, we expect that there is more room for improving the capacity by further tuning the system configuration.

4.2.4 Performance Improvement and Feedback to Design

The exploration described above is carried out in the space of LQN models. Table 4-6 lists the most significant changes that have been done and their impact on performance.

Table 4-6 Significant changes and performance results for BSS system

Mitigation	Changes	Performance
Base design	N/A	Support 20 cameras for 1 second deadline
Cloning software bottleneck	4 Buffers and 2 StoreProc threads	Support 40 cameras Doubled base design capacity
Cloning hardware bottleneck	Dual application CPU	Support 50 cameras Improvement 150% of base design
Introduce more concurrency	Moving first phase call to second phase	Support 100 cameras Improvement 400% of base design

As this example shows, some kinds of feedback can be presented in the design model just by using tag values defined in SPT Profile, as shown in Figure 4-6. However, others require deeper changes to be made by the designers. For instance, multithreading may require changes to synchronize threads or to maintain consistency in data shared by the threads; partitioning an object into two concurrent objects would require new classes. Techniques to support these changes, possibly based on patterns to solve typical problems that arise, will be needed.

Figure 4-6 Modified sequence diagram for Acquire/Store scenario

4.3 What Do We Learn From the Example?

This example is a good study case for us to understand how a human expert works on models to systematically improve performance of a software application. From this example, we have the following observations discussed in this section.

4.3.1 Two Causes of Performance Problems

In the BSS example, we saw two types of performance problems: bottleneck and long path.

Starting from the base design, the performance problem is shown as resource saturation, either at a software component, such as Buffer, StoreProc, or AcqProc, or at a processor. The problem is caused by a lack of resources at these saturation spots.

Before making the change for introducing a second phase operation, the system design reaches a point where no resource is saturated other than the very top one, VideoController, and the response time still cannot satisfy the requirement. This is because the execution demand for the entire scenario is too heavy, i.e. there is too much work to do within a cycle in the Video Surveillance scenario.

Figure 4-7 Semantic network of performance problem root cause

Figure 4-7 shows the causes, symptoms, and relationships between these two types of problems. This observation is the foundation of our rules for performance problem localization and mitigation. Section 5.3 gives more detailed discussion on these two types of problems.

4.3.2 A Tower Pattern for Root Bottleneck Identification

A bottleneck exists in a system when there are saturated resources either in hardware or software. However, a saturated resource is not necessarily a root bottleneck. It is quite common that saturation occurs in multiple resources in a system at the same time. Only changes made to the root bottleneck is helpful. In this research, the key point, i.e. the root bottleneck, is defined as the root cause of saturation in a software system. Mitigation on a root bottleneck results in a greater benefit than mitigation on other saturated resources.

Figure 4-8 Bottleneck tower pattern in BSS example

A tower pattern can be clearly identified and has been repeatedly used in this example for locating the real bottleneck. The name of the pattern describes the saturation structure in a LQN model, i.e. saturation usually appears in a series of client-server tasks, with the bottom one being the root bottleneck, and the upper layers are saturated due to being blocked by their underlying services.

Figure 4-8 shows the tower pattern in the BSS example. In the base design case, there were 3 software resources being saturated at the same time, VideoController, AquirProc and Buffer. The lowest task Buffer was identified as the root bottleneck, and the other two are saturated because of waiting for Buffer. Because of the position of the real bottleneck is always at the bottom of the tower, it is named a *Root Bottleneck* in this thesis. This observation leads to the bottleneck identification rule.

4.3.3 Iterative Improvement Process and a Holistic Improvement Principle

The example shows that the performance improvement is done in an iterative process which alternates between a diagnosis stage and a mitigation stage, as shown in Figure 4-9. There are two possible working paths for mitigation action, cloning the bottleneck resource when a root bottleneck can be identified, or other solutions which usually require design changes when no bottleneck can be identified.

Figure 4-9 Iterative process of performance diagnosis and improvement

Mitigation on one bottleneck usually is just pushing the bottleneck away to other resources, and could even result in multiple new bottlenecks. In the BSS example, the StoreProc task, which was not saturated before, becomes the bottleneck after using multiple Buffers.

A notable result here is that the software change could only be identified as beneficial, after the thread configuration and buffer questions had been resolved. For example, the second phase operation introduced between task VideoController and task AcquireProc would not provide much improvement if multiplicities of underlying tasks, such as StoreProc and ApplicationCPU, had not been increased accordingly, because these tasks will become bottlenecks and will block the entire scenario. Therefore, we have evidence for a general principle for software design improvement:

Holistic Improvement Principle: *Software design improvements can only be evaluated in the context of the best possible deployment and configuration alternatives.*

On the other hand, it also means a design change is only necessary when the current design has been evaluated under its best possible configuration and deployment. For instance, if we change the software design at the very beginning before tuning the base design, it may not make great improvement on performance, because the buffer is limiting the throughputs. More concurrency can only result in a heavier workload on the buffer which is already saturated.

4.3.4 Some Mitigation Strategies

The example uses cloning, i.e., increasing resource multiplicity levels, to address bottlenecks including software resource and hardware resource. This is a straightforward and efficient solution. It is under the assumption of no constraint on the availability of resources though. This assumption may not be true in many other cases. For example, it is common to see a system with constraints on thread or buffer pool sizes due to memory limitations, or with hardware limitation from project budget considerations. In that case, the mitigation stage will also go to the lower path in Figure 4-9, which is seeking other design solutions.

There is no single rule to follow for design changes, since they are usually project specific and usually require understanding of business scenarios. But there are general principles for guidance, such as Connie Smith's principles as introduced in Chapter 2, and some common practices, such as caching, batching, prefetching, asynchronous, etc. In this example, a partial asynchronous execution path is applied by using a second phase operation and shows great efficiency.

4.3.5 What Can Performance Diagnosis Tell

First of all, the diagnosis and improvement results in the performance model tell whether performance can be improved, what changes can be done, and what the performance would be under these changes. Table 4-6 in section 4.2.4 listed the performance gain for each step of significant changes done in the BSS example.

Secondly, the requirements for resource capacities, including the threading level of software components and the multiplicity of hardware resources, can be obtained by solving the LQN model under different configurations. For instance, the LQN model tells a dual CPU was enough for the BSS example before changing the interaction pattern between VideoController and AcquireProc. After the interaction pattern was changed from synchronous to asynchronous, the requirement for CPUs was increased to 3. Extra CPUs would be a waste.

Finally, if satisfaction of the performance requirements is impossible at all for current architecture or technology, it can be found out quickly at the beginning without taking much analysis. In the BSS example, the performance requirement of supporting 50 cameras while keeping two scenarios meet the 1 second deadline apparently could not be satisfied without multithreading technology as tested by the base case. Therefore, the decision to adopting a multithreading architecture can be guided by a one-step

performance test. If a new architecture or technology is not allowed, the requirement should be re-examined.

4.3.6 Potential for Rules

This example demonstrated how performance diagnosis process is time-consuming, and with tedious data mining tasks. The entire process took many repetitions and more than 100 model solutions in order to try out different configurations. A performance engineer has to go through a huge amount of estimation data to locate root bottlenecks and propose mitigations. It would be attractive if the process or part of it could be automated.

From the investigation reported in the previous sections, there are clearly some rules that can be followed, such as the tower pattern for root bottleneck identification, the diagnosis-mitigation iteration process, and the holistic improvement principle on evaluating design under improved configuration and deployment. Therefore, this gives the potential for using a rule-based approach to automate the process of software performance diagnosis and improvement.

CHAPTER 5 FORMALIZATION OF PERFORMANCE EXPERT KNOWLEDGE

Many previous approaches improve software performance in ad-hoc ways. Most of the software changes are done manually. The performance principles are general and need performance expertise and deep understanding of the specific software behavior to be applied for individual software projects.

Given the detailed walk through in Chapter 4 on how human experts work on performance improvement and on the general principles provided by Connie Smith, we have gathered some performance expert knowledge. The next step is to organize the knowledge into formal representations as the analysis stage in Figure 3-13 and to expand it to be more complete. In this stage, project-specific techniques, processes and approaches used by human expert will be generalized to abstract rules that can be followed by many projects. The formal representations used in this chapter include semantic networks and abstract rules in premises-conclusion format.

5.1 Understand Performance Requirements

The very first step of performance analysis is to determine whether performance requirements can be satisfied by the current design. Performance requirements include at least two aspects: a target workload and a required performance metric.

Software performance is usually evaluated using two metrics: response time and throughput. Response time is calculated from the time that the client sends out a service request to the moment that a response message is received by the client. Throughput is the number of requests fulfilled in a time unit. The metrics are associated with some target objects in the system, such as a client object, or a service entrance point.

In performance engineering, two types of workload are defined:

- An open workload means clients come in at a given arrival rate, and leave the system upon service complete.
- A closed workload means a fixed number of clients reside in the system. Clients do not leave upon finishing a request, but instead, reenter the system after some thinking time between requests.

Both types of workload may appear in the same system. For instance, in the BSS example, the Video Surveillance scenario has a closed workload with one client, and the User Access Control scenario has an open workload, with an arrival rate of 1 per 2 second.

(a) performance requirement on throughput

(b) performance requirement on response time

Figure 5-1 Performance requirements

Both response time and throughput depend on the workload. Figure 5-1 shows the common performance plots for the two performance metrics versus workload in a closed system with N clients. Performance evaluation is only meaningful under a specific workload. In the figures, N_r represents a specific targeting workload. f_r and s_r represent the required values for throughput and response time respectively. The curve represents the estimates of performance for a given system. Better performance is associated with a

shorter response time or a higher throughput. Any requirement in the shaded area can be satisfied by the system. Open systems have similar considerations.

Sometimes, performance requirements may come with other limitation factors combined with the two metrics. For example, a designer may want to reach certain throughput and also keep the CPU utilization under 70%.

Figure 5-2 Semantic network for performance requirements

Given the above analysis, Figure 5-2 describes the knowledge on performance requirements by a semantic network. Take the requirement in BSS as an example: the requirement for Video Surveillance scenario has a target workload of 50 cameras, a

metric of response time for less than 1 second, with 95% probability, and is evaluated on one cycle of the VideoController object.

5.2 Resource Interaction Model

Figure 5-3 shows a resource interaction model which provides a context for performance diagnosis and mitigations on a computer system.

Figure 5-3 Semantic network for resource interaction model

A system consists of resources and interactions.

A resource has a capacity and a multiplicity that describe its availability. The multiplicity is number of units of the resource configured at run time. The capacity sets the limit on maximum multiplicity available to the resource under a specific design. A resource can be either a hardware entity or a software entity. A hardware resource has a speed and a speed constraint that defines the highest speed available for the hardware resource. A

software resource has an execution demand, which is the execution time required by the software resource on a set of given hardware resources.

An interaction describes the invocation relationship between two software resources named caller and callee. Frequency describes the intensity of an invocation (invocations/second). Pattern describes the resource contention between interactions:

- Blocking interaction: caller resource is being hold during entire or part of the invocation, which includes the execution time on callee resource and any necessary network delay.
- Non-Blocking interaction: caller resource is released immediately after the invocation message is sent out.

Deployment of a software resource to a hardware resource can be seen as creating a special type of blocking interaction between the software resource and the hardware resource. It is presented separately since it will be treated differently in the diagnosis and mitigation discussed in this thesis.

In later discussions in this chapter, the following notations are used for simplicity:

$Rs_{caller} \xrightarrow{B} Rs_{callee}$: Blocking interaction from caller software resource Rs_{caller} to callee software resource Rs_{callee}

$Rs_{caller} \xrightarrow{NB} Rs_{callee}$: Non-Blocking interaction from caller resource Rs_{caller} to callee resource Rs_{callee}

$Rs \xrightarrow{Deploy} Rh$: Software resource Rs is deployed on hardware resource Rh

5.3 Localization of Problems

Next, we look at how to pinpoint a root cause for performance problems in a system. In general there are two kinds of root cause results for a failure to meet the performance target:

- **Long path:** The critical path of the program is too long which results in the entire business scenario failing to meet the target.
- **Bottleneck:** A saturated resource introduces waiting into the response, even if it could be performed in time on an unsaturated system. While a long path is a symptom, the root cause is the saturated resource.

These two kinds of problems are often coupled together, since waiting at a bottleneck can cause a long path and a long path in the holding time of a resource can cause its saturation. On the other hand, each type of problem has its own causes and symptoms. Figure 5-4, which has been shown in section 4.3.1 and is reproduced here, shows the semantic network that captures the causes, symptoms and relationships of the two types of performance problems. These factors decide the identification rules and mitigation strategies that can be used for each type of problem.

Figure 5-4 Semantic network for performance problem root causes

5.3.1 Bottleneck Identification and Root Bottleneck

A bottleneck always has the symptom of saturation of some resource. It can be in either hardware (a processor, storage or network element) or software (a critical section, lock, or pool of buffers or process threads). It can be caused by lack of resources, such as a limitation on threading pool size or processor availability, or by long holding time of the resource.

When there is a bottleneck in the system, it is common to see multiple saturated resources simultaneously. In layered resource architectures [Woodside01], there is a single very general bottleneck pattern discussed in detail in [Franks06] as a “software bottleneck”. This covers essentially all other diagnostic patterns related to resource constraints, including the “one lane bridge” antipattern of Smith and Williams [Smith00]. The software bottleneck pattern accounts for the fact that congestion propagates upward through higher layers of resources, creating a connected complex of resources. The actual causal bottleneck is the lowest saturated resource in such a pattern. The diagnostic rules for a bottleneck below are based on this general layered bottleneck pattern.

Figure 5-5 Bottleneck tower pattern

In this thesis, this general bottleneck pattern is named a *Tower Pattern*, which has been seen in the BSS example. Figure 5-5 shows a common structure what a tower pattern looks like in a layered system. The figure borrows graphical notations from LQN to present the interaction relationship between resources. A *Tower Pattern* consists of a chain of saturated resources, shown as shaded rectangles, connected by blocking interactions, shown as lines with solid arrows in the figure. In the pattern, the bottom layer saturated resource is the root cause of the saturation chain, which is named **root bottleneck** in this thesis, shown with a bold outline in Figure 5-5. Here, the resource *Server1* is saturated. As a result, the shaded resources above, *Broker* and *Client*, are saturated as well because they are blocked by *Server1*.

The root bottleneck is easy to recognize when a single tower pattern exists in the system. Sometimes, there may be multiple tower patterns raised by the same root bottleneck, as shown in Figure 5-6. An even more complex situation is when multiple root bottlenecks exist in the system. Figure 5-7 shows an example. Both *Server1* and *Service4* are root bottlenecks in this case, since they are both at the bottom layer in their respective tower pattern.

Figure 5-6 Multiple tower pattern with one root bottleneck

Figure 5-7 Multiple root bottlenecks

Based on the above study, we have the following abstract rules:

Rule 5-1 Bottleneck Existence Rule

Premises:

System Sys has one or more saturated resources.

Conclusion:

Sys has at least one bottleneck.

Rule 5-2 Software Client-Server Rule

Premises:

Interaction $Rs_1 \xrightarrow{B} Rs_2$ exists.

Conclusion:

Software resource Rs_2 is a server of software resource Rs_1 .

Rule 5-3 S-H Client-Server Rule**Premises:**

Deployment $R_s \xrightarrow{\text{Deploy}} R_h$ exists

Conclusion:

Hardware resource R_h is a server of software resource R_s .

Rule 5-4 Client-Server Transitive Rule**Premises:**

Resource R_2 is a server of resource R_1 , AND

Resource R_3 is a server of resource R_2 .

Conclusion:

Resource R_3 is a server of resource R_1 .

Rule 5-5 Root Bottleneck Rule**Premises:**

Resource R is saturated, AND

(R does not have any server, OR

R has one or more servers AND no server of R is saturated).

Conclusion:

Resource R is a root bottleneck.

Note: In Rule 5-4 and Rule 5-5, a resource can be either a hardware entity or a software entity.

5.3.2 Long Path Identification

As shown in Figure 5-4, a long path can be a result of:

- heavy demand, meaning a long execution time for the operations performed during the holding time of the resource (this will be called a Pure Long Path).
- a long waiting time for some nested resource request, at a congested resource.

A long path describes a scenario, instead of one resource. The operations during the scenario contribute to the resource holding time. One criterion for a long path may be exceeding an “expected value” of delay, which may be available to us from some facts. It may be expected, or required, or a convenient target threshold generated by a strategic rule.

Rule 5-6 Heavy Demand Rule

Premises:

Scenario S_c has response time r , AND

r is longer than expected value r_e , AND

All resources that are required by S_c are not saturated.

Conclusion:

Scenario S_c has a long path problem.

5.4 Mitigation Techniques for Performance Problems

In the last section, we have classified performance problems into two categories based on their symptoms: *Bottleneck* and *Long Path*. This section considers how the two sets of problems are commonly handled by a performance expert, and develops abstract rules for

the mitigation techniques. Figure 5-8 shows some common solutions for the two types of performance root causes.

Figure 5-8 Performance problem mitigations

5.4.1 Bottleneck Mitigation

Since a bottleneck has a symptom of saturation, the mitigation for bottleneck is to reduce the utilization of the bottleneck resource. As discussed earlier, acting on the root bottleneck will result in the best benefit.

