Handling Decisions and Traffic Dependencies in Layered Queueing Networks

by

Lianhua Li, B.A.Eng., M.Sc.

A thesis submitted to the Faculty of Graduate Studies and
Postdoctoral Affairs in partial fulfillment of the requirements
for the degree of

Doctor of Philosophy

in

Electrical and Computer Engineering

Carleton University

Ottawa, Ontario, Canada

©2016

Lianhua Li

Abstract

A Layered Queueing Network (LQN) is a recognized performance modelling technique for performance prediction and evaluation of distributed systems. However, at present LQNs do not handle models with state-based behaviour such as timeouts and aborts, called ‘decisions’ here. This research extends LQNs by incorporating decisions into the model. The XML input language used to describe LQNs has been extended to handle these decisions. Both the LQN simulator, `lqsim`, and analytic solver, `lqns`, were then modified to solve models with decisions.

The analytic solver uses decomposition and mean value analysis to solve models. Unfortunately, mean value analysis cannot be used to solve models with state-based behaviour. To overcome this limitation, a new approach called Dynamic Parameter Substitution (DPS) is used where intermediate results found while solving the model are used to alter the input parameters for subsequent iterations of the solution. To accomplish this goal, Layered Queueing eXperiment (LQX) language functions were derived to handle timeout and retry decisions and to handle fair-share queueing. The results from solving models using DPS were compared to results found from hybrid modelling, simulation, and where feasible, Petri nets.

This research also improves the accuracy of the LQN analytic solver when solving models with traffic dependencies, namely interlocking and sub-chain dependent behaviour. Interlocking occurs from the decomposition of the model into submodels; a single customer from an upper submodel may appear as two or more customers in lower level submodels. Some forms of interlocks were handled by the previous solutions, but with some limitations. Sub-chain dependencies are hidden more deeply in LQN models, and are handled for the first time. Sub-chain dependencies occur when traffic from multiple independent clients share a common intermediate server then diverges to lower level servers when this server is acting as a source of customers. This research presents a generalized solution to handle more general cases of inter-

locks and sub-chain dependencies. The results of the improved solution have good accuracy compared with simulation results and the problems of the extraneous delays and infeasible utilizations are eliminated.

Acknowledgements

I would like to dedicate this thesis to my parents who gave me endless love and encouragement.

I would like to express my sincere gratitude to my supervisor, Professor Greg Franks, for his support and advice throughout the years. His guidance is essential in success of this work.

I would like to thank my family and friends for their moral support. In particular, I would like to thank my husband, Park, and my son, Tom, for their support and encouragement.

I would like to thank Jennifer Poll for her support and timely information. I would also like to thank Naren Mehta, Jerry Buburuz, Kong Chiv, Danny Lemay and Daren Russ for their support and keeping everything in the lab running smoothly.

Contents

1	Introduction	1
1.1	Motivation	2
1.1.1	Making Decisions	3
1.1.1.1	Methodologies Comparison	3
1.1.1.2	Method of Dynamic Parameter Substitutions	6
1.1.1.3	LQN Decision Patterns	8
1.1.2	Traffic Dependencies	8
1.2	Contributions	10
1.2.1	Traffic Dependencies	10
1.2.2	Decision Making	11
1.2.3	Published Papers	12
1.3	Thesis Organization	13
2	Background	15
2.1	Model-Based Performance Analysis and Techniques	15
2.1.1	Model-Based Performance Analysis	16
2.1.2	Model-Driven Performance Engineering	16
2.2	Analytic Performance Modelling Techniques	18
2.2.1	Queueing Networks	18
2.2.2	Extended Queueing Networks	20

2.2.3	State-Based Solution Techniques	21
2.2.4	Stochastic Process Algebra	22
2.3	Mean Value Analysis	22
2.3.1	Exact MVA	23
2.3.2	Bard-Schweitzer AMVA	23
2.3.3	Linearizer AMVA	24
2.3.4	Variants of AMVA	26
2.4	Layered Queueing Networks	27
2.4.1	LQN Components	27
2.4.2	Layered Queueing Networks Solver: LQNS	30
2.4.3	PARASOL: LQN Simulation Engine	33
2.4.4	LQN Simulation Tool: LQSIM	35
2.4.4.1	LQSIM Tasks	35
2.4.4.2	Multi-Server Tasks	37
2.4.5	Layered Queueing Experiment Language: LQX	38
2.5	Hybrid Performance Modelling	40
2.5.1	Hybrid Simulation/Analytic Modelling	40
2.5.2	Multi-Formalism Hybrid Performance Modelling	41
2.5.3	Hybrid Modelling Related to LQN	42
2.6	Modelling Exception Handling Decisions in Resource Allocations . . .	43
2.6.1	Wu's HPMM for Handling Decisions	44
2.7	Summary	46
3	Interlocks	47
3.1	Interlocking Phenomenon	47
3.1.1	Problems in the Current Solver	50
3.1.1.1	Incorrect Distribution of Waiting Time	51
3.1.1.2	Interlock With External Contention	52

3.2	Improved Interlock Calculation	53
3.2.1	The Lower-Level Adjustment	54
3.2.1.1	Interlocking Probability	54
3.2.1.2	Interlocking Ratio and Interlocking Relationship . . .	57
3.2.1.3	The Lower-Level Adjustment	59
3.2.2	The Upper-Level Adjustment	61
3.2.2.1	Calculating Upper-Level Adjustment	63
3.2.2.2	Existence of Mixed Interlocked Flows	64
3.2.3	Testing the Improved Solution	65
3.2.3.1	Split Interlocks	66
3.2.3.2	Split Interlock with External Contention	67
3.2.3.3	External Contention to a Common Entry	68
3.3	Sending Interlocks	69
3.3.1	Types of Sending Interlocks	70
3.3.2	Improvements on Sending Interlocks	72
3.3.2.1	<i>Type1</i> and <i>Type2</i> Sending Interlocks	72
3.3.2.2	Sending Interlocks with External Contentions	74
3.3.2.3	Type3 Sending Interlocks	75
3.4	Interlocking Effects on Second-Phase Servers	79
3.4.1	Second-Phase Technique	79
3.4.1.1	Performance Advantages of Second-Phase Technique	79
3.4.2	Interlocking Effects	81
3.4.2.1	Multi-Phase Clients	82
3.4.2.2	Multi-Phase Interlocked Server	83
3.4.2.3	Multi-Phase Intermediate Server	84
3.4.3	Improvements	86
3.4.3.1	Multi-Phase Interlocked Server	87

3.4.3.2	Multi-Phase Intermediate Servers	88
3.4.4	Results	91
3.4.4.1	Multi-Phase Interlocked Server	91
3.4.4.2	Multi-Phase Intermediate Server	95
3.4.4.3	Randomized Model with Second-Phase Services	95
3.5	Verification	97
3.5.1	The Combined Model	99
3.5.2	Deeply Layered Interlocking Model	100
3.6	Summary	101
4	Sub-Chain Dependencies	103
4.1	Sub-Chain Dependencies	103
4.1.1	Impacts of Sub-Chain Dependencies	104
4.2	Improved Chain Calculation	108
4.2.1	The Maximum Number of Customers	108
4.2.2	Applying MC to Approximate MVA Solutions	110
4.2.2.1	Applying MC to Bard-Schweitzer AMVA Solutions	112
4.2.2.2	Linearizer AMVA Solution	116
4.2.3	Verification	117
4.3	Sub-Chain Dependency in Parallel Executions	122
4.3.1	LQN Activities and Their Precedences	123
4.3.2	Improvements and Results	125
4.4	Summary	126
5	Case Study: An Online Bookstore	128
5.1	RADSBookStore	128
5.1.1	Traffic Dependencies in RADSbookstore	129
5.1.1.1	Sub-chain dependency	129

5.1.1.2	Interlocks	131
5.1.2	Evaluation	132
5.2	Conclusions	133
6	Modelling Decisions in LQN	135
6.1	Decisions in LQN	135
6.2	LQN Decision Patterns	138
6.2.1	Queue Pattern	138
6.2.2	Abort Decision Pattern	139
6.2.3	Infinite-Retry Decision Pattern	140
6.2.4	Finite-Retry Decision Pattern	141
6.2.5	Timeout Decision Pattern	142
6.2.6	LQN Decision Schema	143
6.2.6.1	Pattern Template in LQN Input Model	145
6.3	Decisions in LQSIM	146
6.3.1	Timeout/Abort Decisions	147
6.3.2	Retry Decisions	149
6.3.3	LQSIM Results Verification	153
6.3.3.1	Verification Through GreatSPN	153
6.3.3.2	Verification Through CSIM	159
6.4	Dynamic Parameter Substitutions Using LQX	164
6.4.1	Method of Dynamic Parameter Substitutions	164
6.4.2	Abort Decision Pattern	165
6.4.3	Timeout Decision Pattern	168
6.4.3.1	Algorithm 1	168
6.4.3.2	Algorithm 2	172
6.4.3.3	Discussions	173
6.4.4	Infinite-Retry Decision Pattern	175

6.4.5	DPS Demonstration	178
6.4.5.1	Abort Pattern	179
6.4.5.2	Infinite-retry Pattern	179
6.4.5.3	Timeout Pattern	182
6.5	Summary	183
7	Analytic Solution of Fair Share Scheduling	185
7.1	Fair Share Scheduling	185
7.1.1	Group Shares	187
7.2	Dynamic Parameter Substitution Using LQX	188
7.2.1	FS Processor Transformation	189
7.2.2	Mappings to Dynamic Parameters	190
7.2.3	Redistributing Surplus Shares	193
7.2.4	Verification LQX-DPS	196
7.2.4.1	Case 1: Cap Shares	196
7.2.4.2	Case 2: Guarantee Shares Without Spare Resources .	197
7.2.4.3	Case 3: With Surplus Resources	198
7.3	Integrating DPS into LQNS	199
7.3.1	Mapping Dynamic Parameters in LQNS	199
7.3.2	The Implementation in LQNS	200
7.3.2.1	FairShareSubmodel	201
7.3.2.2	FairShareServer	204
7.3.2.3	Convergence Control	205
7.3.3	Verification of the LQNS-DPS Approach	206
7.3.3.1	Resolving the Previous Model	206
7.3.3.2	Randomized Model	208
7.4	Summary	211

8	Conclusions	213
8.1	Handling Traffic Dependencies in LQN	213
8.1.1	Interlocking	214
8.1.2	Sub-chain Dependencies	216
8.1.3	Case study	217
8.2	Handling Decisions in LQN	217
8.2.1	LQN Decision Patterns	217
8.2.2	Handle Decisions in the LQN Simulator	218
8.2.3	Applying DPS Method to Handle Decisions	219
8.2.4	Using DPS to handle FS Scheduling	219
8.3	Remarks	221
8.4	Limitations and Future Work	222
8.4.1	Limitations	222
8.4.2	Future work	223
A	Randomized Model	256

List of Figures

2.1	An example of LQN model notations.	28
2.2	A two-tier client-server model (with results).	31
2.3	The decomposed submodels of the model in Figure 2.2.	31
2.4	Corresponding QNs to the submodels in Figure 2.3.	32
2.5	PARASOL Meta Model. (From [132])	34
2.6	PARASOL Task States (Reduced). (From [132].)	35
2.7	Multi-server task transformation in LQSIM.	37
3.1	Interlocking effects on a client-server model.	48
3.2	A model that combines all forms of interlocks.	49
3.3	Comparing the existing interlock correction and simulation	52
3.4	An interlock with an external contention.	53
3.5	An invalid case of interlocks	55
3.6	Comparing the new interlock correction with simulation	66
3.7	Model with external contention to a common entry.	68
3.8	A LQN model of a sending interlock.	70
3.9	Results from the previous solution for the Figure 3.8 model	71
3.10	Results with the new solution for the Figure 3.8 model.	73
3.11	A LQN model of a sending interlock with an external contention. . .	74
3.12	Results from the previous solutions for the Figure 3.11 model.	76

3.13	Results of the new solution for the model in Figure 3.11.	77
3.14	A LQN model of a type3 sending interlock.	78
3.15	Second-phase server events.	80
3.16	A three-tiered LQN model with a two-phase interlocked server.	81
3.17	Results by the previous solution for the multi-phase client model.	82
3.18	Results by the previous solution for the Figure 3.16 model.	83
3.19	The previous solution for the multi-phase intermediate tasks model.	85
3.20	second-phase clients and interlocked servers.	90
3.21	second-phase clients and interlocked servers.	90
3.22	Results for the model in Figure 3.16 (Part 1).	92
3.23	Results for the model in Figure 3.16 (Part2: one customer).	93
3.24	Results for the model in Figure 3.16 (Part2:two customers).	94
3.25	Results of the multi-phase intermediate server model (Part 1).	95
3.26	Results of the multi-phase intermediate server model (Part 2)	96
3.27	A simplified LQN model with second-phase client and servers.	97
3.28	Results comparison for the model in Figure 3.27	98
3.29	A Complex Model with Interlocking Effects.	100
4.1	A LQN model with a sub-chain dependency.	104
4.2	Submodels from Figure 4.1.	105
4.3	Results from solving the model in Figure 4.1.	106
4.4	An example model for MC calculations.	111
4.5	Two types of sub-chain dependent submodels.	114
4.6	New results from solving the model in Figure 4.1 (varying c_1).	118
4.7	New results from solving the model in Figure 4.1 (varying t_0 and c_1).	120
4.8	An LQN activity model with a sub-chain dependency.	122
4.9	Precedence Schema of Activities. From LQN user manual [77]	123

5.1	An online bookstore model From [105].	130
5.2	The submodel of server InventoryMgr (the 4 th layer).	131
6.1	A decision scenario of an exception handling in resource allocations. .	136
6.2	LQN model for a decision scenario.	137
6.3	Abort decision scenario.	139
6.4	Infinite-retry decision scenario.	140
6.5	Finite-retry decision scenario.	141
6.6	Timeout decision scenario.	143
6.7	Top level schema of LQN models.	144
6.8	Schema of the LQN Decision.	145
6.9	Translated model of Timeout_Decision in LQSIM	147
6.10	Message movements of a Timeout_Decision task.	148
6.11	Message movements in a Retry_Decision task.	150
6.12	A LQN models for a finite-retry decision.	151
6.13	A web server LQN model with an abort decision	154
6.14	The Petri net model of an abort pattern.	154
6.15	Results from LQSIM and GreatSPN of a simple abort pattern.	155
6.16	The Petri net model of a timeout pattern.	155
6.17	Results from LQSIM and GreatSPN of a timeout pattern.	156
6.18	A Petri net model of an infinite-retry decision.	157
6.19	Results from LQSIM and GreatSPN for an infinite-retry decision . . .	157
6.20	The Petri net model of a finite-retry decision.	158
6.21	Results from LQSIM and GreatSPN of a finite-retry decision.	159
6.22	LQN models with decisions.	161
6.23	Results from LQSIM and CSIM (abort decision).	162
6.24	Results from LQSIM and CSIM (infinite-retry)	162
6.25	Results from LQSIM and CSIM (finite-retry).	163

6.26	A LQN model for an abort decision pattern.	166
6.27	Results of DPS, LQSIM and GreatSPN of an abort pattern.	168
6.28	LQN model for the timeout decision pattern.	169
6.29	Another LQN model for LQN timeout decisions.	172
6.30	A webserver system with timeout.	175
6.31	Results from the two DPS algorithms for a timeout decision.	176
6.32	Results from DPS, LQSIM and GreatSPN for an infinite-retry pattern.	178
6.33	Results of the model with the abort decision.	180
6.34	Results of the model with the infinite retries decision.	181
6.35	Results of the model with the timeout decision.	183
7.1	The model Transformation of a FS processor.	189
7.2	A test model with 3 tasks in two groups.	196
7.3	Results of shares redistribution.	199
7.4	Results from LQNS-DPS and LQSIM.	207
7.5	A randomly generated LQN model.	209

List of Tables

3.1	Interlocking Probability for the model in Figure 3.1a.	57
3.2	Results of the Figure 3.1a model with randomized parameters.	67
3.3	Results of the Figure 3.4a model with randomized parameters.	67
3.4	Results of the Figure 3.7 model with randomized parameters.	69
3.5	Results of the Figure 3.8 model with randomized parameters.	74
3.6	Results of the Figure 3.11 model with randomized parameters.	77
3.7	Results of the Figure 3.14 model.	79
3.8	Results from solving 100 different versions of the model in Figure 3.29.	99
3.9	Results of the Figure 3.2 model.	100
3.10	Results of the Figure 3.29 model.	101
3.11	The execution time of two models in this section.	101
4.1	<i>MCs</i> of all client and server entries in the model in Figure 4.4.	112
4.2	Results for the Figure 4.1 model with randomized parameters.	119
4.3	Results for the randomized model with a finite number threads.	121
4.4	The execution time of two randomized models.	122
4.5	Results comparison for the model in Figure 4.8.	126
5.1	The error rates of the queueing delays at server InventoryMgr	132
5.2	The error rates of service time	133
5.3	The error rates of throughput	133

5.4	RMS error compared to simulation of the RADSbookstore model. . .	133
6.1	The performance parameters used in DPS algorithms.	165
7.1	Case 1: Group utilization.	197
7.2	Case1: Relative errors (%) compared to simulation.	197
7.3	Case2: Group utilization.	198
7.4	The execution time of the LQNS-DPS and simulations.	208
7.5	Results for the random model with ($CS_{Group1} = 0.6$, $CS_{Group2} = 0.4$).	210
7.6	Results for the random model with ($CS_{Group1} = 0.2$, $CS_{Group2} = 0.8$).	210
7.7	Results for the random model with ($GS_{Group1} = 0.2$, $CS_{Group2} = 0.8$).	211
7.8	The execution time of the LQNS-DPS and simulations.	211

Acronyms

AMVA Approximate Mean Value Analysis. 19, 213

BVT Borrowed Virtual Time. 183

CFS Completely Fair Share. 28, 33, 183

CIM Computational Independent Model. 17

CoMoM Class-oriented Method of Moments. 20

CORBA Common Object Request Broker Architecture. 27

CPN Coloured Petri Nets. 21, 41

CTMC Continuous-time Markov chains. 21

DPS Dynamic Parameter Substitutions. 6, 132, 216

EQN Extended Queueing Networks. 20, 30

FCFS First Come First Serve. 28, 217

FIFO First In First Out. 34

FS Fair Share Scheduling. 28, 182, 216

GFL General Form Linearizer. 26

- GSPN** Generalized Stochastic Petri Nets. 4, 5, 21, 41, 42, 44, 45, 151
- HIT** Hierarchical Evaluation Tool. 20
- HOL** Head of Line without preemption. 28
- HPMM** Hybrid Performance Modeling Methodology. 42, 44, 45, 216
- HSA** Hybrid Simulation/Analytic. 40–43
- IMSE** Integrated Modelling Support Environment. 42
- JMT** Java Modelling Tools. 20
- LIFO** Last In First Out. 34
- LQN** Layered Queueing Networks. 2, 41, 44, 45
- LQNS** LQN Analytic Solver. 2, 42, 43, 45, 217
- LQSIM** LQN simulation solver. 5, 34
- LQX** Layered Queueing eXperiment Language. 6, 7, 38, 132, 161, 216, 217
- MARTE** Modelling and Analysis of Real-Time and Embedded Systems. 17
- MDA** Model-driven Architecture. 17
- MDE** Model-Driven Engineering. 2, 15, 17, 46
- MDPE** Model-Driven Performance Engineering. 1, 16, 17, 46
- MFHP** Multi-Formalism Hybrid Performance. 41
- MOL** Method of Layers. 20
- MoM** Method of Moments. 19

MVA Mean Value Analysis. 6, 7, 9, 19, 22, 39, 50, 210, 217

MVAC Mean Value Analysis by Chain. 19

PCM Palladio Component Model. 17

PEPA Performance Evaluation Process Algebra. 22

PIM Platform Independent Model. 17

PN Petri Net. 4, 21

PR Priority with preemption. 28

PS Processor Sharing. 28, 217

PSM Platform Specific Model. 17

QN Queueing Networks. 18, 20, 41

QPN Queueing Petri Net. 17, 21

QSA Queue Shift Approximation. 26

RECAL Recursion by Chain Algorithm. 19

RPC remote procedure calls. 2, 21, 43

SFS Surplus Fair Share. 183

SHARPE Symbolic Hierarchical Automated Reliability and Performance Evaluator.
20, 41

SMART Simulation and Markovian Analyzer for Reliability and Timing. 41

SPA Stochastic Process Algebra. 22

SPE Software Performance Engineering. 1, 15

SPN Stochastic Petri Net. 41

SPNP Petri Net Solver. 42, 45

SPT Schedulability, Performance, and Time Specification. 17

SRVN Stochastic Rendezvous Networks. 20, 27

TDA SRVN by Task-Directed Aggregation. 20

UCM Use Case Maps. 44

UML Unified Modeling Language. 17, 132

Chapter 1

Introduction

A major cause for the failure of many large distributed systems projects today is a lack of performance [53]. Predicting system behaviour in the early phase of system development before systems are built can reduce the risk of failure because it is too costly to repair or redesign the system late in the development cycle. Early performance predictions can detect the design flaws, locate potential bottlenecks which may seriously affect the system performance, and determine proper system deployment and system configuration.

Software Performance Engineering (SPE) [195, 196, 197], which is the pioneer of the model-based performance analysis [11, 56, 172] and Model-Driven Performance Engineering (MDPE) [82, 115], applies performance models into software systems at very early stages of software developments. Using performance models, SPE is able to plan the system capacities, locate the potential bottlenecks in the system, and predict system performance. Furthermore, with these performance models, performance analysis and evaluation can be done at any stage of the development process, the time required for the analysis is small, and it is relatively low cost compared to other methods [109, 195]. Consequently, the risks of improper design or failures to deliver the system can be significantly reduced. Research related to *SPE* developed

various performance modelling methodologies (analytic or simulation) or formalisms (queueing-based networks, state-based networks) to study system behaviours and improve system performance.

Performance models and their underlying solution techniques also are the fundamentals of model-based performance evaluations [53, 123, 172] and model-driven engineering. Model-Driven Engineering (MDE) [71, 104, 117, 199] has become lately an important way to develop software systems, predict and evaluate system performance, and study performance related issues.

Layered Queueing Networks (LQN) [74, 81, 183, 219, 220], are a well recognized and powerful method [223] in performance evaluation of many distributed systems [63, 113, 140, 141, 161, 170, 175, 191, 209, 226]. It is particularly well suited to model systems using blocking remote procedure calls (RPC) [26], which are common in distributed systems. LQN models are also used to predict and evaluate many web services [41, 75, 148, 192, 206] and optimize and manage resource provisions in the clouds [8, 131].

Performance modelling is the subject of this research. This research was focused on improving the accuracy of the LQN underlying analytic solution in some particular cases and enhancing the LQN modelling power to handle state-based behaviour.

1.1 Motivation

In general, the LQN analytic solution (LQNS) is able to produce quite accurate performance evaluation results. However, LQNS may produce less accurate results in some special cases with the occurrence of traffic dependencies. Also, LQN models have limitations to model state-based behaviours, for instance, making decisions in handling exceptions [91]. This work was motivated by the problems in the existing LQN analytic solution and the limitation of LQN models, and efforts have been made

to improve the accuracy of the LQN analytic solution with the presence of traffic dependencies and to enhance the modelling power of LQN models to handle state-based behaviours.

1.1.1 Making Decisions

Handling exceptions is an important form of making decisions. Decisions are made depending on dynamic execution behaviour and performance results. For example, in a web server system with facilities to handle timeout exceptions, client requests time out if the processing time exceeds the timeout threshold. A timeout exception is raised to handle the release the resources that a timeout request is occupying before it leaves the system. The threshold of the timeout is an important requirement for the system design and capacity planning. For the designer, two questions may be asked:

1. What the capacity of the system is if the probability of timeout requests is less than 5%, for example?
2. What is the probability of timeout requests, if the system runs under some system configurations?

The first question can be easily answered by a queueing-based formalism. However, queueing-based formalisms face difficulty in answering the second one. Since the decision of whether a timeout exception being raised depends on the current system state at that moment.

1.1.1.1 Methodologies Comparison

Making decisions at some points inside a network is a weakness of queueing-based formalisms. Therefore, queueing models are only suitable for modelling exceptions when

the probabilities of exceptions being raised are known before the model is solved. However, exceptions are often raised because of the results that the performance model produces. For example, excessive response times can lead to timeouts and excessive utilization can lead to lock request failures. This is the major limitation of queueing-based formalisms being used to model systems that have state-based behaviours or dynamic execution behaviours. LQN, as an extension of queueing networks, also faces the same difficulties in modelling systems with state-based behaviours.

Therefore, systems with state-based behaviours have to be modelled by state-based methods, for example, Petri nets (PN) [151, 165, 166] and their variants. However, one well-known problem of state-based methods is their scalability. Even a moderately sized system may have infeasible computation costs, and become unsolvable.

Some attempts have been made to use queueing-based formalisms to model dynamic system behaviour. For example, Zhang [228], used the results provided by previously solved LQN models and fed them to some kind of controller, which could make decisions based on this feedback to generate a new set of system parameters. These decisions can be used to adjust the system architecture [228] in order to realize self-adaptive system architectures [227].

In order to evaluate and predict the systems with state-based behaviours, a hybrid approach defined by Shanthikumar [190] can be used. In this method, a queueing model is used to solve quickly most of the overall model, and a state-based or a simulation approach is used to model and solve the part with a state-based behaviour. For example, Wu [224] developed a hybrid approach, which combines LQN models with Generalized Stochastic Petri Nets (GSPN) [1, 142] models to handle exceptions in resource allocation activities.

Hybrid approaches are being increasingly looked at to model and evaluate the performance of those systems with state-based behaviours. However, combining multiple methods together inevitably increases the complexity of the overall solution.

First, regarding model composition, the system under question has to be expressed in different modelling languages, one corresponding to each of the formalisms. Second, as for model recognition, the entire model or any submodels in different formats may need to exchange information with each other. This can be achieved through an intermediate modelling language, which can be understood by all the involved methods. However, the model translations and transformations required by a hybrid approach may have to use some existing tools which are external to the hybrid solution. The computation cost is another concern of hybrid approaches because multiple solution techniques (solvers) have to be invoked in every iteration. The results produced by each of the solvers are often exchanged by files when switching iteratively among the solution spaces. Using files to exchange data means that file operations happen repeatedly in every solution space and when switching between solution spaces in every iteration.

Another reason for this research abandoning hybrid solutions is that hybrid approaches are not suitable for LQN models. This fact was shown by the author's previous work [133], in which a hybrid solution combining LQNS and LQSIM (the LQN simulation tool) was developed to solve LQN models with processors using fair share scheduling faster than simulation. This hybrid solution works properly only for the models without traffic dependencies. The problems related to traffic dependencies lead to the second motivation for this research.

Wu [224] used GSPN to solve a reduced model which emphasizes the decisions to be made, and to find the probabilities of branches of each decision. It employed LQN to solve the entire model by importing results of submodels found by the Petri nets solver. However, his hybrid solution failed to escape from the problems of accuracy and scalability because of the reasons mentioned above.

1.1.1.2 Method of Dynamic Parameter Substitutions

Making a decision is a result of the consideration of a current system state. Most performance results derived from various performance models are mean values or probabilities. From these observations, the probability of an exception being raised can be found through some performance parameters in the form of mean values, if their relationships are interpreted correctly. This builds the foundation of the method of Dynamic Parameter Substitution (DPS). The probabilities of raising exceptions are referred to as dynamic parameters because they are related to dynamic execution behaviour. DPS maps dynamic parameters to other performance results which can be found by Mean Value Analysis (MVA) [181].

Dynamic Parameter Substitutions (DPS) was developed in this research to evaluate decisions in handling exceptions in LQN models. The method of DPS was inspired by setting up multi-run experiments of LQN models using the Layered Queueing eXperiment language (LQX) [150]. LQX is a programming language with a syntax similar to C, and was developed to design multi-factor experiments and to control the output of execution results of experiments as specified by the user. In fact, the multi-factor input parameters specified by LQX are not limited to a set of user predefined values. Therefore, the results of performance parameters (in mean values) from the previous execution can also be transformed into a new set of input parameters for the next execution. This means that if dynamic parameters are defined as multi-factor input parameters, they can be changed based on the results from the previous execution during the LQX execution. The substitution is achieved by mapping functions between dynamic parameters and specified performance parameters. Dynamic parameters reach their final value after the evolution of LQX iterations. In summary, DPS is an iterative method to solve state-based behaviours by associating them with dynamic parameters, then mapping these dynamic parameters to other performance parameters (in the form of mean values) through predefined substitution

functions.

The advantage of using DPS to handle state-based behaviours is that the modelling power of LQN is greatly enhanced without suffering the scalability problem. The probabilities of taking a normal execution path and raising an exception are calculated via MVA [181], rather than from state probabilities. Consequently, scalable and fast solutions become significant advantages of this research. Being able to handle decisions equips LQN models to provide modelling facilities to evaluate the system behaviours during exception handling in network communication protocols or passive resource allocations. It is also able to evaluate the impact of the fair share scheduling policy to the entire system since the fair share scheduling is being increasingly used in many systems with the requirement of ‘quality of service’ (*QoS*).

However, using DPS to handle state-based behaviours has disadvantages. Since the relationships between dynamic parameters and their associated performance results vary upon decision types, the processes of revealing the relationships and establishing those mappings have to be worked out individually for each type of decisions. This drawback also leads to concerns of the generality and the coverage of the decisions handled by DPS.

Fortunately, these drawbacks can be eliminated by using pre-defined decision patterns. In this work, decisions with predefined mappings are defined into LQN decision patterns. These predefined mappings are developed as substitution functions in LQX and are invoked through each corresponding pattern. Since the decisions defined in this work are based on those scenarios commonly used in many distributed systems, it means that the coverage of LQN decisions is able to model the systems using these decision patterns. The coverage will be increased as more decisions patterns are defined and more dynamic parameter substitution functions are discovered. Furthermore, the defined patterns and their substitution functions may be reused and extended when defining new decision patterns to extend LQN models to handle more complex

behaviours in the future.

1.1.1.3 LQN Decision Patterns

The decision patterns defined in this research are based on Wu’s patterns in [224] with an important addition. Wu identified four patterns of handling exceptions in resource allocations: *queue*, *abort*, *finite-retries* and *infinite-retries* in [224]. The *queue* pattern is essential “no exception”, so the requests wait until allocations succeed. The *abort* pattern ends callers request prematurely. The *infinite-retries* keeps trying until the resource request is successful. Finally, the *finite-retries* pattern makes a fixed number of tries before giving up. In addition to these four patterns, *Timeout*, as an important constraint in the specifications of communication protocols or a performance requirement of many systems, is defined as the fifth LQN decision.

This research generalizes Wu’s patterns to handle more general cases of exceptions. It shows that abort decisions can be treated as a special case of timeout decisions when the timeout limit is zero. In addition, finite-retries and infinite-retries can be generalized with timeout features.

The LQN decision patterns not only hide the implementation of substitutions from modellers, but also provide templates (used in input files of LQN models) for modellers to ensure easy model constructions. With templates of LQN decision patterns, modellers only need to vary the values of input parameters within a template to specify their own decisions. These templates simplify the modelling process in terms of model compositions and relieve the burden from modellers.

1.1.2 Traffic Dependencies

Traffic dependencies arise when solving a LQN model analytically by subdividing the original model into a set of submodels, then solving each submodel independently of each other. There are two types of dependencies, interlocks [72, 220] and sub-chain

dependencies described below.

The reason that traffic dependencies affect the LQN analytic solution is that LQN uses decomposition as the fundamental mechanism to generate layers and creates a submodel corresponding to each layer. Each submodel is solved individually using MVA and client chains in each submodel are considered to be independent of each other. However, this assumption is not true in many systems, because of traffic dependencies between layers. Therefore, when models are solved using the method of decomposition, interlocking and sub-chain dependencies become inevitable. Not only does ignoring traffic dependencies seriously affect the accuracy of the LQN analytic solution to certain models, but also, the existence of traffic dependencies in LQN models makes hybrid solutions yield inaccurate results, which was shown in [133].

Interlocks, which are artifacts of the analytic solution using multiple submodels, arise because some requests originating from the same client at an upper layer, arrive at a lower layer server through different paths. By not accounting for interlocks, too much delay is introduced into the queueing network, resulting in overestimated residence times and underestimated throughput. The effect is particularly noticeable with a small number of interlocked clients [72].

Interlocking is recognized as a phenomenon which must be considered, and in some cases had been handled previously in Woodside et al.'s solver [220], Rolia and Sevcik's solver [183], and Franks' solver [81]. The current LQN analytic solver, LQNS [74, 76], handles more general cases of interlocks. However, it fails to handle some newly identified cases of interlocking, as well as sub-chain dependencies.

As opposed to interlocks, a sub-chain dependent client at a lower layer appears when customers from some independent sources at some upper layer submodels arrive and are blended together at a common server. If this server is acting as a source of customers to other servers, it becomes a sub-chain dependent client. Without considering the sub-chain dependent problem, large errors occur especially when the

numbers of customers from the involved routing chains are imbalanced.

The research focuses on the newly identified interlocking cases in LQN models to improve the LQNS solver to handle all the interlocking cases properly. These improvements include correcting new interlocking cases and improving the accuracy of the corrections to the previously handled cases. The sub-chain dependencies are corrected by distinguishing the maximum possible number of customers arrived from the sources at an upper layer.

1.2 Contributions

Contributions are divided into two broad categories, one focused on accuracy and the other on enhancing modelling capabilities.

1.2.1 Traffic Dependencies

The first part of this research is handling traffic dependencies to improve the overall accuracy of the LQNS solver. The contributions are:

1. A general solution to all identified forms of interlocks was developed. The new solution consists of a two-level adjustment: the upper-level adjustment is active at the upper layer, where the common client is, and the lower-level one at the layer where the interlocked server is.
2. The upper-level adjustment removes the excessive queueing delays caused by included service demands to the common client at the upper-level submodel. Without upper-level adjustments, the current LQN solver distributes the queueing delays at both levels of submodels incorrectly. Missing upper-level adjustments is also one of the reasons that causes low accuracy of hybrid approaches related to LQN.

3. The work evaluated interlocking effects at a lower level, through *interlocking probability*, *interlocking ratio* and *interlocking relationship*. The lower level adjustment was refined by using these variables to improve the accuracy of corrections to interlocking effects.
4. Sub-chain dependencies were fixed by correcting the maximum possible number of arriving customers from a routing chain as well as each of its sub-chains. Controlling the correct number of arriving customers also has the effect of reducing the impacts of some forms of interlocks; therefore, this correction was integrated into LQNS, along with the corrections to interlocks to form an improved general solution of LQNS to handle traffic dependencies.

1.2.2 Decision Making

The second part of this research focuses on handling the decisions in exception handling (within the resource allocation context) in LQN models. The contributions are as follows:

1. The modelling capability of the LQN technique has been enhanced by incorporating decision patterns into LQN models using the DPS approach. These patterns allow LQN models to model and evaluate the systems with decisions in a simplified and efficient manner without suffering the problem of scalability.
2. ‘Timeout’, a commonly used decision, was defined as the fifth LQN decision pattern, in addition to the four decision patterns of Wu. The templates corresponding to these decision patterns were defined to simplify model compositions for modellers.
3. The LQN simulation tool, LQSIM, was extended to simulate all the LQN decision patterns first. The LQSIM results were verified by a Petri net solver,

GreatSPN [7, 47, 48] and another simulation tool, *CSIM*. Therefore, *LQSIM* is able to provide accurate and comparable results to evaluate the accuracy of those analytic results found by *LQNS* at any size of *LQN* models.