The utilization of the bottleneck resource can be reduced by these mitigations:

- *add resources*: increasing multiplicity of root bottleneck resource
- *faster device*: increasing the speed of the processor resource that the operations use during the holding time of the root bottleneck resource
- *call less*: avoid requesting the root bottleneck resource as much as possible in scenarios
- *reduce path length*: reducing the holding time of the root bottleneck resource

Among the above 4 mitigation strategies, the first two are configurational from the designer's point of view, i.e. they can be implemented without changing the design, such

as increasing the threading pool size or updating the server hardware. The latter two usually requires redesign of system behaviors. Call less is actually also a path length reduction strategy from the client's point of view.

There are constraints which can prevent the above 4 mitigations from being applied on a system:

- Capacity limit: limitation on maximum multiplicity of resources. It can be a limitation on a thread pool size due to memory constraints or availability of hardware resources due to financial constraints.
- Upgradeability: whether a hardware device with faster speed is available for the type of resource required.
- Critical path: whether an operation is using a resource can be eliminated from the scenario that fulfills minimum business functional requirements.

Considering the above constraints, we have the following abstract rules for bottleneck mitigation:

Rule 5-7 Bottleneck Multiplicity Rule

Premises:

A root bottleneck R_B is identified, AND
 R_B has unused capacity.

Conclusion:

Increase multiplicity of R_B

Rule 5-8 Processor Upgrade Rule

Premises:

A root bottleneck R_b is identified, AND

R_b is a software resource, AND

$R_b \xrightarrow{Deploy} Rh$, AND

Rh is upgradeable.

Conclusion:

Upgrade Rh .

Rule 5-9 Software Bottleneck Bypass Rule

Premises:

A root bottleneck R_b is identified, AND

R_b is a software resource, AND

$R_C \xrightarrow{B} R_b$ exists, AND

R_b is not in critical path of R_C .

Conclusion:

Reduce call from R_C to R_b .

Rule 5-10 Redeploy Rule

Premises:

A root bottleneck R_b is identified, AND

R_b is a hardware resource, AND

More than one software resources are deployed on R_b , AND

An unsaturated hardware Rh_{free} of the same type exists.

Conclusion:

Move a software resource from R_B to Rh_{free} .

Rule 5-11 Software Bottleneck Long Path Rule**Premises:**

A root bottleneck R_B is identified, AND
 R_B is a software resource.

Conclusion:

Reduce path length of scenarios invoked by R_B .

Rule 5-12 Hardware Bottleneck Long Path Rule**Premises:**

A root bottleneck R_B is identified, AND
 R_B is a hardware resource.

Conclusion:

Reduce path length of operations executed on R_B .

In rule 5-11 and rule 5-12, the bottleneck problems are actually transformed into long path problems which may have solutions from path length reduction strategies, which will be discussed in the following section. These rules are also long path problem identification rules.

5.4.2 Long Path Mitigation

A long path problem has a symptom of excessive duration of some service operation. The path is defined by a scenario or sub-scenario consisting of a sequence of operations, and requires changes to those operations or the resources they use. The following approaches, also shown in Figure 5-8, can be used for shortening the long path:

- *shrinkExec*: reduce the budgeted execution time of operations. Approaches have been identified by Smith and Williams [Smith02] as principles such as Locality, which avoids or reduces boundary-crossing overhead, and Fixing of relationships (also called early binding). This can also be achieved by replacement of one vendor's solution by another's.
- *batching*: combine some operations together. We will consider batching of operations requested across a network, to reduce the contribution of network latency to the long path. This can be viewed as a particular case of Locality in [Smith02].
- *fast path*: provide simpler faster processing for special cases.
- *caching* or *prefetching*: can reduce latency for data access.
- *parallelism*: divide a path into parts in parallel.
- *moreAsync*: move some operations into a second phase of the service, i.e. after the reply is sent to the client. This executes the remaining service operations (typically delayed writes, logging, and resource cleanup, but sometimes parts of the service function) in parallel with the continuation of the client, and reduces the service path from the client point of view.
- *removing nested bottleneck*: one of the delays on the long path may be an operation with large contention delays, because it uses a bottlenecked resource.

The last approach shifts the focus from the long path to some nested bottleneck. If a bottleneck is found in the long path scenario, removing it is usually more effective than other long path mitigation strategies.

All long path mitigation strategies except the last one imply design changes. Again, like bottleneck mitigation strategies, not all resolutions are applicable to a given system. In fact, there are even more kinds of constraints that may prevent PB from applying a certain long path mitigation change. On the other hand, it is hard to tell which strategy works better than the others in a particular situation. The approach in this thesis is to try all of them under a set of constraints, and to use ranking mechanisms to rule out less effective strategies. Therefore, we have the following abstract rules for long path mitigation:

Rule 5-13 Path Reduction Tryout Rule

Premises:

A scenario *Sc* is identified as a long path.

Conclusion:

Try *shrinkExec* in *Sc* and evaluate,
 Try *batching* in *Sc* and evaluate,
 Try *fastpath* in *Sc* and evaluate,
 Try *caching* in *Sc* and evaluate,
 Try *prefetching* in *Sc* and evaluate,
 Try *parallelism* in *Sc* and evaluate,
 Try *moreAsync* in *Sc* and evaluate.

Rule 5-13 will result in a set of evaluations for tentative solutions. Ranking mechanisms are needed to provide guidance for user choice or automation implementation. Ranking can be based on performance metrics, cost of changes in terms of money or effort, and

effectiveness which combines the previous two. Some criteria for ranking are introduced in the strategy chapter.

5.5 Configuration Changes vs. Design Changes

Our goal is to improve the performance properties of the design, and find a design that meets requirements. However performance also depends on the run-time configuration (meaning the deployment and the run-time parameters of the software). As already noted, it is fundamental that each design should be evaluated for the best configuration (within constraints). In the diagnosis and improvement process, human experts improve the configuration for each design before making a decision, as illustrated in Figure 5-9. Here it will be done automatically.

Figure 5-9 Configuration improvement sub-problem

Formally, we can describe the goal of the rule-based system as a search for feasible values of **D** in:

$$\mathbf{D}: \mathbf{P}(\mathbf{D}, \mathbf{C}) < \mathbf{R}, \mathbf{C} \in \mathcal{C}, \mathbf{D} \in \mathcal{D}$$

where **D** is the set of design choice parameters, restricted to values in the feasible set $\mathcal{D}$, **C** is the set of configuration parameters, constrained to values in the feasible set $\mathcal{C}$, and **P** is the performance evaluation vector which is required to be less than the required values **R** (direction has been taken into consideration of the evaluation, e.g., in the case of

evaluation using throughput, a negative direction will be applied on the evaluation result value and the required value, which makes the less than condition still valid).

In the performance model, the set **C** of configuration properties includes parameters related to the hardware platform (i.e. multiplicity and speed of devices) and the parameters of deployed processes (size of thread and buffer pools, process priority, allocation of processes to processors). Design properties are represented by a set **D** of surrogate model properties which we can call the *design state*.

A change in configuration can be achieved by changing the configuration parameter set from **C** to **C'**, and is named a *configuration transformation*, noted by $T_{C'}$. A change in design can be achieved by changing the design parameters from **D** to **D'**, and is named a *design transformation*, noted by $T_{D'}$.

This leads to our abstract rule on mitigation control:

Rule 5-14 Configuration Priority Rule

Premises:

A root bottleneck B_r is identified, AND

A set of configuration changes T_{CB_r} are applicable, AND

A set of design changes T_{DB_r} are applicable.

Conclusion:

T_{CB_r} has higher priority than T_{DB_r} .

A design can be improved by a series of design changes and configuration changes before reaching a satisfactory level. The differences from the starting design **D** and configuration **C** to the latest design **D''** and configuration **C''** are noted by $\Delta\mathbf{D}$ and $\Delta\mathbf{C}$.

We will also define cost-related weights $\mathbf{W}$ for the changes $\Delta\mathbf{D}$ and $\Delta\mathbf{C}$, which may be constrained by a maximum total weight V :

$$W(\Delta\mathbf{C}, \Delta\mathbf{D}) < V$$

A related problem is to improve performance subject to the weight constraint, which will be discussed in detail in Chapter 6.

5.6 Performance Model Surrogates for Design Properties

If the rules succeed in identifying an improved performance model, the problem remains as to how to translate the result into a change to the software design.

The approach in this thesis is based on identifying design changes with model changes. This identification has some limitations (there are some design changes that have no direct representations, such as caching or prefetching). On the other hand, it appears to be possible to represent many or most important configuration attributes. The changes that have been investigated here are (labeled for configuration change or design change):

- (Configuration) *multiplicity*: increase the multiplicity of a processor or of a task, (the thread pool size for a process).
- (Configuration) *redployment*: move a task from a saturated processor to a lightly utilized one.
- (Design) *shrinkExec*: reduce the execution-time budget of an entry.
- (Design) *moreAsync*: move a fraction of phase-1 work (host demand and service calls) of an entry into phase 2.

- (Design) *partitioning*: split a heavy task with multiple entries into multiple smaller tasks.
- (Design) *batching*: replace a repeated call that includes a network latency, by a single call to convey all the requests.

Design properties relate to the specification and include concurrency (partitioning of operations between processes, allocation of operations to processes, and postponed server operations), and path- properties such as parallelism, and execution and service demands. Some of these properties are represented by parameters (called here surrogates), indicated by * in Table 5-1.

Table 5-1 Software Design Properties represented in the Performance Model

Design Property	Model Property in set D
*Execution-time	*Entry service time
*Postponed server operations	*Second-phase execution and calls in an entry
*Batching of operations	*Frequency/size parameters of the operations
Partitioning/allocation of operations between processes	Operations are aggregated into entries
*Asynchronous interactions	*Asynchronous interactions
Prefetching	Parallel fetch and execution
Asynchronous RPC	RPC and client execution in parallel
Parallel execution	Parallel paths within an entry

In PB, the amount of change is constrained for practical reasons to a limit Δp for each parameter p . These change limits define the boundaries of the feasible set $\mathcal{D}$ defined in Section 5.4. The value of Δp is chosen as a plausible target for planning purposes, and the actual design change (carried out manually) may achieve a greater or lesser amount. The expected effort required to make the change Δp is represented by a weight which may be in units of developer-days of effort. Assigning a value which is large will discourage the rules from selecting that change, on a cost-effectiveness basis.

Changes in surrogate properties have to be mapped back to software design changes manually. In most cases the search described here suggests what should be changed, and in what way, but not how to do it. For example, reduced execution can be approached in a number of ways, some of which are suggested by the principles stated by Smith and Williams [Smith00].

MoreAsync

MoreAsync increases concurrency by postponing some server operations until after a reply to a service request, to unblock the requester and allow it to proceed in parallel. In the LQN model, it is represented by second phase operations. The server operation has a CPU demand S and demands for lower-layer service operations, Y_j for operation j . The maximum fraction of each phase-1 parameter, which can be moved to second phase, is specified by $\Delta_{\text{ma}} S$, $\Delta_{\text{ma}} Y_j$.

At a bottleneck resource, the second phase is introduced *to its servers* to reduce its blocking time. Figure 5-10 shows how *moreAsync* modifies behaviour described in UML from a designer's point of view.

Figure 5-10 Design change recommended by automatic diagnosis: *moreAsync* operation on DB1 for treating App1

CHAPTER 6 STRATEGIES FOR THE PERFORMANCE BOOSTER (PB) FRAMEWORK

This chapter describes the strategies that have been used to govern the application of the rules described in Chapter 5, and their relationship to the specific performance modeling formalism LQN and the Jess rule engine. The framework presented in this thesis is named Performance Booster (PB), which is built in the hope of being able to effectively help designers develop high performance software applications.

Although the framework is in a prototype stage, it has already demonstrated the power and efficiency of its rules. Most important, the approach provides a framework to which new rules can be added easily. It enables a performance expert to focus on studying performance patterns and properties, without spending much time on repeated data manipulations. On the other hand, it enables performance expert knowledge to be easily reused, and facilitates software designers understanding of performance issues and implementation of design improvements.

6.1 Overview of PB Search Strategy

Section 1.2 described the information flow of PB. It begins with requirements for a set of response times or throughputs of interest. The goal is to meet these requirements. The framework PB combines inference using the Jess rule engine with evaluation by the LQNS performance model solver. The rule-based framework manages steps 2 to 5 of the process in Figure 1-1, and in particular, steps 3 and 4 are completely controlled by the rule engine.

To begin with, the performance model is solved, giving values for the performance measures such as response times or throughputs. Diagnostic rules are applied to find:

- a list of saturated resources, and
- a list of critical response times (initially, those that violate their requirements).

Configuration improvement is applied first according to our Configuration Priority Rule (Rule 5-13). In an LQN model, configuration change might increase the multiplicity of any root bottleneck. This is a resource optimization to reduce the impact of saturated resources. If it succeeds (giving a performance evaluation that meets the requirements), the initial design is satisfactory and the search terminates at once.

If it does not succeed, a search is made for alternative designs (each one in turn improved for configuration), as described below. The result is a set of designs which are children of the initial design, which are then searched in turn. The search terminates at the first satisfactory design or when useful change options have been exhausted.

6.2 PB Framework Architecture

Figure 6-1 shows the architecture of the rule-based framework PB. The central component is the Inference Engine (IE), which interprets the rules stored in the Rule Base (RB), and instructs the Model Solver (MS) on how to solve the model. The IE analyses the performance results and explores the change space for model improvements. The Diagnosis Assistant (DA) extends the ability of the IE by taking care of the complex work of data calculation and model manipulation.

Jess, introduced in section 3.4, is used in the IE. The main advantage of using a rule-based platform is its extensibility. New strategies for performance mitigation can be easily plugged in as new rules in Jess with less programming, even at run-time. The

advantage of using Jess rather than some other off-the-shelf rule engine is its simple integration with the existing LQN editor tool (JLQNEditor, also written in Java).

Figure 6-1 PB - Rule-based knowledge system framework

The workflow of PB is controlled by organizing the rules into different modules and by firing different modules at different stages. The current RB is defined in 5 main modules: diagnosis, configuration mitigation, design change mitigation, model transformation, and logging. The flow alternates between configuration and design change processes, as shown in Figure 6-2. This will be described in more detail in the next section.

The rules coded in the Jess language are designed to be general in order to be applicable for as many LQN models as possible. However, performance diagnosis and design changes are unavoidably project specific. To accommodate this, a configuration file is used to define parameters and constraints for diagnosis control. This configuration file customizes the general rules to the specific software model and project. Details about the configurable control parameters are discussed in section 6.6.

6.3 Evaluation of a Design by Its Best Configuration Found by PB

The performance of the system is iteratively diagnosed and improved. The iteration may be terminated by any of the following 4 conditions:

- the performance requirements are met;
- no more changes can be made under the current constraints and rules;
- the maximum number of iterations has been reached;
- or at the user's choice, based on observing the search progressing.

Figure 6-2 Alternating configure and design change process

Figure 6-2 describes the cycle of design change and configuration change. Each design is evaluated only after improving its configuration (a point discussed in section 5.4), within given constraints on the deployment and configuration options. Then a set of performance models representing potentially better designs is generated, giving a search tree in the space of design alternatives.

One cycle in Figure 6-2 is called a *Round* of the algorithm. It begins with one design state **D** and its evaluation, and ends with a set of new candidate design states and their evaluations.

In this research, the configuration changes are limited to modification of the resource multiplicities in a LQN model. They can be mapped to various run-time configuration parameters such as thread pool size, buffer pool size, and processor multiplicity. All other model modifications including change of model structure or workload (execution demand and call frequency) are taken to describe design changes, because the corresponding interpretation in the software design usually involves the redesign or reprogramming of part of the software.

A *Round* is a set of *steps*. A *step* of the algorithm begins when a new model is generated at any point. A new version of the performance model is created and solved, and the performance results are attached to it. The step gives a *StepRecord* which includes the model, the results, a unique id, a link back to the parent *StepRecord*, and a change log that describes the changes made from the parent.

In a given algorithm step, multiple change options may be suggested by the rules. Each option results in a child step. Therefore, we create a tree of steps as the diagnosis-mitigation progresses. Since only versions of the model with exactly the same structure and demand values are considered to be identical, the possibility of creating identical models by following different branches is low, and is ignored here.

Step ids start from 0, which denotes the base model. The numbering of the step id does not imply any relationship between steps (such as a parent-child relationship). It only shows the sequencing of step creation.

Figure 6-3 shows an example of a diagnosis tree. As we discussed earlier in this section, a configuration process is initially performed on the base model in order to find a best configuration before trying any design change. The configuration search is directed by

the bottleneck rules. It results in a series of steps increasing the length of the first branch of the diagnosis tree, as shown by steps 0 to 2. The procedure for configuration search will be further discussed in the next section.

Figure 6-3 Tree structure of a diagnosis-mitigation progress

When a best configuration is reached, such as at step 2 in the Figure 6-3, the control is handed over to the design change process. There are usually multiple design changes that can be evaluated and ranked by effectiveness or other factors such as cost. In this case, 3 change options are created and result in 3 new versions of the model (represented by nodes Step3, Step6 and Step7). In each case, the control is handed back to the configuration process. Each of the new model versions are processed and form 3 new branches in the tree. The whole procedure uses a depth-first approach.

The starting point of each configuration process is defined as a *configuration base*, represented by the grey shaded nodes in Figure 6-3. The step with the best configuration

for each branch is at the end of the branch and is called a *design change base*, meaning it is a candidate for design change. The design change bases are shown as double bordered nodes in Figure 6-3. As we can see in the diagram, a design change process generates new steps from a design change base. These new steps, in turn, become new configuration bases, such as steps 3, 6 and 7 in the diagram.

The first round in Figure 6-3 begins at Step 0 and ends at Steps 5, 6, and 8. The second round shown in the figure begins at Step 8 and ends at Steps 10, 13 and 15. There may be descendants of Step 5 and Step 6 but they are not shown.

When multiple design change branches are obtained at the end of a round, the performance improvement of each branch is listed and ranked. The process can continue along one of the branches according to user's choice or pre-configured selection criteria.

6.4 Search for the Best Configuration

The basic approach for seeking the best configuration for a model is to increase the multiplicity of bottleneck resources step by step using the bottleneck mitigation rule until the multiplicity limit is reached or the bottleneck has been removed. Figure 6-4 shows the decision chart for this process.

The abstract rules for problem identification described in Chapter 4 are coded into concrete rules using the Jess rule language. For easy understanding, however, the concrete rules introduced in this chapter will still be presented in pseudo code. LQN surrogates for design properties will be used in the representations. The complete Jess rule-base is attached in Appendix A of the thesis.

Figure 6-4 Process of searching for best configuration

6.4.1 Bottleneck Identification Rules

Root bottleneck identification has been discussed in detail in section 5.3.1. The bottleneck resource can be hardware or software, and in the abstract rules, they are treated the same. In an LQN model, software resources are represented by tasks, hardware resources are represented by processors, and they have different features and have to be differentiated. To implement the abstract rules, we need to define the following concepts:

Saturation Level: Normalized utilization of a task T or a processor P is named the saturation level of T or P , and noted by SL_T or SL_P . The range of SL_T or SL_P is $[0, 1]$, with 1 meaning 100% utilized.

Saturation Threshold: The practical threshold of saturation level value, beyond which the corresponding resource is considered saturated, noted by SL_{th} .

The current version of PB supports different saturation thresholds for processors and tasks, but no further differentiation of threshold for individual resources, i.e. all tasks in a model share the same software saturation threshold, and all processors in a model share the same hardware saturation threshold. There does not appear to be any justification for defining more detailed thresholds for each resource. The task threshold SL_{thT} and the processor threshold SL_{thP} are configurable parameters for the diagnosis process. Choosing different values for the thresholds may result in different mitigation results, especially when the model has evenly distributed execution demands.

Rule 6-1 Hardware Bottleneck Identification Rule

```
IF processor  $P$  has saturation level  $SL_P$  &  $SL_P > SL_{thP}$ 
THEN assert hardware bottleneck  $P$ 
END
```

Rule 6-2 Software Bottleneck Identification Rule

```
IF task  $T$  has saturation level  $SL_T$  &  $SL_T > SL_{thT}$  ,
AND NO server of  $T$  is saturated,
AND NO processor is saturated,
THEN assert software bottleneck  $T$ 
END
```

When multiple root bottlenecks exist in the model, the rules can identify all of them at the same time. All of the root bottlenecks, either in hardware resources or software resources are then handled together in a single step in the mitigation.

6.4.2 Configuration Mitigation

In this implementation, the only mitigation that is considered as a configuration change is adding resources, which is coded as increasing the multiplicity of a task or a processor.

Rule 6-3 Configuration Mitigation Rule

```

IF resource  $R$  is a root bottleneck,
  IF  $M_R < M_{R \max}$ 
    THEN set  $M_R = \min \left( M_{R \max}, \left\lceil \frac{SL_R}{0.9 \times SL_{thR}} \right\rceil \right);$ 
  ELSE assert irresolvable bottleneck  $R$ 
END

```

In this rule, M_R stands for the multiplicity of resource R . R can be either a task or a processor. $M_{R \max}$ stands for the maximum possible multiplicity of resource R .

The nature of this rule is to increase the multiplicity of resource R in order to reduce its normalized utilization to below the threshold (e.g. reduce it to 90% of the threshold), unless limited by R 's capacity $M_{R \max}$.

$M_{R \max}$ is also a configurable parameter for model diagnosis. It can be set on each individual task or processor. With no explicit setting, the default value would be infinite, i.e. no constraints on capacity.

When a resource is limited by availability constraints (such as thread pool size or requirement of mutual exclusive control), the bottleneck cannot be solved by configuration mitigation, and will be marked as irresolvable or be transformed into long path problem in order to trigger design changes.

6.5 Generate Design Change Options

Design changes are triggered by long path problems or irresolvable bottlenecks. The rules for design change define *possible* software design transformations in a general way. The user may also set project specific constraints as defined in section 6.6. These constraints function together with the general rules, and customize the general transformation to suitable design changes on a given software model.

It might be difficult for designers to provide all constraint values. There are default values based on experience to help designers use the framework when there is less information available. Also, with fewer constraints, the system will simply try out more options. Users can always discard the options that are not applicable.

6.5.1 Long Path Identification Rules

Two conditions identify long path problems. One is an entry with a service time which is longer than allowed by the requirements (i.e. expected value for service time). Another is a bottleneck that cannot be resolved by adding resources (because of a resource limitation), so one has to reduce the resource holding time. Long path problems are associated with service given by entries in the LQN model. Therefore we have the following long path rules (in which E is an entry, T_E is its task, P_{T_E} is its processor, s_E is its service time, and s_{Ereq} is required minimum value for s_E):

Rule 6-4 Pure Long Path Rule

```

IF an entry  $E$  has  $s_E$  &  $s_E > s_{Ereq}$  ,
AND no server task of  $T_E$  is saturated,
AND processor  $P_{T_E}$  is NOT saturated
THEN assert long path at  $E$ 
END

```

Rule 6-5 Soft Bottleneck Long Path Rule

```

IF a bottleneck task  $T$  is irresolvable
THEN assert long path at each entry of  $T$ 
END

```

This rule says that, if a software bottleneck cannot be mitigated by configuration, look into path length reduction for each entry in it.

Rule 6-6 Hard to Soft Bottleneck Rule

```

IF a bottleneck processor  $P$  is irresolvable
THEN assert each task on  $P$  as an irresolvable bottleneck
END

```

This rule attempts to reduce workload on saturated hardware resources.

Under these rules, multiple long path problems may be identified at the same time. Long paths that are associated with the same bottleneck will be handled together.

6.5.2 Design Change Rule: *ShrinkExec*

One way to improve performance is by “tightening” the code and reducing the number of machine instructions that are executed, for a given operation. There are many detailed ways to do this, some of them indicated by some of performance principles of Smith [Smith86] (e.g., test for a fast special case of a part of the operation, a “fast path”, or do early binding of objects to their targets). PB does not consider this level of detail, but allows the user of PB to set a target shrinkage factor Δ_{ct} which may be achievable.

Code tightening is much more rewarding for some operations than for others, in terms of performance improvement it provides to the system. What PB does is identify operations which appear to provide the most worthwhile opportunities. The actual work of redesigning the code for those operations is left to the ingenuity of the designer, and may be more or less effective than Δ_{ct} . The factor Δ_{ct} is applied to both the entry service time, and the frequency of calls going out of the entry in question. PB allows different factor values for entry service time and out-going call frequency, noted as $\Delta_{ct}S$ and $\Delta_{ct}Y$ respectively in the following rule:

Rule 6-7 ShrinkExec Rule

```

IF entry  $E$  has a long path
AND  $E$  has  $\Delta_{ct}S > 0$  or  $\Delta_{ct}Y > 0$  ,
THEN reduce execution demand of  $E$  by the factor  $\Delta_{ct}S$  ,
AND reduce the frequency of  $E$  's outgoing calls by the
factor  $\Delta_{ct}Y$ 
AND set  $\Delta_{ct}S = 0$  &  $\Delta_{ct}Y = 0$  for  $E$ 
END

```

This rule reduces the execution demand of the problem entry itself in order to shorten the scenario path. The potential for further shrinkage of a same entry is prevented by setting shrinkage factors $\Delta_{ct}S$ and $\Delta_{ct}Y$ to 0 after the transformation being applied on the entry.

6.5.3 Design Change Rule: *MoreAsync*

When an entry execution blocks a caller, the duration of blocking can be reduced by making part of the execution asynchronous, which is by executing that part after replying to the caller. This can only be done when some operations by the entry do not affect the

reply, so they can be done after it; examples include delayed writes and buffer cleanup operations. The asynchronous part is called in LQN a second phase of service. Asynchronous execution does not reduce the holding time, but does reduce the blocking of the calling resource, and thus reduces its holding time.

Just as for code tightening, the details of reorganizing the flow of execution of the entry are not determined by PB, but good opportunities for introducing second phases are identified. The user states a fraction Δ_{ma} which it is believed can be moved to second phase, and PB attempts to apply this fraction and find where it could be most useful in improving performance. Same as the *ShrinkExec* rule, Δ_{ma} can also have different values for the entry service time, and the frequency of calls going out of the entry in question, noted $\Delta_{ma}S$ and $\Delta_{ma}Y$ respectively below.

Rule 6-8 MoreAsync Rule

```

IF entry  $E$  has a long path,
AND  $E$  has a target entry  $E_T$ ,
AND  $E_T$  has an asynchronous ratio  $\Delta_{ma}S > 0$  or  $\Delta_{ma}Y > 0$ ,
THEN move fraction  $\Delta_{ma}S$  of the 1st phase CPU demand of  $E_T$ 
to its second phase,
AND move fraction  $\Delta_{ma}Y$  of its 1st phase calls to its 2nd
phase,
AND set  $\Delta_{ma}S = 0$  &  $\Delta_{ma}Y = 0$  for  $E_T$ 
END

```

This rule reduces the blocking time of the problem entry for its underlying services, therefore reducing the scenario path length. Like in the *shrinkExec* rule, the control factors $\Delta_{ma}S$ and $\Delta_{ma}Y$ are set to 0 to prevent overuse of asynchronous mitigation in the same entry.

6.5.4 Design Change Rules for Bottlenecks

Bottlenecks that are not solved by configuration mitigation have been marked as *irresolvable* by the configuration mitigation rules described in section 6.4.2. The following three rules can further mitigate them in the design change process by moving some of the operations away from the bottleneck resources.

The first lightens the load on a bottleneck processor by balancing the load

Rule 6-9 Redeployment Rule

```

IF a bottleneck at processor  $P$  is irresolvable
AND multiple tasks are deployed on  $P$ 
AND processor  $P'$  is the most lightly loaded processor
AND  $P' \neq P$ 
THEN move a task from  $P$  to  $P'$ 
END

```

The decision regarding which task to be moved can employ two selection criteria: *lightest first*, i.e. move the task with shortest service demand, or *heaviest first*, i.e. move the task with longest service demand. Heaviest first moving strategy may be more effective than lightest first strategy in mitigating the current processor bottleneck, but increases the chance of saturating the receiving processor. The overall impact depends on the load distribution among tasks and processors. When a moving opportunity appears, PB will try out both strategies and give estimations on performance and bottleneck shifting to assist design decisions.

The second rule for a bottleneck task partitions its load, by partitioning its operations among several tasks. It then can be load-balanced across processors in a later step. As for

code tightening and *MoreAsync*, the details of how this is done must be found by the designer, and PB simply finds *opportunities* for this strategy to be successful.

Rule 6-10 Partitioning Rule

```

IF a bottleneck task  $T$  is irresolvable
AND  $T$  has multiple entries
AND entry  $E_i$  has the longest execution demand in  $T$ 
THEN create a new Task  $T'$  on the same processor as  $T$ 
AND move  $E_i$  to  $T'$ 
END

```

The third strategy is an important and successful one for performance improvement. In many systems where a large number of detailed operations are carried out on related data, it is more efficient to group or batch these operations together into a single larger operation. This is particularly true where each small operation involves a network delay, and the batched operation requires only one network delay. PB addresses this particular case (other kinds of batching would require different logic):

Rule 6-11 Batching Rule

```

IF a bottleneck task  $T$  is irresolvable
AND  $T$  has network latency as a (direct or indirect)
server resource  $T_{net}$  ,
AND  $T_{net}$  has incoming call frequency  $?cf_{to-net}$  &  $?cf_{to-net} > 1$ 
AND  $T_{net}$  has outgoing call frequency  $?cf_{from-net}$ 
THEN set  $?cf_{to-net} = 1$  ,
AND set  $?cf_{from-net} = ?cf_{from-net} \times ?cf_{to-net}$ 
END

```

The network delay is assumed to provide a pure latency and is modeled by a pseudo task with a thinking time. In order to exploit the batching rule, the LQN model is extended to include a special flag on a task to indicate a network latency. The rule is also based on the assumption that the extra overhead of processing multiple messages or blocks at the network interface is essentially the same as the overhead of processing one message or block.

6.5.5 Ranking Design Change Options

Each of the design change rules in the previous section generates a design change option. Here the design change options obtained are transformations to be applied on a LQN model. It is possible that a transformation on the LQN model could be interpreted as different design model changes for different software system or by different designers.

All options generated from the same base need to be ranked in order to assist user selection. The following factors can be used for the ranking index:

- Performance: the total throughput or response time of the testing scenario.
- Cost: the total cost in terms of money or time for making the recommended design changes.
- Design Change Efficiency (DCE): the ratio of performance improvement versus design change, i.e. the percentage of performance improvement by 1% of design change

$$DCE = \frac{pi\%}{ch\%} \quad (1)$$

pi: percentage of performance improvement (% increase in throughput or % decrease in service time)

ch: percentage of design change (e.g. % in code tightening, % increased asynchronous operation, % partitioned into separate task, etc)

- Cost Efficiency (CE): the percentage change achievable for a cost unit. For instance, if the cost unit is a dollar, CE represents the percentage performance improvement achievable per dollar.

$$\text{Efficiency} = \frac{pi\%}{ch\% \times uc} \quad (2)$$

uc: unit cost, i.e. cost of making 1% of design change

6.6 Using Control Parameters to Customize Rules

A configuration file is used to set up parameters and constraints for the diagnosis control of the model. This configuration set up customizes the general rules to each specific software model in order to get meaningful results for that model. Its content includes:

Control parameters:

- Required minimum throughput (f_r) at a client task
- Requirement on the service time (s_r) at a client entry
- Saturation threshold for utilization of tasks and processors (SL_{thT} and SL_{thP})
- Maximum iterations (I_{max})

Constraints on changes

- Resource capacity (C_r): maximum multiplicity constraint of a resource. This constraint is defined for every hardware and software resource in a LQN model.
- Maximum shrinkage ratio (Δ_{ct}): maximum fractional reduction of the current execution demand that is deemed possible by code tightening. This constraint is defined for each entry in a LQN model.

- Maximum asynchronous ratio (Δ_{ma}): maximum percentage of the current synchronous demand that is deemed possible to be moved to asynchronous part, i.e. to be moved the first phase execution to the second phase of an entry in LQN. This constraint is defined for each entry and each call in a LQN model.
- Cost factors (CW_{change}): cost in terms of time or money to achieve recommended configuration and design change, such as fund for purchasing more hardware equipments, time for modify design and program, or payment for programmers to do so. In diagnosis on a LQN model, the factors are simplified as a cost weight value for a change option.

The required minimum throughput and response time deadlines represent performance requirements for the system. The saturation threshold for utilization is used for bottleneck identification. The constraints are used by mitigation rules as conditions for action selection. Cost factors affect the priority level of mitigation candidates. Resource availability limits the multiplicity level of a resource, which can further limit the concurrency level of the model.

The PB framework works under the following assumptions:

- The whole amount of code tightening factor or asynchronous factor is applied for an Entry in one step. Then the factor becomes 0 so that the reduction will not be applied again.
- A model can have more than one class of workload. Each class of workload is generated from a separate reference task that has only one entry.
- Performance is evaluated by throughput of client tasks (reference tasks) and response times of client entries.

6.7 Mapping Feedback to the UML Design

This section describes how the decisions made based on LQN model analysis can be interpreted as changes in the UML design, which is intuitive to designers and developers.

6.7.1 Resource Multiplicity Change

Configuration mitigation in PB results in multiplicity increasing for software and hardware resources. Figure 6-5 shows a common way of representing resource multiplicity in a LQN model and in an annotated UML model. The LQN model contains a task *Server* with multiplicity of m running on a processor *AppProc* with multiplicity of n . The UML model shows a deployment diagram. The software resource *Server* is annotated with a `<<PAresource>>` stereotype, which is deployed on the hardware resource *AppProc* annotated with stereotype `<<PAhost>>`. Resource multiplicities in the UML model are annotated by using the `PAcapacity` tag of the SPT profile. A `PAcapacity` in UML represents a resource run time configuration, not the planned resource capacity C_r in PB control parameters as in section 6.6. A change for increasing multiplicities would be represented as a change in the values of m and n in both the models.

Figure 6-5 Resource multiplicity in LQN and UML

Multiplicity for a software resource can be implemented by multithreading. When multithreading is introduced into a process, care must be taken that the concurrent threads do not make conflicting accesses to shared data. It may be a significant design change to go from a single threaded design, to multithreading at any level.

The multithreaded component itself may not have a design change if its original threading level is greater than 1, but it may introduce design changes for other related components. Mutually exclusive resources cannot be multithreaded. When a multithreaded client requests a resource that requires exclusive access, contention delay will occur. Extra logic on access protection is also needed which results in overhead as discussed in Smith's Shared-Resource Principle. The configuration change approach in PB does not consider the extra overhead introduced by multithreading. In order to take it into account, the LQN model needs to be modified according to the change. Parameters about overhead have to be acquired from the designer.