4. The mappings between dynamic parameters and certain performance results (in the form of mean values) were created for each decision pattern. The dynamic parameter substitution functions corresponding to those mappings were implemented through *LQX* functions.
5. The *LQN* analytic solver, *LQNS*, has been built to handle more state-based behaviours by providing the mechanism for defining and handling dynamic parameters analytically. As a result, the processors (or other resources) with a fair share (*FS*) scheduling [116] policy can be handled analytically by *LQNS*, while handling the fair share scheduling was previously implemented only in the *LQN* simulator (*LQSIM*) in [132].

1.2.3 Published Papers

The following is a list of papers which are published by the author during the research.

1. Lianhua Li and Greg Franks. Analytic Solution of Fair Share Scheduling in Layered Queueing Networks. In 13th European Workshop on Performance Engineering (EPEW 2016), (Accepted). Chios, Greece, October 5-7, 2016.
2. Lianhua Li and Greg Franks. Modeling Decisions in Layered Queueing Networks. In 48th Summer Computer Simulation Conference 2016 (SCSC 2016), Montreal, Quebec, Canada, July 24-27, 2016.
3. Lianhua Li and Greg Franks. Improved Chain Calculation for Sub-chain Dependencies in Layered Queueing Networks. In 15th IEEE International Conference

on Scalable Computing and Communications (ScalCom 2015), Beijing, China, August 10-13, 2015.

4. Lianhua Li and Greg Franks. Improved interlock correction when solving layered queueing networks using decomposition. In IEEE 28th Canadian Conference on Electrical and Computer Engineering (CCECE 2015), Halifax, NS, Canada, May 2015.
5. Lianhua Li and Greg Franks. Hybrid performance modeling using layered queueing networks. In CMG-2012, volume 1, pages 85–93, Las Vegas, Nevada, USA, December 3–7, 2012. Computer Measurement Group.
6. Greg Franks and Lianhua Li. Efficiency improvements for solving layered queueing networks. In Proc. 3rd WOSP/SIPEW International Conference on Performance Engineering (ICPE 2012), pages 279–282, Boston, MA, USA, April 22–25, 2012.
7. Lianhua Li and Greg Franks. Performance modeling of systems using fair share scheduling with layered queueing networks. In Proceedings of the 17th IEEE/ACM International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunications Systems (MASCOTS 2009), pages 1–10, London, September 21–23, 2009.

1.3 Thesis Organization

The rest of this thesis is organized as follows:

1. Chapter 2 provides background information related to the research.
2. Chapter 3 describes the improved analytic solution to handle interlocks and shows that their effects are greatly reduced.

3. Chapter 4 discusses the developed method to overcome the sub-chain dependent problem.
4. Chapter 5 presents an online bookstore application system, which has both interlocks and sub-chain dependencies, to evaluate the effectiveness of the improved solution.
5. Chapter 6 explains how LQN models are extended to handle decisions through LQN decision patterns in both the analytic solution through LQX and the LQN simulator LQSIM.
6. Chapter 7 describes how to use the DPS approach to analytically evaluate processors with a fair share scheduling [116] policy in LQNS and through LQX.
7. Chapter 8 summarizes the research and also describes future research directions.

Chapter 2

Background

This section provides the necessary background knowledge which is related to this research. First, the concepts and techniques of model-based performance analysis and model-driven performance engineering are described. Then, the major analytic performance techniques and Mean Value Analysis are explained. Next, LQN models and their solution techniques are explained, which are the subject of this research. After that, a description of hybrid modelling techniques follows. Finally, related works for modelling decisions in the context of exception handling are described.

2.1 Model-Based Performance Analysis and Techniques

Model-based performance analysis [11, 56] is a successor of Software Performance Engineering, SPE [195, 196, 197]. It applies performance modelling into any stage of software system design to predict system performance, locate potential bottlenecks and identify possible designing flaws.

More recently, as Model-driven engineering (MDE) [71, 104, 117, 199] becomes an important way to develop software systems, model-driven performance engineering

(MDPE) [19, 82, 115, 172] has emerged, which combines model-based performance analysis and model-driven techniques.

2.1.1 Model-Based Performance Analysis

Model-based performance evaluation methodologies are grouped into three categories: analytic, simulation and hybrid modelling. Simulation is able to model more complex or some specific system behaviour. However, execution time and computational resource consumption are concerns that can limit its use when evaluating large-scale systems. On the other hand, many analytic solutions surveyed in [11, 94] can solve models in fast and efficient ways. However, they have to use assumptions or approximations, so accuracy is a big concern for these methods. Consequently, models with more complicated system behaviour do not have precise analytic solutions today.

Hybrid performance modelling techniques, as an alternative choice, have drawn the attention of many researchers in recent decades. Hybrid modelling techniques aim to model systems with more complex behaviour. More precisely, Hybrid Simulation/Analytic models bridge the gaps between analytic solutions and simulations, while multi-formalism hybrid techniques take advantages of multiple analytic formalisms to model those systems which a single technique alone cannot or has difficulties to solve. Analytic and hybrid performance modelling methodologies are described in Sections 2.2, 2.3 and 2.5.

2.1.2 Model-Driven Performance Engineering

Model-driven performance engineering uses model-driven techniques to transform software architecture models into performance analysis models [115]. Therefore, the core activities of model-driven techniques are model transformations [59, 86, 169]. Those transformed performance models are solved by model-based performance techniques. The major model-driven techniques are:

1. From software architects to performance models [222]. Software architects are represented by the UML diagrams with requirement specifications (e.g. performance annotations), such as the UML profiles: SPT (Schedulability, Performance, and Time Specification) [157], and MARTE (Modelling and Analysis of Real-Time and Embedded Systems) [158]. The performance models can be generated from UML diagrams are: queueing models in [12, 57], LQN models by UML2LQN [93, 173], performance evaluation process algebra [101] (*PEPA*) by [205] or Petri nets by [5, 23, 24, 193], and their variants, for example, coloured Petri nets (from UML activity diagrams) in [200].
2. Palladio Component Model (PCM) [20, 21]. PCM is an architecture modelling language to describe Component-based Software Architecture (CBS), which is a successor of Object-Oriented Software Architecture. PCM models can be automatically transformed into various performance models, such as queueing models, Petri net models by [65], LQN models by PCM2LQN [124], and queueing Petri net models (QPN) [17, 18] by PCM2QPN [144].
3. Model-Driven Architecture (MDA) [119]. MDA is a framework based on UML to generate three types of meta-models: computational independent models (CIM), platform independent models (PIM) and platform specific models (PSM) [147]. In MDA, a CIM model, a conceptual model which specifies the business context and requirements of a system under design, is first transformed into a PIM model, and then the PIM model generates a PSM model through model transformation techniques [86, 169].

All the activities of MDE and MDPE eventually evaluate systems through model-based performance analysis. These performance models, which can be analytic, simulation, or hybrid models, and their solution techniques are fundamental building blocks of all the performance evaluations.

Therefore, LQN performance modelling and analysis is the subject of this research, more precisely, this research is focused on enhancing the LQN performance models and improving their solution techniques.

2.2 Analytic Performance Modelling Techniques

In the model-based performance tools [11] that exist today, analytic performance techniques [94] can mostly be classified into three categories: queueing-based, state-based, and process algebra techniques.

2.2.1 Queueing Networks

In computer systems, jobs or processes compete against each other with a limited number of resources (hardware or software). Queueing Networks (QN) [10, 129] are a natural choice to model this behaviour. In a queueing network, resources are modelled as server stations, while jobs or processes are clients or customers, and clients are connected to stations by routing chains. A queueing network (with a single customer class) is either open, in which customers arrive from outside the system, then wait and receive service, and then leave the network; or closed, in which no customers come from outside (all customers stay in the network forever), and when after receiving all services, customers spend some time to think then repeat the cycle again.

A product-form solution for queueing networks was first discovered by Jackson [107] for single client open class networks. It states that the distribution of a network state can be represented by a product of the distributions of each of the individual nodes [85]. Later, the product-form solution was extended to closed queueing networks with exponential servers by Gordon and Newell [92], and to the *BCMP* networks, by Baskett, Chandy, Muntz and Palacios [16]. Beyond BCMP networks, production-form solutions were also extended to some other QNs, such as

G-networks [70, 83, 84], which allow positive and negative customers in the networks and QNs with batch arrivals and batch services [95, 96].

Convolution [33, 127, 179, 180] and Mean Value Analysis (MVA) [181] are two computationally efficient algorithms for product-form solutions. The key part of the convolution algorithm is the calculation of normalization constants, which are used to scale all the state probabilities. However, convolution algorithms have numerical instability problems. Similarly, RECAL [55] also calculates normalization constants but uses a recursive algorithm based on chains. Casale [36] developed an exact algorithm for computing normalizing constants of closed multi-class product-form networks, by solving a linear system of equations.

Mean Value Analysis is based on the relationship between the mean waiting time and the mean queue length with one customer less, as stated by the Arrival Theorem [189]. MVA is able to find the mean values of performance metrics without calculating state probabilities. MVA has been a primary solution for production-form queueing networks. The Approximate MVA (AMVA) solutions, Bard-Schweitzer [15, 186] and Linearizer [43] AMVAs are two major iterative AMVA solutions, which are described in more detail in the next section. MVA also has some other implementations, for example, a recursive implementation based on chains, *Mean Value Analysis by Chain* (MVAC) [54].

More recently, Casale introduced the Method of Moments (MoM) [38], which is an exact algorithm for solving closed queueing networks with multi-class and a large population. MoM recursively solves higher-order moments of queue-lengths by a linear system of equations instead of mean values. These equations are those used to calculate normalization constants in RECAL [55] and LBANC [44]. MoM only requires log-quadratic time and log-linear space in the total population size [38], whose computation costs are less than MVA, since MVA computation complexity grows combinatorially as the product of class populations. Apart from MoM, Casale

also proposed a new algorithm called CoMoM [37], which can efficiently compute normalization constants for the closed queueing networks with a large number of classes. However, Method of Moments may potentially be affected by numerical problems. MoM was also generalized by Casale in [39] to overcome the inefficiency problem of MoM when the number of queues and classes grow simultaneously.

Tools for solving queueing networks are *QNAF* [208] (discontinued), an integrated toolset Java Modelling Tools (JMT) [25, 40], and a library package *qnetworks* [143]. Solutions to queueing networks were also integrated into some performance modelling and analysis tools or frameworks, such as the hierarchical evaluation tool HIT [22] and the Symbolic Hierarchical Automated Reliability and Performance Evaluator (SHARPE) [185].

2.2.2 Extended Queueing Networks

Normal QNs have difficulties in handling synchronization, simultaneous resource possession, and splitting of customers, because these situations do not have product-form solutions, so some extended queueing networks (EQN) have been developed. Simultaneous resource possession has arisen in many distributed systems and is handled by the Method of Surrogates [108], Stochastic Rendezvous Networks (SRVN) [219, 220], SRVN by Task-Directed Aggregation (TDA) [171, 174], the Method of Layers (MOL) [183, 184], Layered Queueing Networks (*LQN*) [74, 81], Ramesh's decomposition [177]. The common technique used by these EQNs (excluding Flow-Equivalent QNs), is that they all decomposed the entire network into a set of submodels. However, they used different layering strategies, which were compared in [76].

Among all these EQN solutions, Layered Queueing Network (*LQN*) [74, 76, 81], (which combines *SRVN* [171, 174, 219, 220] and *MOL* [183, 184]), is widely used to solve models of distributed systems with layered architectures, for example, client-server systems [63, 140, 161, 170, 191, 209]. *LQN* is a popular choice because it is

particularly well suited to model systems using RPC [26].

2.2.3 State-Based Solution Techniques

As an important State-based formalism, Petri nets (PN) [90, 151, 165, 166], is a mathematical modelling language, which was invented by Carl Adam Petri [166]. Petri nets and its variants are suitable to model synchronization and concurrency in distributed systems. Petri nets use places, transitions and arcs to construct networks and represent systems. Places may contain a discrete number of indistinguishable tokens; tokens move from one place to some other places when a transition is fired. The distributions of tokens represent system states.

Petri nets have many variants to extend their modelling power. In a Coloured Petri net (CPN) [110, 111, 112], tokens are distinguishable in order to carry information. A stochastic Petri net (SPN) [149] allows that a PN has exponentially distributed transition rates in order to represent the duration of activities. A generalized stochastic Petri net (GSPN) [1, 142] can have transitions with stochastic or deterministic timing characteristics.

Petri nets are solved by numerical solutions to their underlying continuous-time Markov chains (CTMC). This can result in state space explosion, even in some moderately sized systems, thus limiting their use. Some tools used to solve stochastic Petri nets are *GreatSPN* [7, 47, 48], *SPNP* [51, 103, 207], *DSPNexpress* [138] and its later version *TimeNET* [229, 230], and *PIPE2* [31, 64].

Queueing Petri Net (QPN) [17, 18] is another formalism which combines a queueing network and coloured generalized stochastic Petri nets. However, QPN is not a hybrid modelling approach because it is an independent and formally defined formalism. Since QPN combines the expressiveness of queueing network and stochastic Petri nets, QPN can be used in performance evaluations of distributed systems with simultaneous resource possession, blocking behaviour, and asynchronous processing [122].

QPN is able to describe some complex scheduling behaviour of some server queues. However, QPNs, as an extension of PNs, also potentially has the state explosion problem when solving large sized systems. The QPN simulation solver (*SimQPN*) [120] was firstly developed to solve QPN models. More recently, the QPN modelling environment (*QPME*) [121, 198] was developed, which integrated the QPN graphical Editor (*QPE*) and the Simulator (*SimQPN*).

2.2.4 Stochastic Process Algebra

Process Algebra (*PA*) [9] is a formal language to model concurrent systems concurrency and dynamic system behaviour and prove the structure and behaviour of systems [101]. PA has two primary constructs: actions and processes, to build system specifications, then complex system behaviour can be combined through operations. The compositional language provides benefits for model solution as well as model construction. A system can be decomposed into a set of subsystems, which can be replaced by a collection of small parts that have equivalent behaviours.

Stochastic Process Algebra (SPA) [52, 100] and Performance Evaluation Process Algebra (PEPA) [101] were introduced by Hillston in 1996. PEPA extends classical process algebra by using random variables to represent the duration of actions [102]. PEPA is a formal description language for compositional performance modelling using stochastic process (Markov process) to describe the underlying SPA. Tools support PEPA are PEPA Workbench [89] and TIPPTool [98].

2.3 Mean Value Analysis

Mean Value Analysis (MVA) is based on the Arrival Theorem [189], which states that when the N^{th} customer from a chain k arrives at a server m , it sees $N-1$ customers already there. Mean Value Analysis of a closed network can be solved by an iterative

solution of equations (2.1)~(2.3). The mean residence time $W_{m,k}(\mathbf{N})$ for a customer from a chain k at a server station m is found by using (2.1). The throughput $X_k(\mathbf{N})$ of the system for chain k is calculated by using (2.2). Finally, the queue length $L_{m,k}(\mathbf{N})$ at the station m for class k (with population $\mathbf{N}$) is obtained by (2.3). MVA can be solved exactly or approximately depending on the calculation of $L_{m,k}(\mathbf{N} - \mathbf{e}_j)$ in (2.1), which is the queue length of the chain k at the server m with one customer removed from a chain j .

$$W_{m,k}(\mathbf{N}) = \begin{cases} S_{m,k} & ; \text{ Delay servers} \\ S_{m,k} + S_{m,k}L_{m,k}(\mathbf{N} - \mathbf{e}_j) & ; \text{ Queueing servers} \end{cases} \quad (2.1)$$

$$X_k(\mathbf{N}) = \frac{N_k}{\sum_{m=1}^M W_{m,k}(\mathbf{N})} \quad (2.2)$$

$$L_{m,k}(\mathbf{N}) = X_k(\mathbf{N}) \times W_{m,k}(\mathbf{N}) \quad (2.3)$$

2.3.1 Exact MVA

Exact MVA [178, 181] recursively calculates each queue length $L_{m,k}(\mathbf{N})$ at the full population $\mathbf{N}$, starting from zero customers, which means equations (2.1)~(2.3) have to be calculated for all of the possible populations. Therefore, memory and computation complexity are the two concerns of Exact MVA, and when it is used to solve networks with large populations, the computational costs may be prohibitively high.

Linearizer [43] and Bard-Schweitzer [15, 186] approximate MVAs are two widely used *AMVA* techniques and both have substantially reduced computation costs.