Replication

Using replicas is another way of adding more resources into the system, which is currently not included in PB. There is no formal notation defined for replicas in UML and its profiles. The information about replicas can be represented by plain text notes attached to corresponding resources in the UML model. This thesis proposes the notation as shown in Figure 6-6 to represent replication in a UML sequence diagram.

The impact of replicas on software is similar to multithreading. The difference is that instead of serving requests in the same incoming queue as in multithreaded server, each replica has its own incoming queue. Requests to the server are evenly distributed into replica queues by a load distributor. The performance of replication is not as good as multithreading because a replica could be idle while there are still waiting requests in another replica's queue. Also extra logic on load dispatching and balancing is needed.

The advantage is that different replicas can be allocated on different computing nodes, and thereby improve system reliability.

Figure 6-6 Resource replication in LQN and UML

6.7.2 ShrinkExec Change

Figure 6-7 shows how the *ShrinkExec* change is implemented in a LQN model transformation and its interpretation in a UML model change. The LQN model assumes a single entry on all three tasks for simplicity. The *ShrinkExec* transformation in the LQN model is done on the entry of *Broker*. Different shrinkage factors $\Delta_{ct}S$ and $\Delta_{ct}Y$ are applied on the service demand $\bar{d}$ and outgoing call frequency $\bar{f}$ of the entry. The UML model is changed accordingly by reducing the assumed mean value of $PA\bar{d}$ from $\bar{d}$ to $\bar{d} * (1 - \Delta_{ct}S)$, and reducing the repetition time for service invocation between *Broker* and *Server* from $\bar{f}$ to $\bar{f} * (1 - \Delta_{ct}Y)$.

Figure 6-7 *ShrinkExec* in LQN and UML

6.7.3 MoreAsync Change

MoreAsync change in PB introduces second phase workload which is a LQN specific concept. The BSS example in Chapter 4 has shown a way of representing second phase workload in UML. Figure 6-8 shows a general case for the mapping of *MoreAsync* transformation between a LQN model and a UML model.

Figure 6-8 *MoreAsync* in LQN and UML

In LQN, only the first phase workload of a task blocks its client. Moving the workload, including the execution demand and any outgoing requests to lower level services, from the first phase of task to its second phase can release the client task earlier, thus increasing the concurrency level and parallelism, and shortening a path that includes only the first phase. In UML, the second phase workload can be represented by a self invocation that being executed after the reply message being sent to the client.

Using a second phase workload may require increasing the multithreading level of the changed task, e.g. the Broker task, in order to show its benefit. Otherwise, when the next request comes, the Broker may still be busy with its long second phase work. A server with a long second phase may become a bottleneck if it has a low multithreading level.

6.7.4 Batching Change

A batching opportunity is identified in a LQN model by recognizing a special network task with pure delay and multiple outgoing calls, as shown in Figure 6-9 (a). The batching change in the LQN model is to move the call frequency (f) from between the sender task (Client) and Net to between Net and the Server task as shown in Figure 6-9 (c). The change is based on the assumption that the network delay of transmitting the data for a single service request has no significant difference from that of transmitting a set of data for all requests. By batching the network operations, extra network overhead is saved which shortens the path that contains it.

To represent the batching change in a UML model, the network resource has to be presented explicitly as an active object instead of as an external operation as used in the BSS example. Figure 6-9 (c) and (d) show the change in the UML model.

Batching on operations other than network transmission is also possible, such as batching database operations, file system operations. To enable batching opportunity being identified for these operations, the overhead related to these operations have to be modeled explicitly in the UML model and the derived LQN model.

Figure 6-9 Batching in LQN and UML

6.7.5 Partitioning Change

Figure 6-10 shows how partitioning transform is done in a LQN model and its interpretation in a UML model. Task *Server*, which has multiple entries, is identified as a root bottleneck. By applying the *Partitioning* rule (rule 6-10) in the LQN model, entry *service2* is separated from *Server* into a new task *Server_p1*. In the changed UML model as in Figure 6-10 (d), a new resource *Server_p1* is created as well which responds to the invocation for *service2*.

Figure 6-10 Partitioning in LQN and UML

Partitioning is applied by PB when a root bottleneck reaches its capacity limit. The nature of it is still introducing more resources into the system. After partitioning, different services of the same software resource do not withhold each other. In contrast with *MoreAsync* change which can increase parallelism, partitioning increases the concurrency level only. The difference between parallelism and concurrency is: parallelism is usually considered when inside a single response, while concurrency is between independent responses. Partitioning enables services for different clients to interleave with each other. When there are orders enforced on the execution of partitioned resources, the partitioning transformation results in a pipeline pattern.

6.7.6 Redeployment Change

Mapping redeployment change between a LQN model and a UML model is straightforward. Figure 6-11 shows an example of moving a software resource Server1 from processor AppProc1 to processor AppProc2.

Figure 6-11 Redeployment in LQN and UML

CHAPTER 7 IMPLEMENTATION OF THE PERFORMANCE BOOSTER (PB)

This chapter introduces the implementation of PB in Java and the Jess language. In the implementation of PB, the Jess rule engine is combined with functionalities for performance diagnosis, experiment control and LQN model manipulation, such as model parsing, model solving, model structure recognition and model transformation.

The result of the implementation is a performance expert system with a set of rules that can automatically diagnose a LQN performance model, point out specific performance problem spots, propose ranked mitigation options and give estimations on improved performance for each option. Given preset ranking mechanisms, the system can also automatically perform continuous performance diagnosis and improvement until achieving the performance requirements, and give a list of steps taken to achieve such performance level for designers to follow.

7.1 Implementation Overview

Implementation of PB contains mainly of two parts: functionalities for LQN model manipulation, experiment control (LQN solver invocation), and file manipulations are implemented in Java; rules for diagnosis and model transformation strategies are written in the Jess language.

PB is integrated with three existing tool sets:

- JLQNEditor: provides a graphical user interface for LQN model construction and file I/O operations. It also includes the LQN Core Model which represents LQN

model entities as Java objects. PB operates on these objects for model diagnosis and transformation.

- Jess rule engine: for rule interpretation and inference.
- LQN solvers: model solving and model file format transformation.

Figure 7-1 Class diagram of PB implementation

Figure 7-1 shows important classes and their attributes or operations implemented in Java for PB, and their relationships with the three existing tool sets, shown as grayed packages in the figure. Some utility classes, such as an XML reader for the diagnosis trace, and a property reader for the LQN configuration, are also implemented but not shown here.

Class `DiagnoController` provides the central control of model diagnosis and improvement. A diagnosis procedure starts with the `JLQNEditor` reading in a LQN model file (`model.lqn`) and invoking the diagnosis service on `DiagnoController`. `DiagnoController` reads in the control parameters (presented in section 6.6) from the model configuration file (`model.config`) and stores them as a `ModelConfig` object. The model is then sent to `ModelSolver` for solving. The results are read in and returned to `DiagnoController` which then asserts facts about the results into Jess working memory accordingly. The Jess rule engine applies rules on the asserted facts and proposes actions for model transformation either as configuration changes or as design changes. Model transformations are performed by the `DiagnoAssistant` class on the java objects representing model entities. Changed models are then solved and transformed again. This procedure is repeated until a termination condition (discussed in section 6.3) is met.

Rules in Jess language are stored in a text file (`rulebase.clp`), which is loaded into the Jess engine at the beginning of a diagnosis procedure. As introduced in section 3.4.4, the Jess rule base can be partitioned into modules for process control purpose. The following 5 modules are defined for PB rule base:

- `diagnosis`: rules for locating root bottlenecks, including software bottleneck and hardware bottleneck.
- `configmitigation`: rules for addressing software bottlenecks and hardware bottlenecks by configuration change.
- `structuremitigation`: rules for asserting design change requests for hardware bottlenecks and software bottlenecks that are not resolved by configuration changes. Long paths are treated as software bottlenecks at clients (top level tasks).

- `transmodel`: rules for applying design changes on the model.
- `changelog`: logging for changes done on the model and presents feedbacks to user.

There is also a default main module, which contains data structure definition and simple utility functions. Complex utility functions involving model manipulations are implemented by `DiagnosisAssistant` and are invoked through the Jess plug-in class `DiagnoFunctionImpl`.

The complete content of `rulebase.clp` is attached in Appendix A.

7.2 Virtual Profiling - Fast Path in Configuration Search

Searching within the feasible configuration set $\mathcal{C}$ for the configuration that provides the best performance of the design can be time consuming and needs proper planning. The simplest plan is a basic incremental search which starts with the lowest resource configuration, such as a single thread for all software resources and a single processor for all hardware resources, and increases each resource multiplicity step by step by repeatedly using the Root Bottleneck Rule (Rule 5-5) and Bottleneck Multiplicity Rule (Rule 5-7). This procedure may be time consuming, especially for cases where initial design has very low multiplicities. We may see half or more of the diagnosis effort spent on improving configuration for initial model. There are some fast paths that can be considered for improving the performance of the diagnosis process itself.

PB implemented another approach named Virtual Profiling (VP) in this thesis. VP first solves the LQN model with infinite multiplicities on all of the software and hardware resources except those that have design constraints, e.g. a single thread of control component or a mutual exclusive point.

Performance results of the model with all resources set to maximum possible resource units gives insight into how much actual resource are needed in order to achieve a similar performance. The LQN results show the mean number of busy servers for each resource. These numbers represent the average resource requirements for the model under a given workload. Configuring the model according to these numbers is close to the best configuration possible. For example, if a resource has a mean utilization of U_{vp} , we can set its multiplicity to $U_{vp} / (0.9 \times U_{threshold})$ in order to obtain a configuration that has a utilization lower than the saturation threshold.

Litoiu et al. showed this approach in [Litoiu98]. In this way, the requirements for resource capacities and the needs of architecture change may be obtained with fewer experiments. For instance, in the BSS example, the need of changing the interaction pattern between `VideoController` and `AcquireProc` could be discovered immediately, rather than waiting until the multiplicity configurations on the `Buffer`, `StoreProc` and `ApplicationCPU` has been adjusted step by step.

However, when resources have finite multiplicities, there will be waiting time incurred in the model. Therefore, the resource utilizations will increase from the numbers obtained in VP and cause saturation. In this case, the basic incremental configuration search is still needed in order to obtain the realistic best configuration.

7.3 How to Extend the PB Rule Base

One of the benefits of using a rule-based framework is its extensibility. The current rule base includes 5 modules that have been introduced in section 7.1. The rule base has a full set of diagnosis and configuration mitigation rules, and a partial set of design change

rules. With only a few of the latter and the corresponding model transformation functions, the PB framework already can run a full round of diagnosis, and make substantial improvements.

The most fruitful points for extensions in PB appear to be at the long path assertion from bottleneck mitigations, and the design change rule module. There are a limited number of design changes rules in the current rule base. New rules can be introduced once a set of surrogates are fully identified for a design pattern.

To add a new design change option to the rule base, first, an entry for generating a change request of the new type is needed in the corresponding mitigation Jess rule in the `structuremitigation` module, depending on the type of issue (i.e. hardware or software) it addresses. Then a set of functions must be implemented in Java classes `DiagnoAssistant` and `DiagnoFunctionImpl` for model manipulation according to the transformation needed. A set of configuration parameters must be defined and set up for control of the triggering of the transformation rule. Finally, a transformation is needed in the `transformation` module that connects the request with transformation functions in Java.

CHAPTER 8 CASE STUDIES AND APPROACH VALIDATION

This chapter shows study cases that demonstrate the effectiveness of the PB framework and the rules. With even a small set of rules, the PB framework has shown potential for proposing valuable suggestions to software designers. LQN model scripts for all models studied here have been attached in Appendix B of the thesis.

8.1 A Tutorial Example

The prototype PB tool was applied to a simplified web application system to examine the search strategy in action. The search was run until no more improvements could be obtained (thus the goal response time was set to zero). Extended cases are also described to show the effectiveness of more complex rules (Partitioning and Batching).

Figure 8-1 UML sequence diagram for a web application

The base version is specified by a Sequence Diagram annotated with the SPT profile, shown in Figure 8-1. The User invokes a service published on a web server WS, which, in turn, calls services provided by two application servers (App1 and App2). Each application service requires some database operation. In the base design, all software resources are single threaded ($PA_{capacity} = 1$ is omitted from the diagram).

The annotated specification is transformed [Woodside05] into a layered queueing network (LQN) performance model, shown in Figure 8-2.

Figure 8-2 Derived LQN model for the web application system

8.1.1 Base Design of the Web Application (Cases 1 – 4)

As mentioned before, experiments are first performed on the base version of the web application design to demonstrate the most straightforward improvement techniques. Three kinds of improvement were considered, multiplicity for configuration improvement,

and *shrinkExec* and *moreAsync*, for design improvement. For all tasks, the maximum shrinkage ratio (Δ_{ct}) were set to 0.20 for both service times and call frequencies. The maximum asynchronous ratio (Δ_{ma}) were set to 0.50 for both service times and call frequencies. The thread multiplicity constraints are listed below:

Table 8-1 Thread multiplicity constraints

Task	Initial multiplicity	Maximum
WS	1	10
APP1	1	5
APP2	1	3
DB1	1	10
DB2	1	10

The cost weights of changes were set at 1 for *moreAsync* (CW_{ma}) and 2 for a *shrinkExec* change (CW_{ct}), on the grounds that the former only moves blocks of code around and might in general require less effort than revisiting the code in detail.

Two ranking criteria were applied for choosing the best design change: best response time and lowest cost. For each criterion, the search was run until no further improvement in response time could be obtained, giving five rounds of trials in each case. Ranking by the best response time gave the steps shown in Figure 8-3. The figure only shows the design base cases; the steps for configuration improvement are omitted. The initial evaluation (Round 0) took 15 steps, and identified WS as the problem entity. The two possible design changes are represented by arcs labeled *A* and *C*, where *C* applies *shrinkExec* to task WS, while *A* introduces a second phase (*moreAsync*) in the same task.

Figure 8-3 PB diagnosis tree (showing only the rounds)

(Label A identifies improvements by *moreAsync*, C identifies improvements by *shrinkExec*)

After configuration improvement the evaluations of *A* and *C*, found at Steps 16 and 22 show that *A* is better, so that path is taken for Round 1. The bold circles and arcs represent the best choices selected by PB, showing 5 rounds of improvements and we will call this the diagnosis trace.

Table 8-2 shows the values and costs over the 5 rounds. A total of 58.9% improvement is achieved for response time.

Table 8-2 Performance analysis and improvement results**(using best response time as the ranking index)**

Round	Step	Response time	Cost weight
0	15	72.68	0
1	22	47.13	1
2	27	36.34	2
3	29	34.66	4
4	32	30.00	6
5	36	29.88	7

Following the diagnosis trace, at the end of Round 0, PB identifies a bottleneck at task WS. Using the *A* option, in Round 1 the bottleneck is pushed to App1. The *A* option is chosen again, and the bottleneck stays at App1. Since its maximum asynchronous change has already been used, the second choice, *shrinkExec*, is applied, which pushes the bottleneck to App2 at Step 29. The *C* option is chosen next, after which the bottleneck is still at App2, so the *A* option is applied to it to give the final model at Step 36.

The choice of the ranking index affects the sequence of decisions, although if the process is followed to the end (as here) it may give the same final result. Figure 8-4 compares the sequence of response times and cost weights found when ranking choosing based on response time (case 1), or on weight (case 2). Response times are labeled on the left, weights on the right. The first three choices happen to be the same for both criteria. At Round 4 cost dictates a different choice, with lower cost but not as good response time.

Figure 8-4 Performance result vs. cost weight for case 1 and 2

Figure 8-5 Performance vs. cost weight for case 3 and 4

Constraints on the final choice bring the changes back together in this case, but if the search terminated at the satisfaction of a performance criterion, the two strategies might, in general, have different solutions.

A small variation on the above case shows how the search may diverge earlier in the process and may thereafter follow a different path. In case 3 and case 4, the maximum limit for *moreAsync* was reduced to 0.20. This changed the comparative evaluations and led to different choices. Figure 8-5 shows the sequence of response times and weights over the rounds of the search. If we use cost weight to rank the proposed changes (case 4), the performance improvement finally achievable is less than if we use performance as a ranking index (case 3).

To interpret the improved model as a changed software design represented in UML and SPT,

- the *shrinkExec* change is interpreted as a decreased execution demand, and
- the *moreAsync* change is interpreted by replacing a synchronous invocation by an asynchronous invocation with an (early) asynchronous reply, and part of the service placed in an operation executed after the reply.

Section 6.7.3 has shown an example of the interpretation of the *moreAsync* transformation in the software interaction diagram.

8.1.2 First Extended Design (*Partitioning* Mitigation – Case 5)

Suppose the functionality requirements for the Web Application expand to a design with multiple services (entries) offered by *App2*. The LQN model that reflects the first extended design is shown in Figure 8-6, with entries *App2_E*, *App2_E2*, and *App2_E3*, all with the same average demand 1.25ms.

Figure 8-6 Derived LQN model for extended design of the web application system with multiple entries on App2 (case 5)

Table 8-3 Performance results for case with partitioning (case 5)

Round	Step	Response	Design Change
0	16	95.41	base case
1	17	75.37	<i>shrinkExec</i> on WS
2	20	64.07	<i>shrinkExec</i> on App2
3	24	60.70	<i>moreAsync</i> on WS
4	31	39.15	<i>partitioning</i> on task App2
5	32	34.91	<i>shrinkExec</i> on App1
6	38	33.96	<i>partitioning</i> on task App2
7	43	28.55	<i>moreAsync</i> on App1

The same multiplicity limits and change limits as in case 1 were used in diagnosis. Table 8-3 and Figure 8-7 show the performance results of the system with multiple entries on App2 being improved by using response time as the ranking index. The results show that the Partitioning option is selected in Rounds 4 and 6, and gives favorable improvements especially in Round 4 (35.5.% improvement in response time comparing with its last round).

Figure 8-7 Performance improvement for case with partitioning (case 5)

Figure 8-8 shows the improved LQN model. Entries App2_E2 and App2_E3 which originally belonged to Task App2 are now separated in 2 new tasks (App2_p1 and App2_p2). By breaking a big component into smaller pieces, more concurrency is introduced into the system. Before the changes of partitioning, when a client requires App2_E service on an App2 thread, the service App2_E2 and App2_E3 on that copy are also blocked. After partitioning, App2_E, App2_E2 and App2_E3, each has its own task resource. The execution of the three entries can be interleaved for different client copies now. If there were a strict order of execution for the three entries, they now form a *Pipeline* software pattern. On the other hand, by partitioning one task into three, this actually introduces more threads into the system which also shortens waiting time.

Another benefit of partitioning a big task is that it brings potential for redeployment. In this case study, the capacity of processor P2 was not limited. In the case of P2 becomes

hardware bottleneck because of limited capacity, redeploying some of the smaller tasks onto other more lightly loaded processor would further improve the system performance. For example, task App2 with only entry App2_E could be moved to processor P4 which is under utilized. This change, in the real system, would also save communication time between App2 and DB2.

Figure 8-8 Improved LQN model for case 5

8.1.3 Second Extended Design (*Batching* Mitigation – Case 6)

Assuming that the network latency between App1 and DB1 is captured in the design model, we obtain the LQN model in Figure 8-9 that represents the network latency with the pseudo task Net.

Figure 8-9 Derived LQN model for extended design of the web application system with task Net (case 6)

The same change limits as case 1 were used, and the threading limit of WS is reduced to 7. Table 8-4 and Figure 8-10 show the performance results of the system with network latency. Batching was picked at the first round for its high efficiency, i.e. 56.2% improvement in response time. The result of batching is that one message is sent with all the data (instead of 10 messages), and DB1_E carries out 10 operations on the 10 operands. Figure 8-10 shows the part of LQN model that are being changed.

Table 8-4 Performance results for case with batching (case 6)

Round	Step	Response	Design Change
0	9	187.5	base case
1	16	82.13	<i>batching</i>
2	22	56.76	<i>moreAsync</i> on WS
3	25	43.63	<i>shrinkExec</i> on App1
4	30	36.04	<i>shrinkExec</i> on WS

Figure 8-10 Performance improvement for case with batching (case 6)

Figure 8-11 Batching change in the LQN model

8.2 BSS Example Revisited

The Building Security System was revisited to verify the diagnosis results of PB against the results obtained by a human expert.

The control parameters for PB diagnosis were configured as following:

- Requirement: 500ms mean value of service time at client entry `acquireLoop` (i.e. $s_r = 500$).
 - Note: the deadline requirement is adjusted from 1 second with 95% confidence to 500ms mean value due to XML parser limitation on collecting confidence intervals of performance results. If the delay is exponentially distributed, a mean of 500 ms gives a 95% level close to 1 second.
- Saturation thresholds: 0.7 for all tasks and 0.9 for all processors (i.e. $SL_{thT} = 0.7$ and $SL_{thP} = 0.9$) respectively
- Maximum iterations: $I_{max} = 50$ steps
- Resource capacities: No limitation (i.e. $C_r = \textit{infinite}$ for all resources).
- Maximum shrinkage ratio: $\Delta_{ct} = 0.2$ for all entries.
- Maximum asynchronous ratio: for the entry of task `AcquireProc`, $\Delta_{ma} S_{acqProc} = 0$, $\Delta_{ma} Y_{acqProc} = 1.0$; for all entries on other tasks, $\Delta_{ma} S = \Delta_{ma} Y = 0.5$.
- Cost factors: flat rate, i.e. $CW_{change} = 1$ for all design change options.

The diagnosis took 2 rounds in a total of 19 steps to finish:

- Round 0 (steps 0-4): configuration change on base case. Bottleneck was pushed to top level task `VideoController`. Due to the business requirement for a single thread of control as stated in Chapter 4, the bottleneck cannot be mitigated by configuration change. Design change opportunities have to be searched for.

- Round 1 (steps 5-19): opportunities of *MoreAsync* and *ShrinkExec* on task *AcquireProc* are identified. The change for *MoreAsync* is picked for its better performance over *ShrinkExec*.

The model was solved by the simulation solver LQSIM. To work around a bug in the LQN XML parser, the deterministic demands were made exponential and the processor *ApplicCPU* was set to infinite multiplicity. Therefore no hardware bottleneck was captured in the diagnosis. The LQN model used for PB diagnosis is attached as a PB version in Appendix B of the thesis. The diagnosis tree and performance results of important steps are shown in figure 8-12.

Figure 8-12 PB diagnosis tree of BSS model

The design change suggested by PB diagnosis is the same as that by the human expert as presented in Chapter 4. At step 4, the system is saturated at the client task *VideoController* with no underlying resources being saturated. In the situation of the client task being the only bottleneck in the system, the system has a pure long path

problem. Two mitigation opportunities are identified for changing the entry of high level task `AcquireProc`:

- *ShrinkExec*: shrink its execution demand and frequency of outgoing calls by 20% ($\Delta_{ct} = 0.2$), or
- *MoreAsync*: move the call between `AcquireProc` and `BufferManager` to second phase ($\Delta_{ma} S_{acq Proc} = 0$, $\Delta_{ma} Y_{acq Proc} = 1.0$).

ShrinkExec is tried at step 5 and no further configuration mitigation was identified after this change. *MoreAsync* is tried at step 6. After the change, the bottleneck is pushed away from the client task, and the configuration of the model is improved by step 7 to step 19. From the results we can see that *MoreAsync* does not outperform *ShrinkExec* by the initial change itself. However, it introduces more concurrency into the system and finally wins after configuration improvement.

Although PB chooses the *MoreAsync* option based on performance requirement, the opportunity for *ShrinkExec* has also been identified with its performance impact being estimated. This option could be useful if *MoreAsync* is finally determined inapplicable by the designer. All these change options were identified by PB in 19 steps with much less time than that taken by human experts.

8.3 300 Cases with Fixed Model Structure and Different Workload Parameters

Figure 8-13 shows a LQN model for a more complex web service system, which has a web server, three application servers and 4 underlying services. The figure shows the initial design and configuration of the system. Omitted execution demands for entries are 0.0ms (entries on a pseudo task), omitted call frequencies are 1.0, and omitted resource multiplicities are 1.

Figure 8-13 LQN model structure for random parameter study case

With the model structure fixed, different values of the execution demand of *Serv3_E1* ($\$dem31$) and of the mean number of calls from the three entries on WS to the three

entries on App2 (\$req1, \$req2 and \$req3) were generated, giving 300 cases. All combinations of the values shown in Table 8-5 were generated to give the 300 cases.

Table 8-5 Iterative generated values for four variables in 300 cases

Variable	Generation formula (from:to, step)	Values generated
\$dem31	(1:16, 5)	1, 6, 11, 16
\$req1	(0.2:5, 2)	0.2, 2.2, 4.2
\$req2	(0.2:10, 2)	0.2, 2.2, 4.2, 6.2, 8.2
\$req3	(0.5:10, 2)	0.5, 2.5, 4.5, 6.5, 8.5

Table 8-6 summarizes the automatic diagnosis and improvement results. All 300 cases are improved effectively, with improvement on throughput from 33% to 6352% higher. The entire process for handling 300 models finished within 5 hours, with less than 1 minute per case. Figure 8-14 shows the number of steps taken for each case by PB automatic diagnosis. Figure 8-15 shows the improvement on system throughput made by PB automatic diagnosis for each case.

The 300 cases demonstrate the robustness of PB. Performance expert knowledge is reused automatically and effectively.

Table 8-6 Diagnosis and improvement summary on the 300 cases

Total cases:	300
Average steps taken:	36
Average improvement in throughput: Improvement is calculated by: $\frac{f_{best} - f_{base}}{f_{base}}$ (f_{base} is base case throughput, f_{best} is best case throughput)	10.11 (1011%)
Max improvement %:	6352
Min improvement %:	33
Max steps:	76
Min steps:	12

Figure 8-14 Number of steps taken for 300 cases

Figure 8-15 Improvement on throughputs of 300 cases

8.4 An Industrial Case Study: Image Exchange System

In this section, we study on an Image Exchange System (IES), which is based on an industrial project by the author. The purpose of this project is to enable information sharing between partners. The images being transferred carry credential information, such as scans of identification cards, signatures, or legal documents. The functionality requirement for the system is to convert images that were originally in different formats and dimensions into files with a common format according to a partner's preferences, and then to drop off the generated files at a public server for the partner to pick up. The images were originally stored in a database system as raw binary data. For security reasons, the image files must be encrypted before being placed on the public server. The File Transfer Protocol (FTP) is used for file transfer. The partners are scheduled to pick up the image files on a 15 minute cycle. Therefore, a processing cycle is expected to be finished within 15 minutes.

The sequence diagram shown in Figure 8-16 describes the main scenario of the IES. Each actor `Client` models an operator who is representing one of the four partners. A `Client` invokes the service on the `Control Module (CtrlM)` to kick off a transmission processing cycle. The `CtrlM` then passes a set of source IDs to `ImgServer` for processing. A source ID indicates an image source that the partner required. A partner may require images from different sources. `ImgServer` loops for each source ID to generate image files for the partner. In order to generate the image files from a specific source, the `ImgServer` first retrieves control parameters for image rendering and then pulls raw data for each individual image. A standard Image Processing web service (`ImgWS`) is then invoked to render every image according to the previously retrieved control parameters. Rendered image data is written out to image files on the file system (`FileSys`). After all files have been generated on the `FileSys`,

the `ImgServer` sends a statistical report to the `Printer` and passes the control back to the `CtrlM`. The `CtrlM` then requests the `Transmission Controller` (`TransCtrl`) to transmit the image files to the public server (`PubServer`). The `TransCtrl` repeatedly pulls each image file from the `FileSys`, encrypts it with partner's public key and then sends the file to the `PubServer`.

The example uses an estimated volume for a processing cycle with an average of 10 partners. Each one requests images from an average of 4 sources, each of which provides an average of 100 images.

Table 8-7 Resource capacity limit for IES

Resource Type	Resource Name	Capacity Limit
Task	<code>CtrlM</code>	5
Task	<code>AppServ</code>	5
Task	<code>TransServ</code>	5
Processor	<code>TransP</code>	2
Processor	<code>AppP</code>	2

Figure 8-17 shows the LQN model manually created for the above described scenario. Capacity limits for hardware resources and software resources are shown in Table 8-7. Resources not listed in Table 8-7 have infinite capacity. *ShrinkExec* ratio (Δ_{cl}) is 0.2 for all tasks, and *MoreAsync* (Δ_{ma}) ratio is 0.5 for all tasks.

Figure 8-16 Sequence diagram of the image transfer scenario of IES

Figure 8-17 LQN model for the image transfer scenario of IES

Figure 8-18 PB diagnosis tree for IES

(\$c is the time for a cycle (desired value = 9.0E5 ms))

Figure 8-18 shows the diagnosis tree generated by PB with cycle time (\$c\$ noted in the figure) for the best design change option of each round. Bolded arrows and nodes represent the design improvement path in the tree. The bottlenecks identified at the end of each round are TransP at round 1 (step 10) and TransServ at all other rounds.

The cycle time is reduced from 2858870ms (47.6 minutes) to 196122ms (about 3.3 minutes) in 25 steps including configuration changes and design changes, an improvement of 93.13%. Design changes suggested by PB are:

1. Put some operations of task TransServ into second phase (step 6 to step 10)
2. Move task Encrypt away from processor TransP (step 10 to step 17)
3. Batching the network operation (Net) between TransServ and PubServ (step 17 to step 22)
4. Shrink task TransServ (step 22 to step 23)
5. Shrink server tasks of TransServ (step 23 to step 25)

At Step 22 the cycle time is already well within the desired value of 15 minutes, so the *shrinkExec* changes may not be strictly necessary.

CHAPTER 9 CONCLUSIONS

The thesis presents a rule-based Performance Booster (PB) framework for automatic software performance diagnosis and improvement. This research has followed the general process of developing a rule-based system, as introduced in section 3.4.3, to construct the framework. From Chapter 4 to Chapter 7, each chapter describes a stage towards building a performance knowledge system.

9.1 Achievements

This thesis has presented the first system to improve designs using a combination of performance evaluation by a model and rules for diagnosis and improvement. The rules were applied automatically and recursively by a Jess rule engine. Features of the system include:

- the use of well-known performance engineering principles;
- the evaluation of each design by its best configuration (found by PB);
- the introduction of new rules based on performance engineering insights;
- basing the search on quantitative evaluation of performance;
- search control parameters that can be adjusted by the user.

PB shows the ability to save a significant amount of effort for designers in performance diagnosis and improvements. The tool provides a step along the path towards improved performance, to identify critical design factors and architecture level solutions.

The rules required formalization of performance principles and attributes. The performance knowledge observed in the BSS example (Chapter 4) and the general principles (introduced in Chapter 2) were further expanded into a more complete set of rules and organized into formal representations in Chapter 5. Elements of performance requirements and their relationships were shown in a semantic network. A resource interaction model was defined to provide a context for performance diagnosis and mitigations on a computer system.

Root causes of performance issues in a software system were classified into bottleneck problems and long path problems. 14 abstract rules independent of performance modeling techniques were derived for locating either type of root causes and their mitigations, including control strategy between mitigation by configuration changes and mitigation by design changes.

A set of performance model surrogates for design properties were identified which enable encoding of the abstract rules into concrete rules in a Layered Queuing Network (LQN) performance model context.

The strategies for the PB framework are presented in Chapter 6. The capability of the rules encoded into PB, to identify performance problems and find solutions at the level of the performance model, has been demonstrated. The rules appear to be efficient, in the sense that (at least in the examples shown in Chapter 8) they find the most important changes first. The translation of these performance-model changes into meaningful design changes has also been described for each study case and in a general summary in section 6.7.

Thus the interpretation of performance model changes into design change recommendations appears to be a tractable problem.

The case study on the tutorial Web Application (section 8.1) shows an improvement of 46.8% (case 4) to 80.8% (case 6) in system response time with about 30 to 40 steps all

processed automatically. Revisiting the BSS example (section 8.2) demonstrates PB's capability of identifying the same design change options as done by a human expert. The diagnosis was done in 19 steps with much less time than that being taken by the human experts. Robustness of the approach was demonstrated over 300 cases of a complex Web Service system with fixed model structure and different workload parameters (section 8.3). All cases were improved, with 12 to 72 steps. The improvements are from 33% better, to 64 times better than the initial designs on throughput. An Image Exchange System (IES) based on an industrial project gained 93.13% improvement on cycle time. 5 design change steps were suggested by PB.

The rules for design change can be customized to a particular project by setting suitable search parameters and constraints. Design change options can be ranked by preferred indexes in order to assist in the choice of which design changes to apply.

9.2 Limitations and Assumptions

The use of PB is limited to improving one LQN model at a time. The prerequisite of applying PB is that a LQN model be available for the studied system. Multiple scenarios represented by separate models can be merged into a single LQN model, so this is not a limitation. It is assumed in this work that the LQN model is generated from scenarios defined in UML, and our experience is that most UML designs can be converted in this way. If entries are represented by activities in the LQN model, they must be converted to phase1-phase2 form to be processed by PB.

The advice from PB must still be interpreted by the designer. For instance *shrinkExec* modifies an execution-time budget, but the realization of the code and its execution time is still to be done.

It may be difficult for the designers to supply control parameters. However, once they have a solution, it is relatively simple for them to steer the solution by changing these parameters.

The accuracy of performance results obtained from the LQN model solvers has an important impact on the quality of design suggestions provided by PB. When using the simulation solver *LQSIM*, the simulation period has to be set to long enough in order to generate stable results. A confidence interval can be used as a quality indicator of the performance results obtained from *LQSIM*. Generally, a confidence interval of less than 1% with a 95% confidence level would be considered indicative of an accurate result. Greater than that, the results may vary between different solutions of the same model with the same configuration, thus may result in different improvement decisions by PB. The analytical solver *LQNS* always generates stable results. However, it shows limitation on solving models with specific operations, such as heavy asynchronous workload including second phase operations. The errors of *LQNS* results may mislead the decisions generated by PB.

9.3 Future Work

Now that feasibility is demonstrated, a more comprehensive set of rules is possible. A potential new change rule could provide caching. With a set of constraints such as a hit rate, a cache execution factor (the ratio of cache execution time over non-cache execution time), and a cacheable flag for each resource, it is possible to derive a caching rule that could automatically identify caching opportunities for performance improvement. The batching rule could be generalized.

Currently, the LQN models used accept point valued parameters as input and produce point valued performance measures as outputs. [Majumdar95] analyzed performance

models with interval values of performance measures and key system parameters. The model calculations are much more complex. The Software Performance Evaluation (SPE) method in [Smith02] also discussed using measurements of best case, worst case and average case values to provide ranges of performance. These methodologies can be integrated into future design of PB.