2.3.2 Bard-Schweitzer AMVA

Bard-Schweitzer AMVA [15, 186], also referred to as proportional estimation (*PE*), iteratively solves a queueing network at the full customer population $\mathbf{N}$. In Bard-

Schweitzer AMVA, it is assumed that $L_{m,k}(\mathbf{N} - e_j)$ is proportional to $L_{m,k}(\mathbf{N})$. Therefore, the queue lengths $L_{m,k}(\mathbf{N} - e_j)$ at station m for each chain k with one customer removed from chain j , is estimated using (2.4).

$$L_{m,k}(\mathbf{N} - e_j) = \begin{cases} L_{m,k}(\mathbf{N}) & ; k \neq j \\ \frac{N_j-1}{N_j} L_{m,k}(\mathbf{N}) & ; k = j \end{cases} \quad (2.4)$$

The estimated $L_{m,k}(\mathbf{N} - e_j)$ is used to calculate the mean residence time $W_{m,k}(\mathbf{N})$, throughput $X_k(\mathbf{N})$ and the mean queue length at the full population $L_{m,k}(\mathbf{N})$ in the next iteration. Compared to Exact MVA, Bard-Schweitzer AMVA is far more efficient in space and time, because Bard-Schweitzer AMVA only calculates the queue lengths at populations $\mathbf{N}$ and $\mathbf{N}-1$. However, Bard-Schweitzer AMVA often has large errors because the estimated $L_{m,k}(\mathbf{N} - e_j)$ is not a simple ratio of the number of customers in chain j . Bard-Schweitzer AMVA was extended to multi-server stations by Suri et al. in [202]. Bard-Schweitzer AMVA is implemented by the algorithm called *SchweitzerCore* in the LQN analytic solver LQNS, shown in Listing 2.1, in which all the $D_{mkj}(\mathbf{N})$ at line 9 are zeros. The convergence test is based on the difference of the queue lengths between any two successive iterations, shown in (2.5) from [43].

$$\max_{m \in M, k \in K} \frac{|L_{m,k}^I(\mathbf{N}) - L_{m,k}^{I-1}(\mathbf{N})|}{N_k} \leq \frac{1}{4000 + 16|\mathbf{N}|} \quad (2.5)$$

2.3.3 Linearizer AMVA

Linearizer [43] AMVA improves the accuracy of the Bard-Schweitzer approximation by applying a scaling factor $D_{m,k,j}$ to the estimation of the $L_{m,k}(\mathbf{N} - e_j)$ rather than using the simple ratio of $\frac{N_j-1}{N_j}$. The scaling factor $D_{m,k,j}(\mathbf{N})$ is a value used to adjust the proportion that one customer contributes to the length of a queue, and is

Listing 2.1: SchweitzerCore(N)

```

1  bool convergence = false;
2  while ( !convergence ){
3      /* Estimate L(m,k) */
4      for( m = 1; m <= M; m ++){
5          for( k = 1; k <= K; k ++){
6              F[n, m, k] = L[n, m, k] / N[k];
7              for( j = 1; j <= K; j ++){
8                  nk = ( j==k )? (N[k] - 1) : N[k] ;
9                  L[nej, m, k] = nk * (F[n, m, k] + D[n, m, k, j]);
10             }
11         }
12     }
13     /* One-step MVA */
14     step(N);
15
16     convergence = termination_test();
17 }

```

calculated by (2.6).

$$\begin{aligned}
 F_{m,k}(\mathbf{N}) &= \frac{L_{m,k}(\mathbf{N})}{N_k} \\
 D_{m,k,j}(\mathbf{N}) &= F_{m,k}(\mathbf{N} - e_j) - F_{m,k}(\mathbf{N}) \\
 L_{m,k}(\mathbf{N} - e_j) &= (\mathbf{N} - e_j)_k (F_{m,k}(\mathbf{N}) + D_{m,k,j}(\mathbf{N}))
 \end{aligned} \tag{2.6}$$

In order to find $D_{m,k,j}(\mathbf{N})$, Linearizer has to calculate all the queue lengths at both the full customer population $\mathbf{N}$, and $\mathbf{N} - e_j$ (i.e., one customer removed from chain j) for all client chains, to calculate $F_{m,k}(\mathbf{N})$ and $F_{m,k}(\mathbf{N} - e_j)$ in (2.6). At each of the population, Linearizer uses the Bard-Schweitzer approximation to solve the queue lengths for all client chains. Therefore, the Bard-Schweitzer AMVA is a special case of Linearizer AMVA, with $D_{m,k,j}(\mathbf{N}) = 0$.

Linearizer AMVA is implemented by executing *Schweitzer Core* algorithm multiple times.

1. *Schweitzer Core* is executed using the full client populations $\mathbf{N}$.

2. *Schweitzer Core* is executed K times (K is the number of client chains). Each of these executions is under a population $\mathbf{N} - e_j$ with one customer removed from each client chain j ($1 \leq j \leq K$).
3. The scaling factors $D_{m,k,j}(\mathbf{N})$ are calculated, and *Schweitzer Core* is run again with the full population ($\mathbf{N}$) after the scaling factors $D_{m,k,j}$ are applied to the estimation of all the queue length $L_{m,k}(\mathbf{N} - e_j)$.

2.3.4 Variants of AMVA

More recently, some variants of approximate MVA algorithms were developed. Wang and Sevcik [214] developed two AMVA algorithms called the queue line (*QL*) and the fraction line (*FL*) algorithms, which are based on Bard-Schweitzer AMVA. The FL algorithm can produce results with a higher accuracy compared with Bard-Schweitzer AMVA in the experiments on multi-class networks with a cost of less efficiency.

As for the variants of Linearizer, E Silva and Muntz developed a fast AMVA called *Improved Linearizer* [67] which is faster than the original Linearizer [43] by a factor of C (the number of client classes). Later, Schweitzer et al. developed another AMVA algorithm called Queue Shift Approximation (QSA) [187]. The number of levels of the QSA determines the level of accuracy of the QSA algorithm. Two-level QSA requires the calculation of queue length at the full population $\mathbf{N}$ and the population with one less customer $\mathbf{N}-\mathbf{1}$, while the three-level QSA also need to calculate the population with two fewer customers. Two-level QSA is less accurate than the original Linearizer. Three-level QSA has a higher accuracy than Linearizer but at a higher computation cost, since it needs to calculate all of the queue lengths at the population of $\mathbf{N}-\mathbf{2}$.

More recently, Wang et al. developed the General Form Linearizer (GFL) [215] algorithm. Linearizer and QSA are considered to be two special cases of GFL. The two-level GFL algorithm was proved to have the same numerical properties and computational complexities as the *Improved Linearizer* [67] algorithm. Their experiments

showed that these two-level GFL algorithms are generally faster and accurate than the *Improved Linearizer*.

2.4 Layered Queueing Networks

Layered Queueing Networks have evolved from *active servers* [221], Stochastic Rendezvous Networks (SRVN) [171, 174, 219, 220], Layered Queueing Network (*LQN*) [81], to the Enhanced LQN (*ELQN*) [74, 76]. (This entire research is based on Franks's ELQN analytic solution. ELQN is referred as to LQN in this thesis.) LQN is particularly suitable for solving performance models of distributed systems with a client-server architecture, in which remote procedure calls (*RPC*) [26], are typical inter-process communications. With this type of software systems, clients are blocked as they wait for replies to requests. Furthermore, some of the servers may make further requests to other servers in order to complete the requests they have received. LQN models have been used successfully in performance evaluations of distributed databases [175, 191], CORBA (Common Object Request Broker Architecture) [88, 163], middleware [140, 170, 209], systems with distributed replicated servers [161, 162], web services [41, 75, 148, 192, 206], and resource provision in clouds [8, 131].

The modelling artifacts of LQN models, the LQN analytic solver (*LQNS*), the LQN simulator (*LQSIM*) and its simulation engine *PARASOL* [153, 154], and the LQN eXperiment language (*LQX*) [150] are briefly introduced in the following sections.

2.4.1 LQN Components

Major elements in the LQN models to construct systems are processors, groups, tasks, entries, requests, phases and activities. A LQN model is graphically represented by

a directed graph consisting of the processors, tasks, entries and requests [77]. An example of a LQN model is shown in Figure 2.1.

Figure 2.1: An example of LQN model notations.

Processors are pure servers that only accept requests, and are used to model hardware devices, such as CPUs or disks, on which tasks or processes consume time. Processors are represented by circles, for example, processors *client* and *p1* in Figure 2.1. The scheduling policies of processors recognized by LQN are First Come First Serve (FCFS), Processor Sharing (PS), Priority with preemption (PR), Head of Line without preemption (HOL), and Fair Share (FS) scheduling [116]. As for FS scheduling, only Completely Fair Share Scheduling (CFS) [164] was implemented in the LQN simulator, LQSIM, and the *PARASOL* simulation engine in the author's previous work [132]. The analytic solution of FS scheduling in the LQN analytic solver is described later in this thesis.

Groups (not shown in Figure 2.1) are used to model FS scheduling behaviour of

processors. All the tasks run on a processor with a FS scheduling policy, are divided into groups. Each group has a share, and the ‘share’ for a group is distributed evenly among the group’s tasks.

Tasks can be both clients and servers. LQN tasks are used to model software modules, software resources (threads, buffers, or locks) and even some hardware operations. Tasks are represented by large parallelograms, for example, tasks *d1* and *d2* in Figure 2.1. Multi-server tasks represent homogeneous threads that allow parallel execution. Server tasks or processors with multiple copies are referred to as multi-servers and are represented by stacked parallelograms or circles, for instance, the infinite processor *client* and the server task *t1* (in the center of the figure). The multiplicity of a reference tasks (task *client1* in the figure) represents the number of customers in the model.

Entries are used to model the different services provided by a task, and are represented by small parallelograms within a task parallelogram, for instance, entry *d1* and *e2* in Figure 2.1. Entries perform the actual services provided by a task and consume time and resources. For a processor, server entries are referred as device entries and are used to consume time for the tasks running on the processor.

LQN processors, tasks and entries are connected by requests, which are shown using directed arcs in a network graph. Requests can be:

1. synchronous, which is represent by a solid arc with a solid arrowhead;
2. asynchronous [177], which is represented by a solid arc with an empty arrowhead;
3. forwarding, which is represented by a dashed arc with a solid arrowhead.

All the requests in Figure 2.1 are synchronous. Synchronous requests, like remote procedure calls, block senders until replies arrive while asynchronous requests do not

block senders. Forwarded calls do not block the intermediate tasks because replies are sent to the original sender.

Activities [78] are used to model internal concurrency with behaviour of fork-join, or internal loops within a task. The sequence or the order of the execution of activities is modelled by ‘precedence’ in LQN.

In addition to the explicit parallelism through multi-threading and fork-joins, early replies are another way to provide concurrency. A second phase [155] is a mechanism that allows early replies to senders and is supported by some distributed operation systems. Phase-one (PH1) is a synchronous phase used to perform the services in which senders are blocked. Phase-two (PH2) is an asynchronous phase and can be used to perform some clean-up operations without the sender’s participation after the reply is sent. Early replies reduce the blocking time of senders and allow parallel execution of a sender and a receiver. Early replies can also be used to model software artifacts, like buffers [80]. LQN allows three phases at maximum on clients and servers [79, 80].

LQN also supports distributed replicas [74, 161], and the quorum scheme of replications [159, 160].

2.4.2 Layered Queueing Networks Solver: LQNS

The Layered Queueing Networks Solver (*LQNS*) [74, 76, 81] is an analytic solution to LQN models. It uses hierarchical decomposition and the method of surrogates [108] to solve LQN models. A LQN model is solved by splitting the model up into a set of submodels, then solving each of these models using Mean Value Analysis [181].

For example, Figure 2.2 is a LQN model of a simplified client-server system; Figures 2.3a and 2.3b show the decomposed submodels for the LQN model in Figure 2.2 using LQNS. Other EQN solvers [114, 145, 183, 220] construct submodels differently based on the layering strategies; however the solution approaches for submodels are

Figure 2.2: A two-tier client-server model (with results).

Figure 2.3: The decomposed submodels of the model in Figure 2.2.

the same. The comparisons of these layering strategies were presented in [76, 81].

Each submodel has a corresponding queueing model. Figure 2.4 shows the two queueing network models corresponding to the two submodels in Figure 2.3. In each submodel, client tasks make requests to server tasks or processors, so clients tasks become *chains* while server tasks and processors are *stations* in the corresponding queueing model. Any of the server tasks in the original model as a server station only appears once among the queueing models, while it may be included in several queueing models as a client chain. The number of customers in each chain is determined by the multiplicity of the corresponding client task, with the exception caused by the existence of sub-chain dependencies described later in Chapter 4.

Figure 2.4: Corresponding QNs to the submodels in Figure 2.3.

The residence times found from solving the queueing model in Figure 2.4b, which is the lower submodel, is used as a surrogate delay [108] in the queueing model in Figure 2.4a, which is the upper layer submodel. Similarly, the utilization of servers in Figure 2.4a is used to find the think times in Figure 2.4b. The two submodels are iteratively solved until the entire model converges.

Listing 2.2 shows the steps of LQNS to solve a LQN model. This algorithm iterates over each of the submodels until it converges on a solution. Different MVA solutions can be used in LQNS when solving each submodel (in Line 8 in Listing 2.2), such as Exact MVA, Bard-Schweitzer AMVA, or Linearizer AMVA and its variants. The algorithm terminates when the root of the sum of mean differences in utilization ($\sqrt{\sum \Delta U}$) of all the entities in the model is less than a user-defined value, or the pre-defined iteration limit is reached. This approach generalizes to any number of layers. LQNS scales very well regarding both the number of nodes in the network and the number of customers that can visit these nodes. Furthermore, the approximate Mean Value Analysis solutions in LQNS can be very robust even when many of the underlying assumptions of product-form queueing networks are violated.

On the other hand, LQN models can also be solved using a variety of approaches

Listing 2.2: LQNS Algorithm.

```

1  begin
2      Read input file and create DOM model;
3      Create LQNS model;
4      Generate layers;
5      Create submodels for each layer;
6      do while (not convergence):
7          for each submodel s:
8              Solve submodel s using an MVA;
9              Save waiting times for submodel s;
10             Set think times for submodel s+1;
11         Save results to DOM and write to output file;
12 end

```

including simulation, Markov chains and fluid approximations [204], and process algebras [203].

2.4.3 PARASOL: LQN Simulation Engine

PARASOL [153, 154] is a discrete event-driven simulation [13, 14] engine. It is efficient in simulating distributed and parallel systems. It provides basic building blocks of hardware components to construct hardware systems or network environments. These basic components are *nodes*, *hosts*, *tasks* and *groups* and the communication facilities like *buses*, *links* and *ports*, shown in Figure 2.5.

A *node* (or processing node) may contain one or more processors, which are referred to as hosts. Nodes are used to simulate single core or multi core processors. The communications between hosts within the same node are through two-direction buses, while links are one-direction communication channels to connect two nodes.

Similar to the LQN groups, Parasol groups are also dedicated to simulate fair share scheduling. A CFS processor [132] has two ready-to-run-queues, one for ready tasks, and one for groups. The groups with most remaining shares stay in front of the group ready-to-run-queue. Each group has several tasks, which are evenly sharing the group share.

Tasks are processes running on a host. Any process (kernel or non-kernel), or user-defined tasks can be modelled by a Parasol task.

Inter-task communications are all performed through ports. *Ports* are places to queue messages to be received by tasks. Every task has a `std_port` to receive messages, and a `reply_port` to receive replies to the requests sent by this task. Both ports are created when the task is created. Some tasks may own some additional ports, which are either private ports, or shared ports shared by multiple tasks. For example, a multi-server task has a `ready_port` to check the availability of its workers. A message is sent to a port through a system call `ps_send()`, and is accepted from a port through another system call `ps_receive()`. Messages queued at a port, may be received in a FIFO, LIFO or random order. Messages are often sent to a destination port directly in a one-hop manner. However, some messages may be re-sent to another destination port after it is received. Sending a message in this multiple-hop manner is used to simulate some complex behaviour of tasks. For example, in Section 6.3, multi-hop messages are used to simulate decisions in the simulator, LQSIM.

Figure 2.5: PARASOL Meta Model. (From [132])

A Parasol task may be in one of six states: ready, computing, blocking, sleeping, suspended (not in the figure), and *hot* states, shown in Figure 2.6 from [73]. A task enters the computing state, (consuming time on a *host*) through `ps_compute()`, and

Figure 2.6: PARASOL Task States (Reduced). (From [132].)

enters the sleeping state through `ps_sleep()`. The *hot* state is a special running state which user-defined simulation code is running (which would invoke `ps_compute()` and `ps_sleep()`). Transitions between states are caused by Parasol system calls or the occurrence of certain events.

2.4.4 LQN Simulation Tool: LQSIM

The LQN simulation tool, LQSIM is designed for simulating LQN models using PARASOL simulation engine. Parasol nodes, groups and task objects are created for all of the processors, groups and tasks defined in LQN input models. The multiplicity of a LQN processor determines the number of hosts for the corresponding Parasol node.

2.4.4.1 LQSIM Tasks

LQN tasks are the primary modelling artifacts of LQSIM. Listing 2.3 from [132] is a typical service routine of a single server task, and it also shows the basic template of implementing LQN tasks through PARASOL.

The code shown in Listing 2.3 is a typical service routine of a single server task. The main loop of a single server task begins with receiving a message on its `std_port`. The message contains the information of the service entry and the reply port of the sender (`reply_to`). Line 5 is used to spend time on its own service. After the

Listing 2.3: Main task loop. (From [132])

```

1  while (true){
2      message msg_in = receive(std_port);
3      bool not_done = true;
4      while (not_done) {
5          compute(msg_in.entry);
6          entry dst = choose_dst(msg_in.entry);
7          if (dst != nil) {
8              message msg_out;
9              msg_out.entry = dst;
10             msg_out.reply_to = reply_port;
11             send(dst.std_port, msg_out);
12             receive(reply_port);
13         }else{
14             not_done = false;
15         }
16     }
17     send(msg_in.reply_to, nil);
18 }

```

completion of its own service, the task either makes further requests to another task through lines 8~11 or replies to the sender through line 17. If a further request is required, the task is blocked by waiting for the reply at line 12.

The other LQSIM tasks are:

1. **Reference Tasks** are the client tasks of LQN models (at the top layer), which generate customers into the network. Reference tasks use the same task template in Listing 2.3, except omitting lines 2 and line 17, because they do not accept requests from other tasks. The think times of customers are simulated by using `ps_sleep()`.
2. **Multi-servers** are not supported by PARASOL directly. Rather they are composed of a queue task acting as a dispatcher and a number of worker tasks. Multi-servers are described in the next section.
3. **Infinite servers** are similar to multi-servers except that worker tasks are created as needed without limit. Therefore, worker tasks are always available to

the dispatcher (the queue task), so requests never queue at the dispatcher.

4. **Worker tasks** are server tasks but owned by a multi-server task, infinite-server task or a semaphore task. Workers for a multi-server use a similar template code, shown in Listing 2.3, with one addition line:

```
send(worker_port).
```

This line is inserted before line 2 to inform the dispatcher of the multi-server task that the worker task is available and ready to accept requests.

5. **Semaphore and Signal tasks** were implemented in LQSIM in [73] to model passive resources. Semaphores and signal tasks are similar to multi-server tasks.

2.4.4.2 Multi-Server Tasks

In LQSIM, a multi-server task comprises a queue task as an intermediate *dispatcher* task and a number of worker tasks, is shown in Figure 2.7. Worker tasks are used to perform the actual services and reply to the original sender directly. The number of workers is determined by the multiplicity of the multi-server task.

Figure 2.7: Multi-server task transformation in LQSIM.