The rules for locating performance problems can equally be applied to measurement results, so it may be possible to apply the same rules in measurement-based tuning of the deployment and design, or to an integrated testing/modeling approach as in [Avritzer04].

PB can be extended to evaluate systems that use vendor provided components, and to make a selection among alternative components.

The PB platform can also be downsized to a configuration only version with all design change rules turned off, and to be used for configuration improvement of software systems.

REFERENCES

- [Andolfi00] F. Andolfi, F. Aquilani, S. Balsamo, P. Inverardi, “Deriving performance models of software architectures from message sequence charts”, Proceedings of the second international workshop on Software and performance, Ottawa, Ontario, Canada 2000, pp. 47-57.
- [Avritzer04] A. Avritzer, E.J. Weyuker, “The Role of Modeling in the Performance Testing of E-Commerce Applications”, IEEE Trans. Software Eng., v 30, n 12, pp 1072-1083, 2004.
- [Asogawa90] M. Asogawa, A. Shiramizu, H. Kubo, “A computer system performance analysis expert system (EXPERFORM)”, Ninth Annual International Phoenix Conference on Computers and Communications, pp. 785-765, 1990.
- [Balbo88] G. Balbo, S.C. Bruell, S. Ghanta, “Combining Queueing Networks and Generalized Stochastic Petri Nets for the Solution of Complex Models of System Behavior”, IEEE transactions on computers, Vol 37, No. 10, Oct 1988, pp 1251-1268
- [Balsamo04] S. Balsamo, A. Di Marco, P. Inverardi, and M. Simeoni, “Model- Based Performance Prediction in Software Development: A Survey”, IEEE transactions on software engineering, vol. 30, No. 5, pp. 295-310, May 2004.
- [Bernardi02] S. Bernardi, S. Donatelli, J. Merseguer, “From UML Sequence Diagrams and Statecharts to Analysable Petri Net Models”, ACM Proc. Int’l Workshop Software and Performance, pp. 35-45, 2002.

[Cortellessa00] V. Cortellessa and R. Mirandola, "Driving a Queueing Network Based Performance Model from UML Diagrams", *ACM Proc. Int'l Workshop Software and Performance*, pp. 58-70, 2000.

[Davis77] R. Davis, B.G. Buchanan, and E.H. Shortliffe, "Production Systems as a Representation for a Knowledge-Based Consultation Program," *Artificial Intelligence* 8 (1977): 15-45.

[Dilley98] J. Dilley, R. Friedrich, T. Jin, and J. Rolia, "Web Server Performance Measurement and Modeling Techniques", *Performance Evaluation Journal*, Volume 33, pages 5-26, 1998.

[Forgy82] C. Forgy, "Rete: A Fast Algorithm for the Many Pattern/Many Object Pattern Match Problem", *Artificial Intelligence*, 19, pp 17-37, 1982

[Franks06] G. Franks, D.C. Petriu, C.M. Woodside, J. Xu, P. Treguno, "Layered Bottlenecks and Their Mitigation", *Proc. 3rd Int. Conference on Quantitative Evaluation of Systems (QEST'2006)*, pp. 103-114, Riverside, CA, Sept. 2006.

[Franks00] G. Franks, "Performance Analysis of Distributed Server Systems", Report OCIEE-00-01, Jan. 2000, PhD. thesis, Carleton University.

[Franks98] G. Franks and C.M. Woodside, "Performance of Multi-level Client-Server Systems with Parallel Service Operations", *Proc. First Int. Workshop on Software and Performance (WOSP98)*, pp. 120-130, Santa Fe, October 1998.

[Franks96] G. Franks, S. Majumdar, J.E. Neilson, D.C. Petriu, J.A. Rolia and C.M. Woodside, "Performance Analysis of Distributed Server Systems", *Proc. Sixth International Conference on Software Quality*, Ottawa, Canada, October 28-30, 1996, pp. 15-26.

- [Franks95] G. Franks, A. Hubbard, S. Majumdar, D.C. Petiu, J. Rolia, and C.M. Woodside, "A Toolset for Performance Engineering and Software Design of Client-Server Systems", *Performance Evaluation*, vol. 24, nos. 1-2, pp. 117-135, 1995.
- [Friedman-Hill08] E. Friedman-Hill, "Jess, the Rule Engine for the Java Platform", <http://herzberg.ca.sandia.gov/jess/>, last modified on Aug 6, 2008.
- [Friedman-Hill03] E. Friedman-Hill, "Jess in Action, Rule-based Systems in Java", Manning publications, 2003.
- [Gomaa00a] H. Gomaa, "Designing Concurrent, Distributed, and Real-Time Applications with Uml", 1st edition, Addison-Wesley Longman Publishing Co., Inc, 2000.
- [Gomaa00b] H. Gomaa, D.A. Menascé, "Design and performance modeling of component interconnection patterns for distributed software architectures", *Proceedings of the second international workshop on Software and performance*, pp. 117-126, sept 2000.
- [Gorton03] I. Gorton, A. Liu, "Evaluating the Performance of EJB Components", *IEEE Internet Computing* Vol. 7, Issue 3, pp.18-23, May 2003.
- [Gu02] G. Gu and D.C. Petriu, "XSLT Transformation from UML models to LQN Performance Models", *ACM Proc. Int'l Workshop Software and Performance*, pp. 227-234, 2002.
- [Hellerstein01] J.L. Hellerstein, "A General-Purpose Algorithm for Quantitative Diagnosis of Performance Problems", *Journal of Network and Systems Management*, 2001.
- [Helm95] B.R. Helm, A.D. Malony, S. Fickas, "Capturing and automating performance diagnosis: the Poirot approach", pp606-613, *IPPS* 1995.

[Hoogenboom92] P.J. Hoogenboom, "System performance advisor: An expert system for unix system performance management", Dept. Computer Science, University of Utah, Salt Lake City, p. 144, 1992.

[Jackson98] P. Jackson, "Introduction to Expert Systems" (1998), ISBN 0-201-87686-8.

[Jogalekar00] P. Jogalekar, C.M. Woodside, "Evaluating the Scalability of Distributed systems", IEEE Trans. on Parallel and Distributed Systems, v 11 n 6 pp 589-603, June 2000.

[Kahkipuro01] P. Kahkipuro, "UML-Based Performance Modeling Framework for Component-Based Distributed Systems", Proc. Performance Eng. Conf., pp 167-184, 2001.

[Karavanic99] K. Karavanic, B. Miller, "Improving Online Performance Diagnosis by the Use of Historical Performance Data", Proceedings of the IEEE/ACM SC99 Conference, November 13 - 18, Portland, Oregon, USA, 1999.

[Litoiu98] M. Litoiu, J. Rolia, G. Serazzi,, "Designing Process Replication and Threading Policies: A Quantitative Approach", 10th International Conference on Modelling Techniques and Tools for Computer Performance Evaluation (Tools '98), Lectures Notes in Computer Science, Ramon Puigjaner, Nunzio N. Savino, Bartolomeu Sera (Editors), Vol. 1469, pp.15-26, 1998.

[Liu04] Y. Liu, A. Fekete, I. Gorton, "Predicting the performance of middleware-based applications at the design level", Proceedings of the fourth international workshop on Software and performance 2004, pp166-170, Redwood Shores, California, January 14 - 16, 2004.

[Maly00] P. Maly, C.M. Woodside, "Layered Modeling of Hardware and Software, with Application to a LAN Extension Router", Proc 11th Int. Conf on Computer Performance Evaluation Techniques and Tools (TOOLS 2000), pp 10-24, Chicago, March 2000.

- [Majumdar95] S. Majumdar, R. Ramadoss, "Interval-based performance analysis of computing systems", proceedings of the Third International Workshop on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems 1995, MASCOTS '95, pp 345-351, Durham, NC, USA, Jan 1995.
- [Melle79] W.V. Melle, "A Domain-Independent Production Rule System for Consultation Programs", International Joint Conference on Artificial Intelligence, pp923–925, 1979.
- [Menasce00] D. Menasce and H. Gomaa, "A Method for Design and Performance Modeling of Client/Server Systems", IEEE Transactions on Software Engineering, vol. 26, no. 11 pp. 1066-1085, 2000.
- [Menasce98] D. Menasce, H. Gomaa, "On a Language Based Method for Software Performance Engineering of Client/Server System", ACM Proc. Int'l Workshop Software and Performance, pp. 63-69, 1998.
- [Merseguer01] J. Merseguer, J. Campos, E. Mena, "Performance analysis of Internet based software retrieval systems using Petri Nets", Proceedings of the 4th ACM International Workshop on Modeling, Analysis and Simulation of Wireless and Mobile Systems, pp 47 – 56, Rome, Italy, 2001.
- [Miller95] B.P. Miller, M.D. Callaghan, J.M. Cargille, J.K. Hollingsworth, R.B. Irvine, K.L. Karavanic, K. Kunchithapadam, and T. Newhall, "The Paradyn Parallel Performance Measurement Tool", IEEE Computer, 28(11):37--46, 1995.
- [Neilson95] J.E. Neilson, C.M. Woodside, D.C. Petriu and S. Majumdar, "Software Bottlenecking in Client-Server Systems and Rendez-vous Networks", IEEE Trans. On Software Engineering, Vol. 21, No. 9, pp. 776-782, September 1995.
- [OMG08a] Object Management Group, "UML Resource Page", <http://www.uml.org/>, last updated on Jan 15, 2008.

[OMG08b] Object Management Group, “Modeling and Analysis of Real-time and Embedded systems”, <http://www.omgmar.te.org/>, last updated on Jan 23, 2008.

[OMG02] Object Management Group, “UML Profile for Schedulability, Performance, and Time Specification”, OMG Adopted Specification ptc/02-03-02, July 1, 2002.

[Pan96] A. Pan, “Solving Stochastic Rendezvous Networks of Large Client-Server Systems with Symmetric Replication”, M.Eng thesis, 1996.

[Parsons08] T. Parsons and J. Murphy: “Detecting Performance Antipatterns in Component Based Enterprise Systems”, Journal of Object Technology, vol. 7, no. 3, pp. 55-90, [http://www.jot.fm/issues/issue 2008 03/article1/](http://www.jot.fm/issues/issue%2008%2003/article1/), March - April 2008.

[Petri04] Petri nets tools database, <http://www.daimi.au.dk/PetriNets>, 2004

[PetriuDB07] D.B. Petriu and C.M. Woodside, “An intermediate metamodel with scenarios and resources for generating performance models from UML designs”, Software and Systems Modeling, issue 2007/02, pp. 163-184.

[PetriuDB04] D.B. Petriu and C.M. Woodside, "A Metamodel for Generating Performance Models from UML Designs, Proc UML 2004, Lisbon, v. 3273 of Lecture Notes in Computer Science (LNCS 3273), pp 41-53, Oct 2004.

[PetriuDB03] D.B. Petriu, D. Amyot, and C.M. Woodside, “Scenario-Based Performance Engineering with UCMNav”, *11th SDL Forum (SDL'03)*, LNCS 2708, pp18-35, Stuttgart, Germany, July 2003.

[PetriuDB02a] D.B. Petriu, C.M. Woodside, “Software Performance Models from System Scenarios in Use Case Maps”, Proc. 12 Int. Conf. on Modeling Tools and Techniques for Computer and Communication System Performance Evaluation (Performance TOOLS 2002), London, April 2002.

[PetriuDB02b] D.B. Petriu, C.M. Woodside, “Analysing Software Requirements Specifications for Performance”, Proc. 3rd Int. Workshop on Software and Performance, Rome, pp 1 – 9, July 2002.

[PetriuDC06] D.C. Petriu, A. Sabetta, “From UML to Performance Analysis Models by Abstraction-raising Transformation”, From MDD Concepts to Experiments and Illustrations, (eds. J.P. Babau, J. Champeau, S. Gerard), ISTE Ltd., pp.53-70, 2006

[PetriuDC02] D.C. Petriu, H. Shen, “Applying the UML Performance Profile: Graph Grammar based derivation of LQN models from UML specifications”, in Computer Performance Evaluation - Modelling Techniques and Tools, (Tony Fields, Peter Harrison, Jeremy Bradley, Uli Harder, Eds.) Lecture Notes in Computer Science 2324, pp.159-177, Springer Verlag, 2002.

[PetriuDC99] D.C. Petriu and X. Wang, “From UML Description of High-Level Software Architecture to LQN Performance Models”, Proc. Applications of Graph Transformations with Industrial Relevance Workshop (AGTIVE’99), pp. 47-62, 1999.

[Pooley00] R. Pooley, “Software Engineering and Performance: a Roadmap”, in The Future of Software Engineering, part of the 22nd Int. Conf. on Software Engineering (ICSE2000), pp. 189-200, Limerick, Ireland, June 2000.

[PUMA08] PUMA (Performance from Unified Model Analysis), www.sce.carleton.ca/rads/puma/, last accessed on Sept 13, 2008..

[Reed92] D.A. Reed, R.A. Aydt, T.M. Madhyastha, R.J. Noe, K.A. Shields, B.W. Schwartz, “An Overview of the Pablo performance Analysis Environment”, Technical Report, University of Illinois, Department of Computer Science, 1992

[Riley08] G. Riley, CLIPS Home Page, <http://clipsrules.sourceforge.net/>, last updated on May 9, 2008

- [Risse04] T. Risse, K. Aberer, A. Wombacher, M. Surridge, and S. Tayllor, "Configuration of distributed message converter systems", *Performance Evaluation*, vol. 58, no. 1, pp. 43-80, Oct. 2004.
- [Rolia95] J.A. Rolia , K.C. Sevcik, "The Method of Layers", *IEEE Transactions on Software Engineering*, v.21 n.8, p.689-700, August 1995
- [Smith02] C.U. Smith and L.G. Williams, "Performance Solutions: A Practical Guide to Creating Responsive, Scalable Software", Addison-Wesley, 2002.
- [Smith00] C.U. Smith and L.G. Williams, "Software Performance Antipatterns", in *Proceedings of the Second International Workshop on Software and Performance (WOSP2000)*, pp. 127-136, Ottawa, Canada, September 17-20, 2000.
- [Smith90] C.U. Smith, "Performance Engineering of Software Systems", Addison-Wesley Pub. Co., c1990.
- [Smith86] C.U. Smith, "Independent General Principles for Constructing Responsive Software Systems", *ACM Transactions on Computer Systems*, volume 4, issue 1, pp 1-31, February 1986.
- [Sowa92] J.F. Sowa, "Semantic Networks", *Encyclopedia of Artificial Intelligence*, Ed. Stuart C Shapiro, Wiley, 1987, second edition, 1992.
- [Tregunno03] P. Tregunno, "Practical Analysis of Software Bottlenecks", MASC thesis May 2003.
- [Trie08] Trie – Wikipedia, the free encyclopedia, <http://en.wikipedia.org/wiki/Trie>, last modified on September 4, 2008.
- [Williams98] L.G. Williams, C.U. Smith, "Performance Evaluation of Software Architectures", *Proc. ACM Int'l Workshop Software and Performance*, pp. 179-189, 1998.

[Woodside05] C.M. Woodside, D.C. Petriu, D.B. Petriu, H. Shen, T. Israr, J. Merseguer, “Performance by Unified Model Analysis (PUMA)”, Proc. of the 5th ACM Workshop on Software and Performance WOSP'2005, pp. 1-12, Palma, Spain, July 11-14, 2005.

[Woodside03] C.M. Woodside, D.C. Petriu, “Performance Analysis with UML”, Chapter 11 in “UML for Real: Design of Embedded Real-Time Systems”, Editors: Luciano Lavagno, Grant Martin, and Bran Selic, Kluwer Academic Publisher, New York, to be published in 2003.

[Woodside02] C.M. Woodside, D.B. Petriu, K.H. Siddiqui, , “Performance-related Completions for Software Specifications”, Proc 24th Int. Conf. on Software Engineering (ICSE 2002), May 2002.

[Woodside01] C.M. Woodside, “Software Resource Architecture”, Int. Journal on Software Engineering and Knowledge Engineering (IJSEKE), v 11, pp 407-429, 2001.

[Woodside95a] C.M. Woodside, J.E. Neilson, D.C. Petriu and S. Majumdar, “The Stochastic Rendezvous Network Model for Performance of Synchronous Client-Server-Like Distributed Software”, IEEE Transactions on Computers, Vol. 44, No. 1, pp. 20-34, January 1995.

[Woodside95b] C.M. Woodside, “A Three-View Model for Performance Engineering of Concurrent Software”, IEEE Trans. On Software Engineering, Vol. 21, No. 9, pp. 754-767, Sept. 1995

[Wu04] X. Wu and C.M. Woodside, “Performance Modeling from Software Components”, in Proc. 4th Int. Workshop on Software and Performance (WOSP 04) , pp. 290-301, Redwood Shores, Calif., Jan 2004.

[Xu08] J. Xu, “Rule-based Automatic Software Performance Diagnosis and Improvement”, Proc. of Workshop on Software Performance (WOSP 2008), pp 1 – 12, Princeton, USA, 2008.

[Xu05a] J. Xu, C.M. Woodside, “Template Driven Performance Modeling of Enterprise Java Beans”, Proc. of 2005 Middleware for Web Services (MWS 2005) Workshop at the EDOC 2005 Conference, pp57-64, Enschede, the Netherlands, Sept 2005.

[Xu05b] J. Xu, A. Oufimtsev, C.M. Woodside, L. Murphy, “Performance Modeling and Prediction of Enterprise JavaBeans with Layered Queuing Network Templates”, Proc. of Specification and Verification of Component-Based Systems (SAVCBS 2005) Workshop at ESEC/FSE’05, pp 39 – 46, Lisbon, Portugal, Sept 2005.

[Xu03] J. Xu, C.M. Woodside, D.C. Petriu, “Performance analysis of a software design using the UML profile for schedulability, performance and time”, Proc. Computer Performance Evaluation, Modelling Techniques and Tools (TOOLS), LNCS 2794, pp291-310, 2003.

[Zheng03] T. Zheng, C.M. Woodside, “Heuristic Optimization of Scheduling and Allocation for Distributed Systems with Soft Deadlines”, Proc. 13th Int Conf. on Computer Performance Evaluation, Modelling Techniques and Tools (TOOLS 2003), Urbana, Illinois, USA, Sept 2003, pp 169-181, vol. LNCS 2794, Lecture Notes in Computer Science, Springer-Verlag, 2003.

APPENDIX A PB JESS RULE BASE

```

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Jess Automatic Software Performance Diagnosis
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;*****
;;* Module MAIN *
;;*****

;; Load our Userfunctions
(load-package lgntools.perfAnalyzer.jessplugin.DiagnosisFunctions)

;; Define Templates
(deftemplate modeltransform
  (slot model)
  (slot problemtype)
  (slot problemname)
  (slot transform)
  (slot log)
  (slot status))

(deftemplate processor
  (slot name)
  (slot sat-level))

(deftemplate task
  (slot name)
  (slot processor)
  (slot sat-level))

(deftemplate bottleneck
  (slot type)

```

```

(slot name))

(deftemplate configchange
  (slot resourcetype)
  (slot name)
  (slot newvalue))

(deftemplate structurechange
  (slot resourcetype)
  (slot name)
  (slot description))

(deftemplate changelog
  (slot log))

(deftemplate question
  (slot text)
  (slot type)
  (slot ident))

(deftemplate answer
  (slot ident)
  (slot text))

(deftemplate recommendation
  (slot transform)
  (slot explanation))

;; Define Initial Facts
(deffacts question-data
  "The questions the system can ask."
  (question (ident choice) (type number)
    (text "What is your choice?")))

;; Define Global Values

```



```

(defglobal ?*crlf* = "
")
(defglobal ?*proc-satu-threshold* = 0.9)
(defglobal ?*task-satu-threshold* = 0.9)
(defglobal ?*global-codetighten-ratio* = 0.2)
(defglobal ?*global-asyn-ratio* = 0.5)

;; Deffunctions
(defun proc-saturated (?s)
  (return (> ?s ?*proc-satu-threshold*)))

(defun task-saturated (?s)
  (return (> ?s ?*task-satu-threshold*)))

;;*****
;;* Module diagnosis *
;;*****

(defmodule diagnosis)

(defrule detect-hardware-bottleneck
  ; if a processor is saturated, it is a bottleneck
  (processor (name ?p)
    (sat-level ?psl&:(proc-saturated ?psl)))
  =>
  (assert (bottleneck (type "hardware") (name ?p)))
  (printout t "hwbn " ?p " asserted" crlf))

(defrule detect-software-bottleneck
  ; if a task is saturated, no processor is saturated
  ; and no server task is saturated,
  ; the task is a software bottleneck
  ; the rule guarantee sfbn occur only when no hwbn exists
  (task (name ?t) (sat-level ?tsl&:(task-saturated ?tsl)))
  (not (processor (sat-level ?psl&:(proc-saturated ?psl))))

```

```

(not (task (sat-level ?tsl2&:(task-saturated ?tsl2))
      (name ?t2&~?t &:(is-server ?t2 ?t))))

=>

(assert (bottleneck (type "software") (name ?t)))
(printout t "sfbn " ?t " asserted" crlf))

;;*****
;;* Configurational Mitigation *
;;*****
(defmodule configmitigation)

(defrule addresource-hwbn
  ?r <- (bottleneck (type "hardware")
                  (name ?p
                    &:(has-capacity "processor" ?p)))

  =>
  (retract ?r)
  (bind ?n (increase-multiproc ?p))
  (assert (configchange (resourcetype "processor")
                      (name ?p)
                      (newvalue ?n))))

(defrule addresource-sfbn
  ?r <- (bottleneck (type "software")
                  (name ?t &:(has-capacity "task" ?t)))

  =>
  (retract ?r)
  (bind ?n (increase-multitask ?t))
  (assert (configchange (resourcetype "task")
                      (name ?t)
                      (newvalue ?n))))

;;*****

```

```

;;* Structural Mitigation *
;;*****
(defmodule structuremitigation)
; Structural mitigation should targeting one problem
; resource at a time.
; this is for purpose of evaluating transform efficiency.
; if mitigation involves multiple resources at one
; transform, it will be hard to evaluate and roll back or
; do partial transform.
(defrule structuralmitigation-hwbn
  ?r <- (bottleneck (type "hardware") (name ?p))
  =>
  (retract ?r)
  (printout t "structuralmitigation-hwbn is fired: handling " ?p crlf)
  (bind ?m (get-curmodel))
  (bind ?m1 (clone-model ?m))
  (assert (modeltransform
            (model ?m1)
            (problemtype "processor")
            (problemname ?p)
            (transform codetighten)
            (log (str-cat "Hardware bottleneck " ?p " cannot be solved:
try code tightening" ?*crlf*))
            (status try)))
  (bind ?m2 (clone-model ?m))
  (assert (modeltransform
            (model ?m2)
            (problemtype "processor")
            (problemname ?p)
            (transform movetask-min)
            (log (str-cat "Hardware bottleneck " ?p " cannot be solved:
try redeploy(min)" ?*crlf*))
            (status try)))
  (bind ?m3 (clone-model ?m))
  (assert (modeltransform

```

```

        (model ?m3)
        (problemtype "processor")
        (problemname ?p)
        (transform movetask-max)
        (log (str-cat "Hardware bottleneck " ?p " cannot be solved:
try redeploy(max)" ?*crlf*))
        (status try)))

(defrule structuralmitigation-sfbn
  ?r <- (bottleneck (type "software") (name ?t))
  =>
  (retract ?r)
  (printout t "structuralmitigation-sfbn is fired: handling " ?t crlf)
  (bind ?m (get-curmodel))
  (bind ?m1 (clone-model ?m))
  (assert (modeltransform
    (model ?m1)
    (problemtype "task")
    (problemname ?t)
    (transform codetighten)
    (log (str-cat "Software bottleneck " ?t " cannot be solved:
try code tightening" ?*crlf*))
    (status try)))
  (bind ?m2 (clone-model ?m))
  (assert (modeltransform
    (model ?m2)
    (problemtype "task")
    (problemname ?t)
    (transform increaseasyn)
    (log (str-cat "Software bottleneck " ?t " cannot be solved:
try increase asynchronous" ?*crlf*))
    (status try)))
  (bind ?m3 (clone-model ?m))
  (assert (modeltransform
    (model ?m3)

```

```

        (problemtype "task")
        (problemname ?t)
        (transform partition)
        (log (str-cat "Software bottleneck " ?t " cannot be solved:
try partition" ?*crlf*))
        (status try)))
    (bind ?m4 (clone-model ?m))
    (assert (modeltransform
        (model ?m4)
        (problemtype "task")
        (problemname ?t)
        (transform batching)
        (log (str-cat "Software bottleneck " ?t " cannot be solved:
try batching" ?*crlf*))
        (status try))))

;;*****
;;* Structural Model Transform *
;;*****
(defmodule transmodel)

(defrule codetighten-hwbn
; Tighten tasks on hwbn processor
?r <- (modeltransform (model ?m)
        (problemtype "processor")
        (problemname ?p)
        (transform codetighten)
        (log ?l)
        (status ?s))
        (task (name ?t) (processor ?p))
=>
    (if (codetighten-task ?m ?t) then
        (printout t "task " ?t " tightened" crlf)

```

```

        (modify ?r (log (str-cat ?l "Do code tightening on task
" ?t ?*crlf*)))

        (status changed))

else
  (if (eq ?s try) then
    (printout t "cannot tighten" ?t crlf)
    (modify ?r (status nochange)))
  )
)

(defrule movetask-min-hwbn
; Move the smallest task to the most lightly loaded
; processor
  ?r <- (modeltransform (model ?m)
    (problemtype "processor")
    (problemname ?p)
    (transform movetask-min)
    (log ?l)
    (status try))
    (processor (name ?p2&~?p)
      (sat-level ?s2))
  (not (processor (name ?p3&~?p2&~?p)
    (sat-level ?s3:(< ?s3 ?s2))))
  (task (name ?t) (processor ?p))
  (task (name ?t2&~?t) (processor ?p))
  (not(task (name ?t3&~?t &:
    (is-smaller-task ?t3 ?t))
    (processor ?p)))

=>
  (move-task ?m ?t ?p2)
  (printout t "task " ?t " moved" crlf)
  (modify ?r (log (str-cat ?l "Move task " ?t " from processor " ?p
" to processor " ?p2 ?*crlf*)))
  (status done)))

```

```

(defrule movetask-max-hwbn
; Move the largest task to the most lightly loaded processor
?r <- (modeltransform (model ?m)
                    (problemtype "processor")
                    (problemname ?p)
                    (transform movetask-max)
                    (log ?l)
                    (status try))
      (processor (name ?p2&~?p) (sat-level ?s2))
      (not (processor (name ?p3&~?p2&~?p)
                    (sat-level ?s3&:(< ?s3 ?s2))))
      (task (name ?t) (processor ?p))
      (task (name ?t2&~?t) (processor ?p))
      (not(task (name ?t3&~?t &:
                    (is-smaller-task ?t ?t3))
                (processor ?p)))
      =>
      (move-task ?m ?t ?p2)
      (printout t "task " ?t " moved" crlf)
      (modify ?r (log (str-cat ?l "Move task " ?t " from processor " ?p " to
processor " ?p2 ?*crlf*)))
      (status done)))

(defrule movetask-min-hwbn-fail
; No task can be moved because only one task is on the
; processor
?r <- (modeltransform (model ?m)
                    (problemtype "processor")
                    (problemname ?p)
                    (transform movetask-min)
                    (log ?l)
                    (status try))
      (task (name ?t) (processor ?p))
      (not (task (name ?t2&~?t) (processor ?p)))
      =>

```

```

(printout t "No task can be moved away from " ?p crlf)
(modify ?r (status nochange)))

(defrule movetask-max-hwbn-fail
; No task can be moved because only one task is on the
; processor
?r <- (modeltransform (model ?m)
                      (problemtype "processor")
                      (problemname ?p)
                      (transform movetask-max)
                      (log ?l)
                      (status try))
(task (name ?t) (processor ?p))
(not (task (name ?t2&~?t) (processor ?p)))
=>
(printout t "No task can be moved away from " ?p crlf)
(modify ?r (status nochange)))

(defrule codetighten-sfbn
?r <- (modeltransform (model ?m)
                      (problemtype "task")
                      (problemname ?t)
                      (transform codetighten)
                      (log ?l)
                      (status try))
=>
(if (codetighten-task ?m ?t) then
  (modify ?r (log (str-cat ?l "Do code tightening on task
" ?t ?*crlf*)))
  (status done))
else
  (modify ?r (status nochange)))
)

(defrule increaseasyn-sfbn

```



```

?r <- (modeltransform (model ?m)
                        (problemtype "task")
                        (transform increaseasyn)
                        (log ?l)
                        (status try)
                        (problemname ?t))

=>
  (if (increaseasyn-targetentries ?m ?t) then
    (modify ?r (log (str-cat ?l "Increase asynchronous on target
entries of task " ?t ?*crlf*)))
    (status done))
  else
    (modify ?r (status nochange)))
)

(defrule partition-sfbn
?r <- (modeltransform (model ?m)
                        (problemtype "task")
                        (problemname ?t)
                        (transform partition)
                        (log ?l)
                        (status try))

=>
  (bind ?msg (partition-task ?m ?t))
  (if (eq ?msg "") then
    (retract ?r)
    (printout t "partition-sfbn is retracted" crlf)
  else
    (modify ?r (log (str-cat ?l "Partitioning task " ?t ?*crlf* ?msg))
              (status done))))

(defrule batching-sfbn
?r <- (modeltransform (model ?m)
                        (problemtype "task")
                        (problemname ?t)

```

```

                                (transform batching)
                                (log ?l)
                                (status try))

=>
  (if (batching-sfbn ?m ?t) then
      (modify ?r (log (str-cat ?l ": Batching on outgoing calls of
task " ?t ?*crlf*)))
      (status done))
  else
      (modify ?r (status nochange)))
)

(defrule retract-transformfailures
  ?r <- (modeltransform (status nochange))
  =>
    (printout t "retract transform" ?r crlf)
    (retract ?r))

;;*****
;;* Module changelog      *
;;*****
(defmodule changelog)

(defrule register-configchangelog
  (configchange (resourcetype ?t) (name ?n) (newvalue ?m))
  =>
    (assert (changelog(log (str-cat ?t ?n " is set to multiplicity of
" ?m)))))

(defrule register-structurechangelog
  (modeltransform (log ?l))
  =>
    (assert (changelog(log ?l))))

(defrule combine-log

```

```

?r1 <- (changelog (log ?l))
?r2 <- (changelog (log ?l2&:(neq ?l ?l2)))
=>
(retract ?r2)
(modify ?r1 (log (str-cat ?l ?*crlf* ?l2))))

;; (set-current-module MAIN)

;;*****
;;* Functions *
;;*****
(deffunction config-model ()
  (focus diagnosis configmitigation changelog)
  (run))

(deffunction transform-model()
  (focus diagnosis structuremitigation transmodel)
  (run))

```

APPENDIX B LQN MODELS

Building Security System (Base Case)

G "Building Security System" .000001 100 1 0.9 -1

P 0

p UserCPU f i

p ApplicCPU f

p Buf_CPU f i

p DB_CPU f

p SCR_CPU f

p DLA_CPU f

p DiskCPU f m 2

p NetProc f i

-1

T 0

t UserT n User -1 UserCPU

t SCR_T n SCR_E -1 SCR_CPU

t AccCtrl_T n AccCtrl_E -1 ApplicCPU

t DB_T n DBReadInfo_E DBWriteInfo_E DBWriteVid_E -1 DB_CPU m 10

t Disk_T n DkReadInfo_E DkWriteInfo_E DkWriteVid_E -1 DiskCPU m 2

t DLA_T n DLA_E -1 DLA_CPU

t VidCtrl_T r VidCtrl_E -1 ApplicCPU

t AcqProc_T n AcqProc_E -1 ApplicCPU

t BufManag_T n BufGet_E -1 Buf_CPU

t BufManag2_T n BufRel_E -1 Buf_CPU

t Buffer_T n Buffer_E -1 Buf_CPU

t GetImg_T n GetImg_E -1 ApplicCPU

t NetOP_T n NetOP_E -1 NetProc i

t PassImg_T n PassImg_E -1 ApplicCPU

```
t StoreProc_T n    StoreProc_E -1 ApplicCPU
-1
```

```
E 0
a User 0.0005
s User 0 0 0 -1
y User SCR_E 0 1 -1
f User 0 1 -1
M User 0 1000 -1
s SCR_E      1 0 -1
y SCR_E      AccCtrl_E 1 0 -1
f SCR_E      1 0 -1
s AccCtrl_E 3.9 0.2 -1
y AccCtrl_E DBReadInfo_E 1 0 -1
f AccCtrl_E 1 0 -1
s DBReadInfo_E 1.8 0 -1
y DBReadInfo_E DkReadInfo_E 0.4 0 -1
s DkReadInfo_E 1.5 0 -1
F AccCtrl_E DLA_E 1 -1
s DLA_E      0 500 -1
y AccCtrl_E DBWriteInfo_E 0 0.2 -1
s DBWriteInfo_E 1.8 0 -1
y DBWriteInfo_E DkWriteInfo_E 1 0 -1
f DBWriteInfo_E 1 0 -1
s DkWriteInfo_E 3 0 -1

Z VidCtrl_E 0 0 0 -1
s VidCtrl_E 1.8 -1
M VidCtrl_E 1000 -1
y VidCtrl_E AcqProc_E 100 -1
f VidCtrl_E 1 -1
s AcqProc_E 1.5 0 -1
y AcqProc_E BufGet_E 1 0 -1
f AcqProc_E 1 0 -1
s BufGet_E 0.5 0 -1
```

```

F BufGet_E Buffer_E 1 -1
f BufGet_E 1 0 -1
s Buffer_E 0 0 -1
y Buffer_E GetImg_E 1 0 -1
f Buffer_E 1 1 -1
s GetImg_E 12 0 -1
y GetImg_E NetOP_E 8 0 -1
s NetOP_E 0 1 -1
y Buffer_E PassImg_E 1 0 -1
s PassImg_E 0.9 0 -1
y Buffer_E StoreProc_E 0 1 -1
s StoreProc_E 3.3 0 -1
y StoreProc_E DBWriteVid_E 1 0 -1
f StoreProc_E 1 0 -1
s DBWriteVid_E 7.2 0 -1
y DBWriteVid_E DkWriteVid_E 8 0 -1
s DkWriteVid_E 1 0 -1
y StoreProc_E BufRel_E 1 0 -1
s BufRel_E 0.5 0 -1
-1