The dispatcher is used to accept requests from clients, and then forwards these requests to any available worker, which is shown in Figure 2.7b. If no worker is available, the requests are queued at the dispatcher. The dispatcher task runs on its

Listing 2.4: Queue task loop.

```

1  while (true){
2      message msg_in = receive(std_port);
3      message worker = receive(worker_port);
4      send(worker.std_port, msg_in)
5  }
```

own node so that it never blocks waiting for a host. It has two ports, one is `std_port` for accepting incoming requests, and one is `worker_port` for receiving the messages from its workers, which indicate which workers are available. The service loop of the dispatcher is shown in Listing 2.4. The dispatcher must receive messages on both ports at line 2 and line 3 in order to proceed to forward the request from the client to a free worker. Otherwise, if no worker is available, the dispatcher queues up the client’s request and waits for available workers; or if no request is available, it waits for a client request and queues up free workers.

2.4.5 Layered Queueing Experiment Language: LQX

The Layered Queueing eXperiment Language (LQX) [150] was developed to run multiple factor experiments. LQX is written as a part of a LQN input model file in the XML format. The changeable input values are defined by using `$`-variables in model input files; these variables are set using a LQX program. `$`-variables may be applied to multiple variables in an input model, and one variable can be used to update multiple parameters simultaneously. The LQX function `solve()` is used to solve the model and can be invoked explicitly after all of the `$`-variables are defined, or is implicitly invoked if `solve()` is omitted in an LQX program.

LQX is a high-level language with syntax similar to C. LQX supports many of language constructs of C, standard mathematics functions, random variable generation function, and file handlers for I/O operations. LQX has four data types: numeric, boolean, string, and object; user-defined types are not supported in LQX. LQX also

is dynamically typed language with a strong type checking (all mixed-type expressions have to be type-casted explicitly). More details can be found in the LQN user manual [77].

In LQX, the *object* type is used to access the results of LQN objects defined in an input model. Objects are processors, tasks, activities, entries, phases, and calls. These objects are *read-only* and are accessed by calling LQX built-in functions, through this *dot* operator to access result attributes such as throughput and utilization.

When combined with LQX, LQNS uses four levels of iterations to solve a LQN model:

1. LQX calls `solve()`.
2. `Solve()` calls Linearizer over all queueing submodels until convergence, which is shown in Listing 2.2 on Page 33.
3. Linearizer calls Bard-Schweitzer at populations $\mathbf{N}$ and with one customer removed from each chain k , $\mathbf{N} - \mathbf{1}_k$, for a submodel.
4. Bard-Schweitzer calls `core()` to find the residence time at the servers until convergence. Response times ($W_{m,k}$) at each server m are found by using the function `wait(m)`.

The first two levels of iterations are referred as to **LQX iterations** and **submodel iterations** respectively. The last two levels of iterations together are referred as to **MVA iterations**. These notions of iterations are used in Section 7.3 to specify the total number of iterations of the DPS approaches.

In [76], all of the inner iterations (*Step 3* and *Step 4* above) were removed leaving only the outer iteration to execute in the MVA iterations. For fixed rate and infinite servers, this approach produced the same results as the more expensive technique. The LQX iteration is the outermost iteration, which is used to perform input parameter substitutions in order to run multiple experiments. It is also the place where the

DPS substitutions take place (discussed in Section 6.4 and 7.3). However, in LQX, there is no requirement to solve a model to completion before changing parameters, which means the computation costs may be reduced even further.

2.5 Hybrid Performance Modelling

Hybrid performance modelling combines two or more solution techniques as an integrated unit to perform evaluation activities. Hybrid performance modelling has two primary categories, one is Hybrid Simulation/Analytic modelling (*HSA*), and the other is Multi-Formalism Hybrid Performance (MFHP) modelling. Hybrid models appeared as early as the 1970s in [118, 128, 188], which were examples of HSA modelling, and aimed to compensate the high execution costs of simulations.

2.5.1 Hybrid Simulation/Analytic Modelling

Hybrid Simulation/Analytic (HSA) modelling was described by Schwetman [188], and defined and classified by Shanthikumar and Sargent in [190]. HSA modelling combines simulation models and analytic models, organizes and controls the sequence of model invocations, coordinates the data exchanges between submodels, and determines the combination of the results. According to the classification of Shanthikumar and Sargent, there are four classes of HSA models [190], which are:

1. Class 1 HSA models consist of an analytic model and a simulation model, which are completely independent of each other. Their solution procedures are also independent without any intermediate use of results from each other. The system behaviour of the HSA model can be determined by the results of the analytic model or the simulation model.
2. Class 2 HSA models consist of an analytic model and a simulation model, whose solution procedures are combined or interact with each other. The system

behaviour of the HSA model is determined by the joint results of the analytic model and the simulation model.

3. Class 3 HSA models use an analytic model as the primary model. The simulation model serves as a subroutine and provides some intermediate results for some parts of the system, which are used to determine or adjust some of the input parameters of the primary analytic model.
4. Class 4 HSA models use a simulation submodel as the main model. The results from solving the analytic model are fed into the simulation model.

This classification of Shanthikumar and Sargent can also be a guide to classify multi-formalism hybrid performance modelling, which extends HSA modelling to combine multiple analytic performance modelling techniques into a hybrid modelling framework, described in the next section.

Most of the previously developed HSA models were in the area of optimization and production planning [34, 35, 130], and performance evaluation of computer systems [27, 49, 125, 126].

2.5.2 Multi-Formalism Hybrid Performance Modelling

Multi-Formalism Hybrid Performance (MFHP) modelling combines multiple solution techniques into a single framework. For example, SMART [50] (Simulation and Markovian Analyzer for Reliability and Timing) is a unified performance evaluation framework integrated QNs, SPNs [149] and simulation; SHARPE [185] integrates directed acyclic task precedence graphs, QNs, Markov and semi-Markov models, and GSPNs [1, 142]; and *Möbius* [58, 61] (a multi-level and multi-formalism modelling framework) integrated the techniques, such as QNs, LQNs, GSPNs, and CPNs [110].

MFHP modelling is a hybrid performance modelling framework enabling the co-operation of multiple formalisms with each other. The cooperation, which includes

solver invocations, data exchange, and convergence control, is accomplished by a controller inside a framework. The first preliminary MFHP was IMSE [99] (Integrated Modelling Support Environment). It combined several solvers into one package, but, none of its solvers in the package exchanged data with the others. Subsequent MFHPs exchanged data between solvers and used the framework to control the overall solution. In MFHP modelling, some frameworks may also need some external model transformation tools to recognize the multiple formalisms, or an intermediate model file format as a bridge for exchanging data among the various formalism after invoking the solvers.

Similar to HSA modelling, MFHP constructs a set of submodels to describe an entire system. These submodels may be represented and solved by multiple formalism or techniques in an MFHP framework. When some submodels represent only parts of the system, the consistency between the submodels and the equivalency between the exchangeable performance metrics can be a challenge.

2.5.3 Hybrid Modelling Related to LQN

In [224, 225], Wu proposed a Hybrid Performance Modelling Methodology (HPMM), which combined LQN and GSPN [1, 142] models into the Möbius hybrid framework, to cope with the systems with decisions made in exception handling. A system is constructed into a LQN model and one or more GSPN models, in which GSPN models are used to handle the decision logic. Two solvers, SPNP [51, 207] for GSPN and LQNS for LQN models, are invoked iteratively until the entire model converges.

There are also other works combining the LQN analytic solution with simulations using HSA modelling. Verdickt et al. [210] developed a Class 4 HSA modelling framework, which integrated the LQN analytic solver into the network simulation tool *NS-2* [106]. The throughput of network entries calculated by LQNS are used to update the packet rates in the network simulation model and the delays found by

simulation are used to update the network delays in the LQN model.

On the other hand, Chiş and Jurca [49] developed a Class 3 HSA model, which combined simulation with an analytic solver using formalism similar to LQN. However, in Chiş’s model, all the calls from higher level servers are ‘forwarded’ to lower lever servers so the hybrid solution fails to capture the blocking effects of synchronous RPC [26] interactions.

Another Class 3 HSA model was developed by Li and Franks [133], which combined the LQN simulation and LQNS to model some specific behaviour or features, for example, fair share scheduling. The advantage of this hybrid model is that no model transformation is required and the data exchange is easier, since the hybrid model only uses LQN models. At the same time, simulation is only used to solve a subset of the entire model, where the special behaviour exists. Therefore, the major part of the system is solved analytically by LQNS, the efficiency and scalability of LQNS will be maintained in this hybrid approach. However, the performance metrics defined in the analytic model and the simulation submodel may not be equivalent in some systems, because of the existence of traffic dependencies among the LQN analytic submodels.

2.6 Modelling Exception Handling Decisions in Resource Allocations

Exception handling is an important form of decision. Exceptions are defined as “*Of the conditions detected while attempting to perform some operation, exception conditions are those brought to the attention of the operation’s invoker. The invoker is then permitted (or required) to respond to the condition.*” [91].

2.6.1 Wu's HPMM for Handling Decisions

Wu's work [224] is closely related to this research. He developed a Hybrid Performance Modelling Methodology (HPMM) to handle the exception decisions in resource allocations. His hybrid solution combines two modelling techniques, LQN and GSPN [1, 142].

His hybrid models were constructed from Use Case Maps (*UCM*) [3, 4, 32] scenarios. UCM is a visual notation to describe causal relationships between responsibilities [3], which are operations or tasks in a system. UCM is widely used to bridge requirements and system designs. Each responsibility in a UCM scenario becomes a behaviour cell with scores for these two techniques. A score of behaviour can be high, medium and low, which depends on how well a selected formalism is able to model a behaviour, how fast the model is solved, as well as the scalability of the chosen formalism. The scores for each kind of behaviours corresponding to LQN and GSPN are predefined in [224]. For example, LQN has 'high' scores on the responsibilities with queueing behaviour, and 'low' score on the responsibilities with decisions, while GSPN is the other way around. Two or more adjacent responsibilities with high LQN scores are grouped together and become an aggregated responsibility. Therefore, the size of a GSPN submodel can be reduced as it is generated from a UCM scenario with aggregated responsibilities. All the timed transitions in a GSPN submodel corresponding to those aggregated responsibilities have to get their mean service demands from the results from solving the corresponding LQN submodel.

The LQN submodel in the hybrid model is the entire model of a system, except that the decisions are represented by probabilities of all execution paths. These probabilities of branch paths are obtained by solving the corresponding GSPN submodel. The LQN submodel can be generated by UCM2LQN [167], which is a model transformation tool in UCMNavigator [146]. In each solution iteration, the LQN and GSPN submodels are solved alternatively; data are exchanged between submodels.

The HPMM framework controls the solver invocations and data exchanges.

Wu's HPMM has some disadvantages:

1. From the hybrid method itself, the HPMM does not enhance the modelling power of the LQN modelling technique (or queueing-based formalism) to handle decisions. Rather, in the HPMM, LQN is a method which is used to solve a part of hybrid models.
2. Overall, the hybrid solution is a state-based method. Therefore, it may still face the state explosion problem.
3. From the aspect of use, a hybrid modelling technique may be more complex in terms of model composition. Using hybrid modelling requires that modellers must have a good knowledge of both techniques, which are used to construct performance models. Understanding performance models expressed using different methods is essential to evaluate the correctness and usefulness of the entire hybrid model.
4. From the execution aspect, two submodels in two modelling techniques require two completely separate solution environments. LQNS and SPNP [51, 103], the solvers for LQN and GSPN submodels used in the HPMM, are invoked alternatively in each solution iteration.
5. Reading input files and writing to output files occur in every invocation of LQNS and SPNP. Also, data exchanges between GSPN and LQN submodels performed in every iteration, also require file I/O operations to retrieve data from an output file of one submodel to modify input parameters in an input file of the other submodel.

Compared to Wu's work, the method used in this research aims to extend the LQN models themselves to handle decisions. It does not complicate model compositions

at all because it does not require any external tools or techniques to recognize or model a decision. The probabilities of execution paths of a decision are modelled using dynamic parameters, which are specified in an input file through the templates of pre-defined decision patterns. The substitutions of all the dynamic parameters are performed internally through substitution functions. These dynamic parameters are recalculated in each iteration. From this point of view, the computation cost is increased, only by the number of iterations needed for achieving convergence with the dynamic parameters.

2.7 Summary

This section provides the necessary knowledge related this research. First, a brief description of major model-based and model-driven performance methodologies was presented. All the activities of MDE and MDPE eventually evaluate systems through performance models, which can be analytic, simulation, or hybrid models. Then, the major analytic performance techniques and hybrid performance modelling frameworks were also described. Next, the subject of this research, LQN performance modelling technique, was explained. Finally, the related work of modelling decisions in the context of exception handling was described.

Chapter 3

Interlocks

Interlocks are an artifact of the analytic solution of a LQN model when using the method of decomposition. With this approach a LQN model is subdivided into a set of submodels, and then each submodel is solved independently of each other. Interlocks arise because some requests that originate from the same client at an upper layer arrive at a lower layer server through different paths.

This chapter enumerates all identified interlocking cases. First, the interlocking phenomenon in LQN models and the problems with the existing interlock correction are described. Then, the improved general solution is explained and verified. Next, sending interlocks and the interlocking impacts to the solution of LQN models with second-phased services are addressed. Finally, the improved solution is applied to two models which combine multiple interlocks.

3.1 Interlocking Phenomenon

Interlocking [72, 220] is a traffic dependency that arises because a single customer from an upper submodel appears in two or more places as a client in a lower level submodel. Each client in the lower-layer submodel is a separate chain in the corresponding queueing model, so in effect, a customer is competing against itself. By not accounting

for interlock, too much delay is introduced into the queueing network of the lower submodel, resulting in overestimated residence times and underestimated throughput. Figure 3.1b shows the difference, in terms of $(\frac{\text{sim}-\text{lqns}}{\text{sim}} \times 100)$, between the analytic solution ('lqns') and simulation ('sim') results when interlocking is not taken into account when solving the two-tier client-server model shown in Figure 3.1a. Note that the largest error occurs when there is only one customer in the model. The non-interlocked solution finds contention delays at the common processor when in fact there is none. The effect is particularly noticeable with a small number of interlocked clients [72].

(a) The LQN model

(b) Without interlocking corrections

Figure 3.1: Interlocking effects on a client-server model.

Interlocks may take different forms when they occur in LQN models, which are:

1. Split interlocks. The flow originated from a common source is split, then through multiple paths (intermediate tasks) arrive at a common server, which is referred to as an interlocked server. Split interlocks are the most common interlocks appearing in LQN models. The model in Figure 3.1a is a typical example of a split interlock.
2. Sending interlocks. Both the intermediate server(s) and the common source make requests to the same interlocked server in the same submodel. The differ-

Figure 3.2: A model that combines all forms of interlocks.

ence between sending interlocks and split interlocks is that the common source of a sending interlock is a direct client of the interlocked server.

3. Interlocks with external contentions. The interlocked server directly receives some requests (flows) that are completely unrelated to the interlocked flows.
4. Mixed interlocking flows. There are two cases: first some non-interlocked flows merge into interlocked flows and arrive at the interlocked server through intermediate task servers; the other is that the flows originated by multiple common sources arrive at an interlocked server.

The LQN model shown in Figure 3.2, combines most of the interlock cases. First, there are two split interlocks, which are tasks $c1$, task $s0$, entry $e1$ (task $s1$), and tasks $c2$, entry $e2$ (task $s1$), task $s2$. Since both of these two split interlocks arrive at processor $p0$, they actually construct the second case of mixed interlocking flows. Next, tasks $c4$ and $s3$ have a sending interlock, and task $s4$ generates “external” flow. Finally, the split interlock from the client $c1$ has mixed flows due to task $c0$. This model has two submodels; **Submodel 2** is the submodel where interlocks occur.

Sending interlocks are considered in Woodside’s solver [220]. The current Franks’s

LQNS solver [74] handles more cases, such as sending and split interlocks, and part of mixed interlock flow. However, the interlocks with external contentions are a newly identified type of interlocking effects and were not handled previously.

A common source of a split or sending interlocks can either be a single entry or a common task. In the case of a common task acting as a common source, multiple entries directly or indirectly make requests to the interlocked server throughput multiple paths. If it is the case in a sending interlock, it is referred as to a *Type3 sending interlock* and discussed in Section 3.3, since it was not handled previously either.

In order to present the rationality of the improved solution to handle all these interlocks, the problems existing in the current solver are described in the following section.

3.1.1 Problems in the Current Solver

The corrections to interlocks in the current LQNS are done as follows. Equation (3.1) shows the queue length calculation for MVA [181] where L_{mk} is the queue length for chain k at station m , $\mathbf{N}$ is the population vector by chain, W_{mk} is the waiting time for chain k at station m and λ_{mk} is the flow to station m for chain k .

$$L_{mk}(\mathbf{N}) = \lambda_{mk} W_{mk}(\mathbf{N}) \quad (3.1)$$

The overall throughput λ_{mk} in (3.1) can be split into two components: the non-interlocked flow $\lambda_{mk}^{\text{NOIL}}$, and the interlocked flow λ_{mk}^{IL} . To correct for interlock, the flow of *one* interlocked customer is removed from the queue length calculation, giving:

$$\begin{aligned} L_{mk}(\mathbf{N}) &= \left(\frac{n-1}{n} \lambda_{mk}^{\text{IL}} + \lambda_{mk}^{\text{NOIL}} \right) W_{mk}(\mathbf{N}) \\ &= \left(1 - \frac{1}{n} \frac{\lambda_{mk}^{\text{IL}}}{\lambda_{mk}} \right) \lambda_{mk} W_{mk}(\mathbf{N}) \end{aligned} \quad (3.2)$$

According to (3.2), all the interlocked flows (without mixed interlocked flows) from a common source receive the same amount of adjustment. The fraction of $\frac{\lambda_{mk}^{IL}}{\lambda_{mk}}$, which addresses mixed interlocked flows, is found by the *Path Finder* [72]. The *Path Finder* is used to locate the paths in the model consisting of nested remote procedure calls from an originating client task to a server. These paths are located by following the requests made from one task to the next using a depth-first recursive search. By incorporating the call rate information associated with each request while tracing a path, the flow component originated by any parent to any entry can be determined.