```

Building Security System (PB version)

```
G "Building Security System" .000001 100 1 0.9 -1
```

```

P 0
p UserCPU f i
p ApplicCPU f i
p Buf_CPU f i
p DB_CPU f
p SCR_CPU f
p DLA_CPU f
p DiskCPU f m 2
p NetProc f i

```

-1

T 0

```
t UserT      n User -1 UserCPU
t SCR_T      n SCR_E      -1 SCR_CPU
t AccCtrl_T n AccCtrl_E -1 ApplicCPU
t DB_T n DBReadInfo_E DBWriteInfo_E DBWriteVid_E -1 DB_CPU m 10
t Disk_T n DkReadInfo_E DkWriteInfo_E DkWriteVid_E -1 DiskCPU m 2
t DLA_T      n DLA_E      -1 DLA_CPU
t VidCtrl_T r VidCtrl_E -1 ApplicCPU
t AcqProc_T n AcqProc_E -1 ApplicCPU
t BufManag_T n BufGet_E -1 Buf_CPU
t BufManag2_T n BufRel_E -1 Buf_CPU
t Buffer_T n Buffer_E -1 Buf_CPU
t GetImg_T n GetImg_E -1 ApplicCPU
t NetOP_T n NetOP_E -1 NetProc i
t PassImg_T n PassImg_E -1 ApplicCPU
t StoreProc_T n StoreProc_E -1 ApplicCPU
-1
```

E 0

```
a User 0.0005
s User 0 -1
y User SCR_E 1 -1
f User 1 -1
s SCR_E      1 0 -1
y SCR_E      AccCtrl_E 1 0 -1
f SCR_E      1 0 -1
s AccCtrl_E 3.9 0.2 -1
y AccCtrl_E DBReadInfo_E 1 0 -1
f AccCtrl_E 1 0 -1
s DBReadInfo_E 1.8 0 -1
y DBReadInfo_E DkReadInfo_E 0.4 0 -1
s DkReadInfo_E 1.5 0 -1
F AccCtrl_E DLA_E 1 -1
```

```

s DLA_E      0 500 -1
y AccCtrl_E DBWriteInfo_E 0 0.2      -1
s DBWriteInfo_E  1.8 0 -1
y DBWriteInfo_E  DkWriteInfo_E 1    0 -1
f DBWriteInfo_E  1 0 -1
s DkWriteInfo_E  3 0 -1

```

```

Z VidCtrl_E 0 -1
s VidCtrl_E 1.8 -1
y VidCtrl_E AcqProc_E 100 -1
f VidCtrl_E 1      -1
s AcqProc_E 1.5    0 -1
y AcqProc_E BufGet_E 1 0 -1
f AcqProc_E 1 0    -1
s BufGet_E 0.5 0 -1
F BufGet_E Buffer_E 1 -1
f BufGet_E 1 0 -1
s Buffer_E 0 0 -1
y Buffer_E GetImg_E 1 0 -1
f Buffer_E 1 1 -1
s GetImg_E 12 0    -1
y GetImg_E NetOP_E 8 0 -1
s NetOP_E 0 1 -1
y Buffer_E PassImg_E 1 0 -1
s PassImg_E 0.9    0 -1
y Buffer_E StoreProc_E 0 1 -1
s StoreProc_E 3.3 0 -1
y StoreProc_E DBWriteVid_E 1 0 -1
f StoreProc_E 1    0 -1
s DBWriteVid_E 7.2 0 -1
y DBWriteVid_E DkWriteVid_E 8 0 -1
s DkWriteVid_E 1 0 -1
y StoreProc_E BufRel_E 1 0 -1
s BufRel_E 0.5 0 -1
-1

```


Simple Web Application (Base Design)

```

G
""
1.0E-6
100
1
0.9
-1

P 0
p P1 f
p P2 f
p P3 f
p P4 f
p UserCPU f i
-1

T 0
t APP1 f E_APP1 -1 P2
t APP2 f E_APP2 -1 P2
t DB1 f E_DB1 -1 P3
t DB2 f E_DB2 -1 P4
t UserT r User_E -1 UserCPU m 60
t WS f E_WS -1 P1
-1

E 0
s E_APP1 1.25 0.0 -1
y E_APP1 E_DB1 1.0 0.0 -1
s E_APP2 1.25 0.0 -1
y E_APP2 E_DB2 1.0 0.0 -1
s E_DB1 2.0 0.0 -1

```

```

s E_DB2 0.25 0.0 -1
s E_WS 1.0 0.0 -1
y E_WS E_APP1 1.0 0.0 -1
y E_WS E_APP2 1.0 0.0 -1
s User_E 0.0 0.0 -1
Z User_E 0.0 15.0 -1
y User_E E_WS 0.0 1.0 -1
-1

```

Complex Web Service System for 300 cases (Base Case with variables)

```

$dem31=1:16,5
$req1=0.2:5,2
$req2=0.2:10,2
$req3=0.5:10,2

```

```

G
"three web applications"
0.0
0
0
0.9
-1

```

```

P 0
p App1P f m 4
p App2P f m 3
p App3P f m 2
p InfP f i
p Serv1P f m 8
p Serv2P f
p Serv3P f m 2

```

p WSP s m 4

-1

T 0

t App1 f App1_E1 App1_E2 -1 App1P m 10

t App2 f App2_E1 App2_E2 App2_E3 -1 App2P m 5

t App3 f App3_E1 -1 App3P

t Ops f Op1 Op2 Op3 -1 InfP i

t Serv1 f Serv1_E1 -1 Serv1P

t Serv2 f Serv2_E1 Serv2_E2 Serv2_E3 -1 Serv1P

t Serv4 f Serv4_E1 -1 Serv3P m 3

t Serv3 f Serv3_E1 TServ3_E2 -1 Serv2P

t User r users -1 InfP m 100

t WS f WS_E1 WS_E2 WS_E3 -1 WSP m 50

-1

E 0

s App1_E1 0.5 -1

f App1_E1 0 -1

y App1_E1 Serv1_E1 1.0 -1

s App1_E2 1.5 -1

f App1_E2 0 -1

y App1_E2 Serv2_E1 1.0 -1

s App2_E1 1.0 -1

f App2_E1 0 -1

y App2_E1 Serv2_E2 1.0 -1

y App2_E1 Serv3_E1 1.0 -1

s App2_E2 0.6 -1

f App2_E2 0 -1

y App2_E2 Serv2_E3 1.0 -1

s App2_E3 0.2 -1

f App2_E3 0 -1

y App2_E3 Serv3_E1 1.0 -1

s App3_E1 0.2 -1

f App3_E1 0 -1

```

y App3_E1 Serv2_E3 0.7 -1
y App3_E1 Serv4_E1 1.0 -1
y App3_E1 TServ3_E2 0.5 -1
s Op1 0.0 -1
y Op1 WS_E1 1.0 -1
s Op2 0.0 -1
f Op2 0 -1
y Op2 WS_E2 1.0 -1
s Op3 0.0 -1
f Op3 0 -1
y Op3 WS_E3 1.0 -1
s Serv1_E1 1.0 -1
f Serv1_E1 0 -1
s Serv2_E1 1.0 -1
f Serv2_E1 0 -1
s Serv2_E2 2.0 -1
f Serv2_E2 0 -1
s Serv2_E3 3.0 -1
f Serv2_E3 0 -1
s Serv3_E1 $dem31 -1
f Serv3_E1 0 -1
s Serv4_E1 1.0 -1
f Serv4_E1 0 -1
s TServ3_E2 1.0 -1
f TServ3_E2 0 -1
s WS_E1 3.0 -1
y WS_E1 App1_E1 2.0 -1
y WS_E1 App2_E1 $req1 -1
s WS_E2 5.0 -1
y WS_E2 App1_E2 1.0 -1
y WS_E2 App2_E2 $req2 -1
s WS_E3 2.0 -1
y WS_E3 App2_E3 $req3 -1
y WS_E3 App3_E1 0.5 -1
s users 0.0 -1

```

```

Z users 20.0 -1
y users Op1 0.4 -1
y users Op2 0.5 -1
y users Op3 0.6 -1
-1

```

Image Exchange System

```

G
"Image Exchange System"
0.0
0
0
0.9
-1

P 0
p AppP f
p CtrlP f
p InfP f i
p TransP f
-1

T 0
t AppServ f procImg -1 AppP
t Client r clients -1 InfP m 10
t CtrlM f startProc -1 CtrlP
t DB f getParm getImgIDs getRawImg -1 AppP
t Encrypt f enc -1 TransP
t FileSys f saveFile getFile -1 TransP
t Net f net_E -1 InfP i
t Printer f printRpt -1 InfP i
t PubServ f store -1 InfP
t TransServ f transmit -1 TransP
t WS f renderImg -1 AppP

```

-1

E 0

s clients 0.0 -1

Z clients 60000.0 -1

y clients startProc 1.0 -1

s getImgIDs 150.0 0.0 -1

s getParm 1.0 0.0 -1

s getRawImg 7.0 0.0 -1

s printRpt 30000.0 0.0 -1

s procImg 0.5 0.0 -1

y procImg getImgIDs 1.0 0.0 -1

y procImg getParm 1.0 0.0 -1

z procImg printRpt 1.0 0.0 -1

y procImg getRawImg 1.0 0.0 -1

y procImg renderImg 1.0 0.0 -1

y procImg saveFile 1.0 0.0 -1

s renderImg 500.0 0.0 -1

s saveFile 12.0 0.0 -1

s startProc 1.0 0.0 -1

y startProc procImg 4.0 0.0 -1

y startProc transmit 1.0 0.0 -1

s transmit 0.5 0.0 -1

s getFile 2.0 0.0 -1

y transmit getFile 400.0 0.0 -1

y transmit enc 400.0 0.0 -1

s enc 500.0 0.0 -1

y transmit net_E 400.0 0.0 -1

s net_E 0.0 0.0 -1

Z net_E 200.0 -1

y net_E store 1.0 0.0 -1

s store 6.0 0.0 -1

-1

APPENDIX C DIAGNOSIS LOG OF IMAGE EXCHANGE SYSTEM GENERATED BY PB

```

- ***** Start Model Diagnosis *****
- Workspace -
D:\data\home\research\development\testing\ImgTrans\ImgTrans
- -----Population Start-----
- Change log: base case
sfbn TransServ asserted
- Change note: task TransServ is set to multiplicity of 2
sfbn Encrypt asserted
- Change note: task Encrypt is set to multiplicity of 2
sfbn CtrlM asserted
- Change note: task CtrlM is set to multiplicity of 2
sfbn TransServ asserted
- Change note: task TransServ is set to multiplicity of 3
sfbn CtrlM asserted
- Change note: task CtrlM is set to multiplicity of 4
sfbn TransServ asserted
- Change note: task TransServ is set to multiplicity of 5
sfbn TransServ asserted
- -----Population End-----
- Only 1 option, proceed automatically: base case
sfbn TransServ asserted
structuralmitigation-sfbn is fired: handling TransServ
partition-sfbn is retracted
- Change log: Software bottleneck TransServ cannot be solved: try
shrink execution(self)
Shrink task TransServ

- Change log: Software bottleneck TransServ cannot be solved: try
shrink execution(servers)

```

Shrink target entires of task TransServ

- Change log: Software bottleneck TransServ cannot be solved: try
increase asynchronous

Increase asynchronous on target entries of task TransServ

- Change log: Software bottleneck TransServ cannot be solved: try
batching

: Batching on outgoing calls of task TransServ

- -----Population Start-----

- Change log: Software bottleneck TransServ cannot be solved: try
shrink execution(self)

Shrink task TransServ

sfbn TransServ asserted

- -----Population End-----

- -----Population Start-----

- Change log: Software bottleneck TransServ cannot be solved: try
shrink execution(servers)

Shrink target entires of task TransServ

sfbn TransServ asserted

- -----Population End-----

- -----Population Start-----

- Change log: Software bottleneck TransServ cannot be solved: try
increase asynchronous

Increase asynchronous on target entries of task TransServ

hwbn TransP asserted

- Change note: processor TransP is set to multiplicity of 2

hwbn TransP asserted

- -----Population End-----

- -----Population Start-----

- Change log: Software bottleneck TransServ cannot be solved: try batching

: Batching on outgoing calls of task TransServ

sfbn FileSys asserted

- Change note: task FileSys is set to multiplicity of 2

sfbn TransServ asserted

- -----Population End-----

- System chose option 2: Software bottleneck TransServ cannot be solved:
try increase asynchronous

Increase asynchronous on target entries of task TransServ

hwbn TransP asserted

structuralmitigation-hwbn is fired: handling TransP

task Encrypt moved

task TransServ moved

task FileSys tightened

task TransServ tightened

task Encrypt tightened

- Change log: Hardware bottleneck TransP cannot be solved: try shrink
execution(self)

Shrink task FileSys

Shrink task TransServ

Shrink task Encrypt

- Change log: Hardware bottleneck TransP cannot be solved: try
redeploy(min)

Move task TransServ from processor TransP to processor InfP

- Change log: Hardware bottleneck TransP cannot be solved: try
redeploy(max)

Move task Encrypt from processor TransP to processor InfP

- -----Population Start-----

- Change log: Hardware bottleneck TransP cannot be solved: try shrink
execution(self)

Shrink task FileSys

Shrink task TransServ

Shrink task Encrypt

hwbn TransP asserted

- -----Population End-----

- -----Population Start-----

- Change log: Hardware bottleneck TransP cannot be solved: try
redeploy(min)

Move task TransServ from processor TransP to processor InfP

hwbn TransP asserted

- -----Population End-----

- -----Population Start-----

- Change log: Hardware bottleneck TransP cannot be solved: try
redeploy(max)

Move task Encrypt from processor TransP to processor InfP

sfbn Encrypt asserted

- Change note: task Encrypt is set to multiplicity of 4

sfbn Encrypt asserted

- Change note: task Encrypt is set to multiplicity of 6

sfbn TransServ asserted

- -----Population End-----

- System chose option 2: Hardware bottleneck TransP cannot be solved:
try redeploy(max)

Move task Encrypt from processor TransP to processor InfP

sfbn TransServ asserted

structuralmitigation-sfbn is fired: handling TransServ

partition-sfbn is retracted

retract transform<Fact-19>

- Change log: Software bottleneck TransServ cannot be solved: try
shrink execution(self)
Shrink task TransServ

- Change log: Software bottleneck TransServ cannot be solved: try
shrink execution(servers)
Shrink target entires of task TransServ

- Change log: Software bottleneck TransServ cannot be solved: try
batching
: Batching on outgoing calls of task TransServ

- -----Population Start-----
- Change log: Software bottleneck TransServ cannot be solved: try
shrink execution(self)
Shrink task TransServ

sfbn TransServ asserted
- -----Population End-----
- -----Population Start-----
- Change log: Software bottleneck TransServ cannot be solved: try
shrink execution(servers)
Shrink target entires of task TransServ

sfbn TransServ asserted
- -----Population End-----
- -----Population Start-----
- Change log: Software bottleneck TransServ cannot be solved: try
batching
: Batching on outgoing calls of task TransServ

sfbn Encrypt asserted
- Change note: task Encrypt is set to multiplicity of 9
sfbn Encrypt asserted
- Change note: task Encrypt is set to multiplicity of 11

```

sfbn TransServ asserted
- -----Population End-----
- System chose option 2: Software bottleneck TransServ cannot be solved:
try batching
: Batching on outgoing calls of task TransServ

sfbn TransServ asserted
structuralmitigation-sfbn is fired: handling TransServ
retract transform<Fact-21>
partition-sfbn is retracted
retract transform<Fact-19>
- Change log: Software bottleneck TransServ cannot be solved: try
shrink execution(self)
Shrink task TransServ

- Change log: Software bottleneck TransServ cannot be solved: try
shrink execution(servers)
Shrink target entires of task TransServ

- -----Population Start-----
- Change log: Software bottleneck TransServ cannot be solved: try
shrink execution(self)
Shrink task TransServ

sfbn TransServ asserted
- -----Population End-----
- -----Population Start-----
- Change log: Software bottleneck TransServ cannot be solved: try
shrink execution(servers)
Shrink target entires of task TransServ

sfbn TransServ asserted
- -----Population End-----
- System chose option 0: Software bottleneck TransServ cannot be solved:
try shrink execution(self)

```

Shrink task TransServ

sfbn TransServ asserted

structuralmitigation-sfbn is fired: handling TransServ

retract transform<Fact-21>

partition-sfbn is retracted

retract transform<Fact-19>

retract transform<Fact-17>

- Change log: Software bottleneck TransServ cannot be solved: try
shrink execution(servers)

Shrink target entires of task TransServ

- -----Population Start-----

- Change log: Software bottleneck TransServ cannot be solved: try
shrink execution(servers)

Shrink target entires of task TransServ

sfbn TransServ asserted

- -----Population End-----

- Only 1 option, proceed automatically: Software bottleneck TransServ
cannot be solved: try shrink execution(servers)

Shrink target entires of task TransServ

sfbn TransServ asserted

structuralmitigation-sfbn is fired: handling TransServ

retract transform<Fact-21>

partition-sfbn is retracted

retract transform<Fact-19>

retract transform<Fact-18>

retract transform<Fact-17>

- -----

- No more design change options available

- =====

- Step 0: base case

Performance: [f,s]

Client 3.49789E-6 2858870.0

- Final results:

- Step 25: Software bottleneck TransServ cannot be solved: try shrink
execution(servers)

Shrink target entires of task TransServ

Performance: [f,s]

Client 5.09887E-5 196122.0

- ***** End Model Diagnosis *****