However, two problems have been identified with the existing interlock adjustment in the current LQNS:

1. The waiting times at intermediate tasks are incorrect, though the residence time at the clients (or overall throughput) is usually quite accurate.
2. Flows from sources which are not associated with the interlock called *external flows* can cause severe overestimation of utilization.

3.1.1.1 Incorrect Distribution of Waiting Time

Figure 3.1a shows a simple model that is used here to demonstrate the effect of split interlocks. A customer from client $c0$ can visit either $s0$ or $s1$, but not both at the same time, when this model is partitioned into two submodels for solving analytically. The one customer from $c0$ is treated as two customers in *submodel 2* because $s0$ and $s1$ are independent.

Figure 3.3 shows the difference in the throughput and selected waiting times from solving the Figure 3.1a model, from one to ten clients using the algorithm from [72]. With one client, the existing interlock correction algorithm is “perfect”; all interlocked flow is removed and no false contention is introduced. However, when the model is solved with more than one client, the existing algorithm initially overestimates the waiting time in *submodel 1* and underestimates the waiting time in *submodel 2*. As

Figure 3.3: Comparing the existing interlock correction and simulation (Figure 3.1a model).

the number of clients increases and the interlock effect is diminished, the error in both estimates drops, though tends towards underestimation. The largest error of the mean queueing delays happens when there are two customers.

3.1.1.2 Interlock With External Contention

Figure 3.4a shows a processor which has contention not only from a common ‘interlocked’ task but also from an ‘external’ source. Figure 3.4b shows the utilization for the interlocked processor $p1$. This processor is a fixed-rate, single server so its maximum utilization cannot exceed one. However, with the existing interlock calculation enabled, its computed utilization always exceeds this limit.

The problem with this model is that both flows from tasks $s0$ and $s1$ (chain 1 and chain 2) are fully interlocked (i.e., $\lambda_{mk} = \lambda_{mk}^{\text{IL}}$) since the reference task $c0$ has only one customer. Therefore, using (3.2), $L_{mk}(\mathbf{N}) = 0$ for both client chains 1 and 2. Consequently, when a customer from $s2$ (chain 3) arrives, it sees nothing in the server queue at all times, which it is not correct. In reality, the customer arriving from $s2$ may see a customer from *either* $s0$ or $s1$, but not both at the same time. A customer arriving by way of either $s0$ or $s1$ may see the one customer from $s2$ in the

Figure 3.4: An interlock with an external contention.

server queue.

3.2 Improved Interlock Calculation

The previous section has shown both the need to correct for interlocks and deficiencies in the existing algorithm. This section shows an improved approach which:

1. applies adjustments to the upper level submodels where the intermediate tasks are acting as servers, in addition to the adjustments made at the lowest level submodel;
2. takes into account of the existence of external contention, which means the interlocked server may accept some non-interlocked flows; and
3. applies the interlock adjustments to the queue length with one less customer $L_{mk}(\mathbf{N} - \mathbf{e}_j)$ rather than $L_{mk}(\mathbf{N})$, where $L_{mk}(\mathbf{N} - \mathbf{e}_j)$ is used to found the mean residence time of chain k at a queueing server m in (2.1).

This new approach is divided into a *lower-level* and an *upper-level* adjustments described next.

3.2.1 The Lower-Level Adjustment

The *lower-level* adjustment is used to remove the contention caused by having a customer from a higher level submodel being represented by two or more chains in the lowest layer submodel where the customer may contend with itself. This adjustment is similar to the previous throughput interlock technique [72, 220].

To address all the cases of interlocks described earlier, three variables, namely the Interlocking Probability ($\text{Pr}(\text{IL})$), Interlocking Ratio (IR), and Interlocking Relationship (ir), are introduced.

3.2.1.1 Interlocking Probability

The *Interlocking Probability* is the fraction of an interlocked flow caused by the additional chains in the lower submodel; this is the flow to be removed. In (3.2), the interlock probability can be expressed as

$$\text{Pr}(\text{IL}) = \frac{1}{n} \quad (3.3)$$

However, the interlock probability is not a simple ratio, as shown in (3.3). For example, in Figure 3.1a, the probability of intermediate tasks s_0 and s_1 competing with each other is the product of their utilization U_{s_0} and U_{s_1} , if they are independent. However, this probability is not the actual probability of these two tasks being busy simultaneously, $\text{Pr}(s_0, s_1)$. The difference between these two probabilities is where the ‘impossible’ interlocked flow hides. The ratio of these two probabilities equals to $1 - \text{Pr}(\text{IL})$, as $\text{Pr}(\text{IL})$ is the part that has to be removed. The joint probability $\text{Pr}(s_0, s_1)$ can be obtained from solving a Markov chain. However, this approach does not scale since solving the Markov chain for finding the interlock probabilities amounts to solving the Markov chain corresponding to the complete model.

To eliminate the problem by solving a Markov chain, an approximation is used

Figure 3.5: An invalid case because of interlocking with only two customers in the model. (Task S1 and S2 are shaded (in blue) to represent that they are busy.)

for finding the interlock probabilities, which does not require finding the state probabilities. This approach, shown in (3.4), uses a simple ratio of the various queueing times found during the solution of the network.

$$\Pr(\text{IL}) = \begin{cases} \frac{1 - \left(\frac{q_i^{\text{upper}}}{\delta_i S_i} \right)}{QL} & \delta_i > 0 \\ 1 & \delta_i = 0 \end{cases} \quad (3.4)$$

where:

δ_i :is the difference between the populations of the common source and an intermediate server task i ;

q_i^{upper} :is the mean queueing delay to server i ;

S_i :is the mean service demand of server i ;

QL :is the sum of queue length of all the intermediate server tasks receiving requests from a common source.

Figure 3.5 is used to explain the (3.4), which shows a case of impossible interlocking flows. Figure 3.5a shows the contention between tasks $S1$ and $S2$ at the

lower-level submodel, when they are both busy. Figure 3.5b shows an impossible flow of interlocking when there are only two customers in the model, which was not detected by the previous solution. After **Submodel1** is solved, the residence time for $S1$ includes waiting at $P2$, due to contention from the customer at $S2$ (and vice versa). The problem is the case 2 (Figure 3.5b) and **customer 1** arrives at $S1$, there can be no queueing delay at $P2$ because $S1$ is a single server. Therefore, any queueing delay found from solving **Submodel 2** must be removed from $S1$ when solving **Submodel1**. This also means that if there are contentions at $P2$ at **Submodel 2**, which is shown in Figure 3.5a, there can be no queueing delay at both $S1$ and $S2$ at **Submodel 1**. Therefore, the flow, which arrives at $S1$ during the contention (q_i^{upper}), must be removed when solving **Submodel 2**. This is the reason why (3.4) uses the queueing delays at the upper level, q_i^{upper} , to estimate the interlocking probability $\text{Pr}(\text{IL})$. The fraction of $\frac{q_i^{\text{upper}}}{\delta_i S_i}$ in (3.4) is used to approximate the percentage of utilizations that needs to be removed. Also, as the number of customers at the common source increases, the fraction of $\frac{q_i^{\text{upper}}}{\delta_i S_i}$ is decreased by the denominator δ_i . The denominator QL is used to remove the effect of the common source task.

Equation (3.4) is for single intermediate servers only. When $\delta_i = 0$, there is only one customer from the common source, so all of the queueing delay occurring at the lower level must be removed, thus $\text{Pr}(\text{IL}) = 1$. When $\delta_i > 0$, the part $\left(\frac{q_i^{\text{upper}}}{\delta_i S_i}\right)$ in (3.4) is used to approximate the probability that the task i cannot see any requests from other intermediate servers when it issues a request to the interlocked server.

Table 3.1 shows the interlock probability derived from the results simulating the model in Figure 3.1a and the interlock probability found using the approximation in (3.4) for various service time values for task $s1$. The approximation is sufficiently accurate and far better than the approach in (3.3), which use $\text{Pr}(\text{IL}) = 0.5$ for all cases. Furthermore, $\text{Pr}(\text{IL})$ changes with the number of customers, unlike the old approximation.

Table 3.1: Interlocking Probability for the model in Figure 3.1a.

Test Case	Interlock Probability $\Pr(\text{IL})$			
	Simulation		Approximation	
	s0	s1	s0	s1
1	0.207	0.411	0.230	0.433
2	0.286	0.348	0.321	0.383
3	0.328	0.302	0.371	0.345
4	0.354	0.266	0.402	0.316
5	0.371	0.238	0.424	0.292
6	0.384	0.216	0.440	0.273
7	0.394	0.197	0.452	0.256
8	0.402	0.182	0.462	0.242
9	0.408	0.168	0.470	0.231
10	0.414	0.156	0.477	0.220

3.2.1.2 Interlocking Ratio and Interlocking Relationship

The second variable, *Interlocking Ratio* (IR), is the amount of interlocked flows among the entire flow arriving from a single client chain:

$$IR_{mk} = \frac{\lambda_{mk}^{\text{IL}}}{\lambda_{mk}} \quad (3.5)$$

If a flow from a single chain k has $IR_{mk} = 1$ or 0 , this flow is either an *entire interlocked flow* or *non-interlocked flow*. If IR_{mk} is in between $(0,1)$, this flow is referred to as a *mixed interlocked flow* from chain k . Mixed interlocked flows received at an *interlocked server* may refer to one of the following two cases:

1. all incoming flows received by the interlocked server m , contain a part of or an entire interlocked flow, which means:

$$\forall k : 0 < IR_{mk} \leq 1$$

This case indicates that external contention exists at some intermediate servers at higher levels in the model.

2. one or more non-interlocked flow(s) exist among the incoming flows received by the interlocked server m , which means:

$$\exists j : IR_{mj} = 0, \text{ and } \forall k : 0 \leq IR_{mk} \leq 1$$

This case shows the existence of external contention at either the interlocked server or higher level intermediate servers. The processor $p0$ in Figure 3.2 is an example of an interlocked server having the case 2 mixed interlocked flows.

The third variable, *Interlocking Relationship* ($ir_{k,j}$) differentiates the “relevance” between a chain pair k and j (chain j is the chain with the incoming customer). This variable can have a value of either zero or one.

If $ir_{k,j}$ is 0, then chain j and chain k are irrelevant. This case includes three situations:

1. Both chains k and j have (entire or mixed) interlocked flows, but from different sources;
2. Only one of chains k and j has an interlocked flow;
3. Neither of chain k nor chain j has an interlocked flow.

If $ir_{k,j}$ is 1, then chain j and k are relevant. In this case, chains k and j may belong to the same split interlock or the same sending interlock; or only part of both flows has the same source.

Figure 3.2 in the previous section also shows all possible cases of the interlocking relationships may appear between chain pair k and j . The flows from chains $e2$ and $s2$ are interlocked because of the shared upper layer task $c2$, therefore, they can see each other partially. Similarly, client $c4$ is a parent of client $s3$, which belongs to the same sending interlock [72]. Flows from $c4$ and $s3$ are not able to see each other, but they can see the customers from the other client chains. On the other hand, the flow

from $s0$ contains only part of the interlocked flow originating from entry $e1$ of chain $s1$. Finally, the flow from chain $s4$ is irrelevant to any of the other chains, so it is able to see customers from them all.

Both interlocking ratio and interlocking relationship are used together to distinguish which flows a customer may see when it arrives from chain j . The interlocking ratio IR_{mk} and interlocking relationships ir_{mjk} are obtained by the *Path finder*. The interlock probability measures how much of each flow an arriving customer may see. Therefore, the lower-level adjustments are performed based on these parameters, which are explained in the next section.

3.2.1.3 The Lower-Level Adjustment

The example shown in Section 3.1.1.2 shows that adjustments made directly to $L_{mk}(\mathbf{N})$ are incorrect because the mean waiting time W_{mk} for chain k at server m does not reflect the queueing delays caused by the external flows seen by an interlocked flow from chain k . Therefore, the adjustments have to be made before calculating the mean waiting time W_{mk} . The new solution is performed by adjusting the queue length with one less customer, $L_{mk}(\mathbf{N} - \mathbf{e}_j)$, instead.

In (3.2), a flow (λ_{mk}) may contain two parts, the non-interlocked flow, $\lambda_{mk}^{\text{NOIL}}$, and the interlocked flow, λ_{mk}^{IL} [72]. The latter flow needs to be adjusted by the factor $1 - \Pr(\text{IL}_{mk})$ to remove the flow that does not exist. Let λ'_{mk} represent the adjusted flow, then the rate of adjustment $\mathbf{r}_{mk}^{\text{lower}}$ (the ratio of λ'_{mk} and λ_{mk}) is found by the following:

$$\begin{aligned}
 \lambda'_{mk} &= \lambda_{mk}^{\text{NOIL}} + \lambda_{mk}^{\text{IL}}(1 - \Pr(\text{IL}_{mk})) \\
 &= \lambda_{mk}(1 - IR_{mk} \times \Pr(\text{IL}_{mk})) \\
 \mathbf{r}_{mk}^{\text{lower}} &= \frac{\lambda'_{mk}}{\lambda_{mk}} = 1 - IR_{mk} \times \Pr(\text{IL}_{mk})
 \end{aligned} \tag{3.6}$$

However, the relationship between the incoming customer and the customers already in the queue must be taken into account, which is addressed by both the interlocking ratios and interlocking relationships, and shown in (3.7).

$$\mathbf{r}_{mk}^{\text{lower}} = 1 - \Pr(\text{IL}_{mk}) \times IR_{mk} \times IR_{mj} \times ir_{mkj} \quad (3.7)$$

The existence of the external contentions can be addressed by IR and ir . If any of the flows from the chain pair k and j have a non-interlocked flow, ir_{mkj} is 0 and one of IR_{mk} and IR_{mj} or both will become zero. Therefore, they can see the unadjusted flows of each other. The product of IR_{mk} and IR_{mj} is used to measure the portion of a flow from chain k that can be seen by chain j , in the case where the chain pair k and j have the same source. The interlocking relationship, ir_{kj} , is especially essential for the case of flows from the chain pair k and j when both have interlocked flows which come from different sources originally. Since both IR_{mk} and IR_{mj} are greater than zero, these two flows are not able to see each other or only see each other partially. The use of ir_{kj} (which is 0 in this case and indicates these two flows are irrelevant), allows these two interlocked flows to see fully each other.

This adjustment rate $\mathbf{r}_{mk}^{\text{lower}}$ is used to modify the $L_{mk}(\mathbf{N} - \mathbf{e}_j)$ that have ‘impossible’ customers. Since the arriving customer should be able to see all the customers from its own chain, i.e. $k = j$, $L_{mj}(\mathbf{N} - \mathbf{e}_j)$ does not need to be changed.

$$L'_{mk}(\mathbf{N} - \mathbf{e}_j) = \begin{cases} L_{mk}(\mathbf{N} - \mathbf{e}_j) & k = j \\ L_{mk}(\mathbf{N} - \mathbf{e}_j) \times \mathbf{r}_{mk}^{\text{lower}} & k \neq j \end{cases} \quad (3.8)$$

From (3.8), if a customer arrives from a chain with non-interlocked flow only, this customer sees all unadjusted $L_{mk}(\mathbf{N} - \mathbf{e}_j)$ from any other client chains, because $IR_{mj} = 0$ and $ir_{mkj} = 0$. Conversely, if a customer arrives from a chain with an interlocked flow, $0 < IR_{mj} \leq 1$, it sees all unadjusted $L_{mk}(\mathbf{N} - \mathbf{e}_j)$ from any irrelevant chains

with a non-interlocked flow or an interlocked flow, and sees either nothing or part of $L_{mk}(\mathbf{N} - \mathbf{e}_j)$ from those relevant chains with interlocked flows, depending on $\text{Pr}(\text{IL}_{mk})$ and IR_{mk} .

For Bard-Schweitzer approximate MVA [186], the queue length with one less customer is estimated using $\frac{n-1}{n}$, then (3.8) becomes:

$$L'_{mk}(\mathbf{N} - \mathbf{e}_j) = \begin{cases} \frac{N_k-1}{N_k} L_{mk}(\mathbf{N}) & k = j \\ \mathbf{r}_{mk}^{\text{lower}} L_{mk}(\mathbf{N}) & k \neq j \end{cases} \quad (3.9)$$

The complexity of these newly added variables, which are used to calculate interlocking effects in the improved solution, is worth to mention since these variables have an influence on the efficiency of the LQNS solver. First, the variables, ‘Interlocking Probability’ ($\text{Pr}(\text{IL}_{mk})$) and ‘Interlocking Ratio’ (IR_{mk}) are based on client chains and are calculated once within the LQN submodels in every iteration. These $\text{Pr}(\text{IL}_{mk})$ and IR_{mk} are passed into MVA submodels and remain as constants during the MVA iterations. The calculations of interlocking Probabilities and interlocking ratios replace the counterpart estimation that existed in the previous solution. The variable ‘Interlocking Relationship’ ($ir_{m,k,j}$) is based on pairs of client chains. The calculation of the interlocking relationships is through *PathFinder* [72] and is performed only once during the entire solution. Therefore, the improved solution does not adversely affect the efficiency of the analytic solution.

3.2.2 The Upper-Level Adjustment

When interlocking occurs, the intermediate servers in an upper layer submodel become load dependent. The mean residence time (W_i) for a request at an intermediate server i is the sum of its own execution time (S_i), the mean queueing time for all requests already in the server’s queue, and the Mean Residual Life (MRL) time of

the request currently in service, which is shown in (3.10).

$$W_i(N) = S_i + S_i Q_i(N - 1) + \text{MRL}_i \quad (3.10)$$

Where, $Q_i(N - 1)$ represents the queue length, *not* including the customer in service. If the service time is assumed to be exponentially distributed, the Mean Residual Life equals the mean service time of the request a being served by the server i . However, the mean service time by the server i may contain some nested queueing delays when it makes further requests to some lower level servers. If these nested queueing delays have the same source as the request currently in service, the MRL of this request may vary upon the number of customers in the queue.

Assume that the intermediate server $s1$ only receives requests from one client task $c0$, the common source of all intermediate servers, for example, the model shown in Figure 3.1a. Suppose the client task has N customers, and there are $N - 1$ customers already in the queue for the server $s1$. If the N^{th} customer also is going to this server, at the moment of arrival, the mean remaining service time of the request in service is its own actual service demand without any nested queueing delay. The reason is that none of the customers can be found in any other queue in the system. The nested queueing delays in MRL result in excessive residence times (W_i) at the upper level. Therefore, the MRL error only happens when all the customers from the common source are in the same queue of server i , and this is proportional to the probability that all customers from the common source are in the same queue at the same time. This effect only becomes significant when there are a small number of customers at the common source. Furthermore, the probability that all of the customers being in the same server queue at the same time decreases exponentially as the number of customers at the common source goes up. For example, for the model in Figure 3.1a, the probabilities are less than 0.3, 0.1 and 0.03 for 2, 3 and 4 customers respectively.

3.2.2.1 Calculating Upper-Level Adjustment

The upper-level adjustment only applies to the flows from the common source, since the key point of the interlock calculation is handling the flows originating from a single source. (It is the common source that issues the interlocked flow which takes multiple paths and arrives at a common lower-level server). At the upper-level submodels, only the flows from the common source arriving at the intermediate servers are affected by interlocks and need to be adjusted. Therefore, all the calculations at intermediate servers use single chain calculations only.

The purpose of the upper-level adjustment is to remove the extra queueing delay caused by the nested queueing delays in the MRL. For example, in Figure 3.1a, with only 2 customers at the common source $c0$, the mean queueing delay at the lower level, q_i^{lower} , contributes nothing to the mean queueing delay q_i^{upper} .

If the number of customers N for a common source is greater than 2, q_i^{lower} affects q_i^{upper} only when all N customers are in the same server queue. Therefore, the extra queueing delay at an upper level, $\Delta W_i(\mathbf{N})$, found by (3.11), is the part of q_i^{lower} caused by those same source requests at the lower level with the probability of all N customers in the same server queue:

$$\Delta W_i(\mathbf{N}) = \Delta MRL = \Pr(n_i = N) \times q_i^{\text{lower}} \quad (3.11)$$

Using MVA, the mean waiting time at a server i was shown in (3.12), where $L_i(N-1)$ is the sum of queue lengths with one customer removed. Thus, the difference in the mean waiting time at the intermediate server i can be represented using the difference in the queue length with one customer removed, shown in (3.13).

$$W_i(N) = S_i + S_i \times L_i(N-1) \quad (3.12)$$

$$\Delta W_i(N) = S_i \times \Delta L_i(N-1) \quad (3.13)$$

By combining (3.11) and (3.13), the upper-level adjustment rate r^{upper} can be expressed as:

$$r^{\text{upper}} = \frac{\Delta L_i(N-1)}{L_i(N-1)} = \frac{P(n_i = N) \times q_i^{\text{lower}}}{L_i(N-1) \times S_i} \quad (3.14)$$

To avoid calculating the probability of all customers from the common source being in the same queue simultaneously, the fraction of $\frac{L_i(N-1)}{(N-1)}$ is used as an approximation to the probability. Then r^{upper} can be calculated using (3.15).

$$r^{\text{upper}} = \begin{cases} \frac{q_i^{\text{lower}}}{S_i \times \frac{(N-1)}{N}} & N > 1 \\ 0 & N = 1 \end{cases} \quad (3.15)$$

The advantages of using this approximation are:

1. there is no need to calculate the state probabilities;
2. the adjustment rate is calculated only once at the beginning of an MVA solution;
3. using a similar adjustment to the queue length with one customer less, simplifies the solver implementation and improves the solver's efficiency.

3.2.2.2 Existence of Mixed Interlocked Flows

Mixed interlocked flows can refer to both cases mentioned in Section 3.2.1.2. The existence of mixed interlocked flows results in q_i^{lower} containing some nested queueing delays caused by non-interlocked flows at a lower level. These delays should *not* be removed during the upper-level adjustment.

The effective q_i^{lower} , is the one used to deduct the extra queue length at an upper level, and is denoted as $q_i^{\text{lower}'}$. It is obtained by finding the rate of the lower level queueing delay caused by non-interlocked flows, (NIR_{mk}) , using:

$$q_i^{\text{lower}'} = q_i^{\text{lower}} \times (1 - NIR_{mk}) \quad (3.16)$$

At a lower level, $L_{mk}^{\text{NOIL}}(\mathbf{N} - \mathbf{e}_j)$ is the part of the queue length (with one customer removed from chain j), caused by non-interlocked flows only, and found by (3.17). Then,

$$L_{mk}^{\text{NOIL}}(\mathbf{N} - \mathbf{e}_j) = L_{mk}(\mathbf{N} - \mathbf{e}_j) \times (1 - IR_{mk}) \quad (3.17)$$

NIR_{mj} is the fraction of the total $L_{mk}^{\text{NOIL}}(\mathbf{N} - \mathbf{e}_j)$ and the total queue length, and can be found by (3.18) when the interlocked server has a processor-sharing (PS) queueing discipline, and (3.19) if the server has a FIFO queue.

$$NIR_{mj} = \frac{\sum_{k=1}^K L_{mk}^{\text{NOIL}}(\mathbf{N} - \mathbf{e}_j)}{1 + \sum_{k=1}^K L'_{mk}(\mathbf{N} - \mathbf{e}_j)} \quad (3.18)$$

$$NIR_{mj} = \frac{\sum_{k=1}^K L_{mk}^{\text{NOIL}}(\mathbf{N} - \mathbf{e}_j) \times S_{mk}}{S_{mk} + \sum_{k=1}^K L'_{mk}(\mathbf{N} - \mathbf{e}_j) \times S_{mk}}, S_{mk} > 0 \quad (3.19)$$

where: S_{mk} is the mean service demand (non-zero) of chain k at server m .

After applying these changes, r^{upper} is found using

$$r^{\text{upper}} = \begin{cases} \frac{q_i^{\text{lower}}(1 - NIR_{mj})}{S_i(N-1)} & N > 1 \\ 0 & N = 1 \end{cases} \quad (3.20)$$

3.2.3 Testing the Improved Solution

This section enumerates all cases of split interlocks, which frequently occur in LQN models, to verify the approximation presented in the previous section. Test cases include a simple split interlock without external contentions, a split interlock with an external contention, and a split interlock with mixed flows and external contention. All the models are solved analytically by using LQNS without any interlock corrections, with the previous interlock adjustment, and the new interlock adjustments discussed in this chapter. All the results [135] are compared to simulations using their error rates. All simulations were run with confidence intervals of 95%, $\pm 1.0\%$. The models presented in this section are models where interlocking effects have serious

impacts.

3.2.3.1 Split Interlocks

Figure 3.6 shows the percentage difference in the waiting time and throughput using the new and old interlock calculation for the model shown in Figure 3.1a. These results show that the waiting time difference between tasks $c0$ and $s0$ (the *upper* submodel) is much smaller with the new method; the maximum difference of about 4% occurs when there are two clients. With the old technique, shown in Figure 3.3, the difference varies from too big by 24% to too little by -12% . At the processor (the lower submodel) the old technique consistently underestimates the waiting time; its maximum error of -37.6% occurs with two customers. The new method overestimates the waits at only 4% with seven customers. Finally, the difference in throughput at task $c0$ is underestimated by roughly 1% with the new method. The old technique overestimates by 2.5% on average.

Figure 3.6: Comparing the new interlock correction with simulation (Figure 3.1a model).

Table 3.2 shows the mean and standard deviation in the error rates (compared to simulations) from using no interlock calculation, the previous correction, and new

calculation when randomly varying the service time and call rates by $\pm 100\%$ from their values, shown in Figure 3.1a, and varying the number of customers between 1 to 4. The new algorithm's results are consistently better when compared to the old approach.

Table 3.2: Results of the Figure 3.1a model with randomized parameters.

Method	Error Rates								
	none			lqns			new		
	λ	$W_{c0,s0}$	$W_{s0,p1}$	λ	$W_{c0,s0}$	$W_{s0,p1}$	λ	$W_{c0,s0}$	$W_{s0,p1}$
mean	-9.16	21.94	3.08	2.91	8.38	-31.72	2.35	-4.09	-8.19
sdev	3.69	18.67	10.14	1.05	11.98	3.83	1.03	4.96	1.25

3.2.3.2 Split Interlock with External Contention

Table 3.3 shows the difference in the results of the throughput from the two client tasks, the utilization of the interlocked processor, and the waiting time along one path for the model shown in Figure 3.4a for 100 randomly parameterized models. The service demands and rates were chosen from a range of $\pm 100\%$ from the original value. The number of customers at $c0$ was chosen from 1 to 4. The solution of this model using the original interlock algorithm resulted in an infeasible utilization of the processor $p2$ ($U_{p2} > 1$). The new algorithm is clearly superior to the old one for all performance measures. Furthermore, the utilization of the interlocked processor

Table 3.3: Results of the Figure 3.4a model with randomized parameters.

$\% \Delta$	λ_{c0}	λ_{s2}	U_{p1}	$W_{c0,s0}$	$W_{s0,p1}$	$W_{s2,p1}$
model4.none						
mean	-9.69	11.53	-0.23	16.81	12.29	-18.35
sdev	4.95	4.13	0.11	12.01	19.43	7.36
model4.lqns						
mean	0.59	51.70	23.65	6.39	-8.68	-57.79
sdev	0.69	28.23	15.10	7.03	5.03	25.08
model4.new						
mean	-0.18	-2.42	-1.04	0.21	-0.85	4.22
sdev	0.58	1.44	0.61	3.02	1.30	2.52

Figure 3.7: Model with external contention to a common entry.

is no longer infeasible.

3.2.3.3 External Contention to a Common Entry

Figure 3.7 shows an interlocked model where one of the interlocked paths receives additional flow from another source. To account for interlocking, the flow from $c0$ must be separated from $c1$.

Table 3.4 shows the difference in the throughput, waiting times and utilization for solutions with no interlock corrections, the old algorithm, and the new algorithm when compared to simulation. The service demands and rates were chosen from a range of $\pm 100\%$ from the values shown in Figure 3.7. The number of customers from clients $c0$ and $c1$ was randomly chosen from a range of 1 to 4. Again, the new algorithm is superior to the old one. However, it too produces larger errors for the results on the interlocked path with contention ($c1, s1, p1$), though these results are still better than those for the old interlock algorithm.

All the results of these test cases show that the improved solution is able to produce more accurate results than the previous solution.

Table 3.4: Results of the Figure 3.7 model with randomized parameters.

$\% \Delta$	λ_{c0}	λ_{c1}	$W_{c0,s1}$	$W_{c1,s1}$	$W_{s1,p1}$	$W_{s2,p1}$	U_{p1}
model7-1.none							
mean	-5.26	-22.69	28.22	82.94	87.33	68.75	-18.50
sdev	0.84	5.71	13.61	43.42	91.26	86.43	4.08
model7-1.lqns							
mean	-1.70	-14.94	13.67	56.73	25.25	30.12	-11.68
sdev	1.84	5.72	8.19	30.46	39.23	51.27	3.92
model7-1.new							
mean	0.37	-9.60	0.54	34.32	4.34	-2.53	-7.20
sdev	2.09	2.71	5.09	17.27	2.86	3.85	1.75

3.3 Sending Interlocks

The section focuses on sending interlocks, which were recognized previously in Woodside's solver [220] and Franks' solver [81]. Sending interlocks refer to interlocks whose intermediate server(s) and the common source make requests to the same interlocked server in the same submodel. In terms of sending interlocks, the common source is also referred to as parent task and an intermediate task server as a child task. The model in Figure 3.8 is a simplified sending interlock. Sending interlocks occur quite often in LQN models, for example, a software task which requires some services from another task and both of these tasks are executing on the same processor. Like other interlocking effects, sending interlocks have extraneous queueing delays among interlocked flows, which result in underestimated throughput and server utilization.

The reason why sending interlocks have to be singled out is that parent and child client chains double the amount of traffic arrived at the interlocked server, because both parent and child are client tasks in the same layer, and they generate flows individually. Therefore, sending interlocks cause impacts to those flows from the client chains irrelevant to them (external contentions) as much as to themselves. Consequently, the way to correct sending interlocks by applying the improved solution (discussed in the previous section) needs to be specified explicitly.

Figure 3.8: A LQN model of a sending interlock.

3.3.1 Types of Sending Interlocks

Three types of interlocking effects caused by sending interlocks are described here.

1. *Type1* sending interlocks were recognized and partially handled by Woodside et al.'s solver [220] and Franks' solver [81], which have a form shown in Figure 3.8. *Type1* sending interlocking effect is significant when the common source has a small number of customers, for example, one or two customers only.
2. *Type2* sending interlocks have the same form as *Type1*, but the parent task of *Type2* has a medium number of customers, for example, 20-30 customers. A large error rate in the parent task's queueing delay is the sign of *Type2* sending interlocks. The interlocking effects do not decrease as the number of the customers of the common source increases.
3. *Type3* sending interlocks refer to the sending interlocks whose common source is a common task. The common task can provide multiple services by multiple LQN entries, and these entries generate traffic individually to the interlocked server through multiple paths. For example, once one of the entries is blocked by its included service, the other entries may be blocked as well if the parent task is a single threaded task. *Type3* sending interlocks were not handled previously.

(a) Without corrections

(b) Previous solution with interlock corrections

Figure 3.9: Results without corrections and with the previous interlock approximation for the Figure 3.8 model.

Type1 interlocking effect decreases rapidly as the number of customers of the parent task increases. Figure 3.9a shows all the queueing delays related to the interlocks as well as the throughput of the common client (the parent task). For the model in Figure 3.8, the error rate of the results (without any corrections) is less than 5% when the parent task has four customers. *Type1* interlocks effect were handled previously; the server utilization and throughput of the common source were accurately calculated. However, calculating the queueing delays along interlocking flows is a weakness of the previous solution. The interlocking effects on queueing delay are reduced by half only using the previous solution, which error rates are up to -20~-30%, shown in Figure 3.9b.

Type2 interlocking becomes significant after the number of customers is greater than 10 in the model in Figure 3.8. Furthermore, both Figures 3.9a and 3.9b show the existence of *Type2* interlocks. The previous solution does not have any correction for them. *Type2* sending interlocks demonstrate the fact that interlocking effects are not only significant in the models with a small number of customers.

Since *Type3* sending interlocks have not been handled yet, it will be discussed in the following section.

3.3.2 Improvements on Sending Interlocks

This section explains how to apply the improved solution to handle all types of sending interlocks, and includes the case with external contentions.

3.3.2.1 *Type1* and *Type2* Sending Interlocks

By applying the improved solution, the interlocking effect between a parent and child client chains of a sending interlock can be reduced by using equation (3.7) in Section 3.2.1.3. For this case, the interlocking relationship $ir_{j,k}$ between the parent and a child is 1. However, the calculation of Pr(IL) in (3.4) may not be suitable

Figure 3.10: Results with the new solution for the Figure 3.8 model.

for the parent of a sending interlock, because the upper layer queueing delay of the parent task is not related to this sending interlock. Therefore, $\text{Pr}(\text{IL})$ is set to $\frac{1}{n_p \times n_c}$ for both parent and child client chains, in which n_p and n_c are the multiplicities of the parent task and child task respectively. Although the same rate is used by the previous solution, the new solution does not overestimate the interlocking effect. The reason is that in the new solution, $\text{Pr}(\text{IL})$, is applied to the queue length with one customer removed ($L_{mk}(\mathbf{N} - \mathbf{e}_j)$) rather than at the queue length at the full population. Therefore, the flow being removed in the new solution is smaller than the one by the previous solution.

The new interlock solution corrects both *Type1* and *Type2* interlocks. The results from solving the model in Figure 3.8 by the improved solution are shown in Figure 3.10. All the error rates (against to simulation) are less than 10%, which are more accurate than the previous solution, shown in Figure 3.9.

Table 3.5 shows the results from solving the model in Figure 3.8 with randomized values for the number of customers n of task s_0 , service demands of tasks s_0 , s_1 and s_2 , and the number of visits to s_1 and s_2 from s_0 . All of the queueing delays are improved between 50-90% when compared to the previous solution. The error rates of tasks' throughput (around 1.6%) are slightly higher than those by the previous

Table 3.5: Results of the Figure 3.8 model with randomized parameters.

$\% \Delta$	λ_{s0}	λ_{s1}	λ_{s2}	U_{s2}	$W_{e0,e1}$	$W_{e1,e2}$	$W_{e0,e2}$
Method: None							
mean	-6.10	-6.06	-6.03	-6.02	10.16	9.67	11.62
sdev	2.83	2.80	2.80	2.84	6.84	24.06	14.13
Method: lqns							
mean	0.87	0.92	0.95	0.96	5.06	-19.85	-21.73
sdev	1.24	1.18	1.19	1.21	4.25	9.78	8.95
Method: New							
mean	1.55	1.60	1.62	1.64	-3.07	-7.25	1.09
sdev	1.01	0.96	0.97	1.01	1.78	4.57	5.21

Figure 3.11: A LQN model of a sending interlock with an external contention.

solution (around 0.9%).

3.3.2.2 Sending Interlocks with External Contentions

The model in Figure 3.11 is a sending interlock with an external contention (issued by task $s2$). The results in Figure 3.12 are produced by the previous solution with corrections (lqns) and without any interlocking corrections (none). These results show that:

1. Without any interlocking corrections, there are large errors of throughput (in Figure 3.12c) and queueing delays (in Figure 3.12a) among the interlocked flows.

2. Using the previous solution, large errors appear on the side of non-interlocked flows (the external contentions), due to the same reason explained in the previous section. For example, the throughput is shown in Figure 3.12c and queueing delays in Figure 3.12b) of the task s_3 .

With the existence of external contention, the impacts of sending interlocks to their irrelevant chains are hidden more deeply and are easy to be neglected. Since parents and children double the amount of flow to the interlocked server, it makes the customers from other client chains (irrelevant to both the client chain of the parent and the child) see more customers than they are supposed to see, as the consequence. The new solution treats the service demands of all children (appeared in the same submodel) as the surrogate delay at the parent. This surrogate delay consumes its time at its own processor. Therefore, the flow from the parent does not contain the flows of its children anymore. Sending interlocks can be corrected using the same method with split interlocks.

The new interlock solution corrects queueing delays on both interlocked flow and non-interlocked flow. The results are shown in Figure 3.13. The error rates of all the performance results are less than 10%.

Table 3.6 shows the results of the model in Figure 3.11 with randomized values on the number of customers n of task s_0 , service demands of tasks s_0 , s_1 and s_2 , and number of visits to s_1 and s_2 from s_0 . All the results are much better than the previous solution, with improvements between 2-10 times. The error rates of the tasks' throughput are less than 2% and for the queueing delays are less than 6%.

3.3.2.3 Type3 Sending Interlocks

The model in Figure 3.14 is an example of *Type3* sending interlock and combines other forms of interlocking, such as a split interlock, a multi-layered sending interlock, and external contentions.

(a) Queueing delays (without interlocking corrections)

(b) Queueing delays (with corrections)

(c) Throughput (both with and without corrections)

Figure 3.12: Results with the previous interlock corrections and without corrections for the Figure 3.11 model.

Figure 3.13: Queueing delays and throughput (by the new solution) for the model in Figure 3.11.

Table 3.6: Results of the Figure 3.11 model with randomized parameters.

$\% \Delta$	λ_{s0}	λ_{s1}	λ_{s2}	λ_{s3}	U_{proc}	$W_{e0,e1}$	$W_{e1,e2}$	$W_{e0,e2}$	$W_{e3,e2}$
Method none									
mean	-8.33	-8.36	-1.12	5.92	-1.09	15.41	12.04	14.11	-10.35
sdev	2.77	2.77	1.77	2.23	1.77	8.72	19.60	17.74	2.52
Method lqns									
mean	-1.03	-1.06	8.57	18.14	8.60	10.17	-7.94	-9.72	-39.38
sdev	2.05	2.05	4.03	5.31	4.04	5.12	5.11	6.04	19.40
Method new									
mean	-0.18	-0.21	2.11	3.78	2.14	-2.73	1.75	5.87	-6.16
sdev	2.11	2.11	0.77	2.10	0.77	2.22	5.94	4.72	3.89

Figure 3.14: A LQN model of a type3 sending interlock.

The parent task has multiple entries, and some of the entries do not directly request service from the child task server but may also be blocked by the child task. *Type3* sending interlocking is significant when the parent task has only a single thread. Like *Type1* interlocks, *Type3* sending interlocking effects decrease rapidly as the number of parent tasks increases. For the model in Figure 3.14, if the number of customers of the parent task *s1* increases to two, *s1* is a *Type1* sending interlock, also, *s1* may become a sub-chain dependent client task.

To recognize *Type3* interlocks inside the LQNS solver is essential to solve *Type3* sending interlocks. Since the *PathFinder* [72] does not consider *Type3* as an interlock, the *PathFinder* in the improved solution is also modified to recognize *Type3* sending interlocks. As a result, the interlocking relationship ($ir_{i,j}$) between chains i and j , which correspond to the entries *s0* and *e2*, becomes 1. The interlocking probability and interlocking ratio are also set to 1.

Table 3.7 shows the results by using the improved solution. The RMS error rates of

Table 3.7: Results of the Figure 3.14 model.

RMS error	none	lqns	new
Waiting	24.53	19.26	11.30
Service	12.86	6.60	2.15
Throughput	13.96	21.85	4.63
Proc. Util.	3.36	8.66	1.17
Proc. Wait.	33.87	22.21	15.07

throughput and utilization are reduced 3~4 times. The RMS error rates of the mean service times (of entries) are reduced 3~5 times. The RMS error rates of queueing delays are reduced 0.5~2 times.

3.4 Interlocking Effects on Second-Phase Servers

This section focuses on how to handle the interlocks that appear in models with two-phase servers and clients.

3.4.1 Second-Phase Technique

The second-phase technique also refers to the second-phase of service [221], aggressive reply or early reply [79, 80]. Second-phase service is the deferred service, which is performed after the reply is sent back to the client. For example, during database write operations, some of the disk operations can be performed after the log file has been written.

In the second-phase technique, *Overtaking* [79] is when a request arrives, the server is executing the phase two service of the previous request. The overtaking event is shown in Figure 3.15a while Figure 3.15b is for a non-overtaking event.

3.4.1.1 Performance Advantages of Second-Phase Technique

The send-receive-reply pattern [79] (using single phase service), also known as a synchronous request, is a major pattern of inter-process communications and commonly

Figure 3.15: Second-phase server events.

used in systems with a client-server architecture. The second-phase technique uses early replies to reduce the overall blocking duration to clients. Therefore, second-phase services increase concurrency and reduce blocking time. By how much the second-phase technique can improve the response time of a synchronous request depends on the following three factors: [79].

1. The ratio of the second-phase service against the total service demand, α . A larger ratio of the second-phase service has less blocking than a smaller ratio. If the ratio is zero, requests are served as synchronous requests while if the ratio is one, requests are served asynchronously without any blocking to senders.
2. The server utilization. Early replies may have the most improvement in terms of response time on lightly loaded servers, but almost no effect with heavily loaded servers.
3. The number of clients. A large number of clients reduce the effect of second-phase services.

Figure 3.16: A three-tiered LQN model with a two-phase interlocked server.

3.4.2 Interlocking Effects

From the discussions of previous sections, the previous LQNS solution is over-estimating interlocking effects by over reducing the amount of impossible flow at the lower level, especially when there are only two customers in the system. Nevertheless, LQNS presents accurate estimation on client throughput and server utilization. Without the interlocking correction, LQNS overestimates queueing delays and underestimates server utilization.

These interlocking effects also appear in LQN models with second phase services. Because of the nature of second-phase services, the number of customers (or tokens) moving around in a system increases with the presence of second-phase servers. Therefore, in general, the second-phase technique reduces the amount of interlocking effect.

In order to figure out the impacts of second-phase services on interlocks, a simplified three-tiered model, shown in Figure 3.16, is used to enumerate the interlocking effects related to second-phase services. In the figure, the two intermediate servers $s0$ and $s1$ both make requests to the common interlocked server $t1$, and both tasks $s0$ and $s1$ run on their own processors. Multiple executions of this model were performed with the existence of second-phases service on the client, the intermediate servers or

Figure 3.17: Results by the previous solution for the multi-phase client model.

the interlocked server, in order to study the interlocking impacts on any of these tasks.

3.4.2.1 Multi-Phase Clients

In this case, only the client c_0 , in Figure 3.16, has a second-phase service. In the experiments, the model varied the number of customers from one to seven and was solved by the previous LQNS solution with and without interlock corrections. Figure 3.17 shows the results of two solutions, in which are labelled as 'lqns' and 'none' respectively. In these figures, the largest error rates occurred when there are two customers. Similar to the models with a single-phase service only, LQNS overestimates the interlocking effects and results in the underestimated queueing delays at the interlocked server. The error rates at the interlocked server are around -15~-25%. But LQNS produces reasonably accurate results of server utilization and client throughput, and their error rates are less than 5%, shown in Figure 3.17b. While without interlocking corrections, the error rates of throughput and utilization are up to -20% in the cases of one and two customers. As the number of customers at the common source client increases, the interlocking effect decreases rapidly.

Therefore, the results of the experiments show that:

Figure 3.18: Results by the previous solution for the Figure 3.16 model.

1. A multi-phase common source client does neither worsen nor improve interlocking effects compared to single-phase models, like the model in Figure 3.1a.
2. The ratio of the second-phase service at the client does not affect the interlocking probability.
3. The severe interlocking effects still occur when there are only two customers in the system because the previous LQNS solution handles interlocks with one customer at the common source perfectly.

3.4.2.2 Multi-Phase Interlocked Server

Since a second-phase server actually increases the number of tokens concurrently executing in the system, the two-phase interlocked server dilutes the interlocking effects, which is indicated by the results without interlocking corrections, shown in Figure 3.18. However, the case with only one customer from the common source needs to be reconsidered, since a phase-one request may have to wait for the completion of either its own phase-two service or a second-phase request from the other client chain.

Figure 3.18 shown the results solved by both the previous solutions. They are

the two cases that have the most severe interlocking effects, with one customer (Figure 3.18a) and two customers (Figure 3.18b) in the system. These results show that:

1. The previous correction slightly underestimated the queueing delay at the interlocked server, even in the case of one customer.
2. The ratio of the second-phase service, α has an observable influence on the interlocking effects. As the ratio α decreases, the error rates for both solutions increase because the interlocked flow dominates the entire flow.
3. The upper-level queueing delay does not have significant errors.

3.4.2.3 Multi-Phase Intermediate Server

The third case is where only the intermediate servers, s_0 and s_1 in Figure 3.16, have second-phase services. In LQN, any phase within an entry is able to make requests to some lower level server. Thus, the interlocked server receives flows (including both phase-one and phase-two requests) from the intermediate tasks. Phase-one requests are interlocked as in the case of single-phase models. Phase-two requests, on the other hand, have fewer dependencies with others, since they are extra customers in addition to those from the common source.

The dependencies rely on the second-phase service demand required by an intermediate task (and excludes the included services), as well as the ratio of second-phase services (and includes the nested services). The first factor reduces the dependency between a phase-one request and its successive phase two request, which makes the phase-two request behave more randomly. The second factor affects the total amount of the impossible flow since the impossible flows hidden in the phase two requests are smaller than those in the phase-one requests.

Figures 3.19 show the results found by both the previous solutions for the case of one customer (3.19a, and 3.19b) and the case of two customers (3.19c and 3.19b). The

Figure 3.19: Results of the previous solution for the multi-phase intermediate tasks model with one and two customers.

results show that the ratio of phase-two services has a considerable influence to the interlocking effects. As the phase-two ratio decreases, the error rates grow larger. The reason is that, in the previous LQNS solution, the interlocked rates are considered as $\frac{1}{n+1}$ on requests from any phase, in which n is the number of customers. The reason for $(n+1)$ is that the second-phase client increases the number of customers. However, as the ratio decreases, the interlocked flows are gradually dominating the entire flow which arrives at the interlocked server. Thus using the fraction of $\frac{1}{n+1}$ as the interlocking probability produces larger errors when the ratio is smaller.

In general, in both of cases (multi-phase interlocked server and intermediate servers), the queueing delays at an interlocked server are underestimated. One-customer models once again are back to attention since there are additional customers appeared internally in the systems. Now, models with only one customer can produce errors since there an addition customer instead to the system due to phase-two.

3.4.3 Improvements

The main reason of underestimation of the queueing delays at the interlocked server is that the previous solution used the sample ratio $\frac{1}{n}$ and $\frac{1}{n+1}$ (n is the number of customers from the common source) as interlocking probabilities. The ratio $\frac{1}{n}$ is for the interlocks with single-phase service while $\frac{1}{n+1}$ is for multi-phase models. These interlocking corrections had some effects in reducing interlocking effects, but cannot reflect the impacts of the ratio of second-phase services.

For the models only with a multi-phase common source client, since second-phase requests from the common source, do not have any influence on interlocking effects, the interlocking probability $\text{Pr}(\text{IL})$ is found by using the equation (3.4) in Section 3.2.1.1. Therefore, this section focuses on finding suitable interlocking probabilities $\text{Pr}(\text{IL})$ for the other two cases with second-phase services in order to improve the solution.

3.4.3.1 Multi-Phase Interlocked Server

At a multi-phase server, the mean residence time of requests is found by (3.21) from [79].

$$W_m = s_{m1} + \Pr\{\text{OT}\}_m s_{m2} + s_m L_m(N-1) + (1 - \Pr\{\text{OT}\}_m) U_{m2}(N-1) \quad (3.21)$$

In this equation, the first term is the mean service demand required by the phase-one service itself. The second term is for the mean queueing delay caused by waiting for the completion of the previous phase-two service (it is executing). This is a case of an overtaking; $\Pr\{\text{OT}\}_m$ is the probability of an overtaking occurring. The third term is the queueing delay because of the blocking by other servers. The last part is the correction of the third term because of overtaking. The details of this equation can be found in [79].

If interlocked flows exist and arrive at the multi-phase server m , the queue lengths, $L_{mk}(\mathbf{N} - \mathbf{e}_j)$, in the third term need to be adjusted by multiplying the interlocking probability $\Pr(\text{IL})$ based on phases.

Since a multi-phase interlocked server dilutes interlocking effects, the interlock probabilities of phase-one requests decrease in a much faster speed compared to single-phase models, as the number of customers from the common source increases. In the experiments, squares of the corresponding interlocking probabilities in single-phase model, $\Pr(\text{IL})_s$, are found as a good approximation of $\Pr(\text{IL})_{ph1}$, which denotes the interlocking probability of phase-one requests. Therefore, $\Pr(\text{IL})_{ph1}$ is used to adjust $L_{mk}(\mathbf{N} - \mathbf{e}_j)$ for the flows arriving at the phase-one server and is estimated using (3.22), where $\Pr(\text{IL})_s$ is estimated by (3.4) in Section 3.2.1.1.

$$\Pr(\text{IL})_{ph1} = \Pr(\text{IL})_s^2 \quad (3.22)$$

During the execution at a phase-two server, there is an extra token appeared in the system. The interlocking probabilities for the flows arrived at the phase-two server are smaller than those at the phase-one server. Therefore, using the similar idea of the previous solution, the interlocking probability at a phase-two server, $\Pr(\text{IL})_{ph2}$ is estimated by (3.23).

$$\Pr(\text{IL})_{ph2} = \frac{1}{n+1} \Pr(\text{IL})_{ph1} \quad (3.23)$$

On the other hand, the importance and necessity of the upper-level adjustments decrease. From the discussions in Section 3.2.2, the purpose of the upper-level adjustment is removing some part of queueing delay in the included (nested) services (in terms of MRL) that are invisible to the upper-level requests. With the existence of second-phase services, the queueing delay at a multi-phase interlocked server consists of the waiting for the completion of its previous phase-two service and the blocking by the requests from other client chains, either in phase-one or phase-two service. Among these delays, only the blocking caused by the phase-one services may be partially visible or completely invisible, and the probabilities of the invisible nested queueing delays are the probability that all the customers arrive at the same intermediate server task, which is explained in Section 3.2.2. Consequently, the fraction of the interlocked queueing delay, which is not visible to the upper-level requests, becomes very small comparing with the ones in single-phase models.

3.4.3.2 Multi-Phase Intermediate Servers

The key point for handling interlocks, in this case, also is differentiating the interlocking probabilities of phase-one requests and phase-two requests arriving at the interlocked server. However, when solving the LQN submodels, the flows from a multi-phase client are not distinguished in the MVA solutions. Eq.(3.24) is used to break down the queue length of a client chain based on phases, by using the ratio of

a phase-two service, α .

$$\sum_k^K S_{m,k} L_{mk}(\mathbf{N} - \mathbf{e}_j) = \sum_k^K ((1 - \alpha) S_{m,k} L_{mk}(\mathbf{N} - \mathbf{e}_j) + \alpha S_{m,k} L_{mk}(\mathbf{N} - \mathbf{e}_j)) \quad (3.24)$$

When a request arrives at the interlocked server, it sees requests from other client chains, either as phase-one requests (the first term in (3.24)) or as phase two requests (the second term in (3.24)). Therefore, the two terms in (3.24) need to be adjusted differently. First, the interlocking probabilities between a phase-one request with the incoming request can be found by using the same approximation, (3.4) in Section 3.2.1.1, for single phase models ($\text{Pr}(\text{IL})_s$). Since phase-two requests are generated by an additional customer, the interlocking probability between the phase-one and a phase-two request is estimated by (3.26), similarly, (3.27) is for two phase-two requests.

$$\text{Pr}(\text{IL})_{ph1,ph1} = \text{Pr}(\text{IL})_s \quad (3.25)$$

$$\text{Pr}(\text{IL})_{ph1,ph2} = \frac{1}{n+1} \text{Pr}(\text{IL})_{ph1,ph1} \quad (3.26)$$

$$\text{Pr}(\text{IL})_{ph2,ph2} = \frac{1}{n+2} \text{Pr}(\text{IL})_{ph1,ph2} \quad (3.27)$$

Then, by applying these interlocking probabilities $\text{Pr}(\text{IL})$ to (3.24), the total interlocking probabilities of the arriving phase-one and phase-two request are found using (3.28), which is used to replace the $\text{Pr}(\text{IL})$ in (3.7) when calculating the lower-level adjustment, $\mathbf{r}_{mk}^{\text{lower}}$.

$$\begin{aligned} \text{Pr}(\text{IL})_{ph1} &= (1 - \alpha) \text{Pr}(\text{IL})_{ph1,ph1} + \alpha \text{Pr}(\text{IL})_{ph1,ph2} \\ \text{Pr}(\text{IL})_{ph2} &= (1 - \alpha) \text{Pr}(\text{IL})_{ph1,ph2} + \alpha \text{Pr}(\text{IL})_{ph2,ph2} \end{aligned} \quad (3.28)$$

However, there is a situation that needs to be considered especially, which is when there is only one customer at the common source. Since phase-one requests from all

Figure 3.20: second-phase clients and interlocked servers.

Figure 3.21: second-phase clients and interlocked servers.

intermediate servers are completely interlocked, they do not see each other in the server queue. But a phase-one request may queue after a phase-two request which is being served currently. However, as for phase-two requests, there are different situations, which depend on the phase-two client service demand, s'_{m2} , required by an intermediate task. Here, the phase-two client service demand, s'_{m2} excludes its nested service's demand. These situations are shown in Figure 3.20 and 3.21.

In Figure 3.20, if s'_{m2} is large enough, and some phase-one requests arrive prior to the release of that phase-two request. Therefore, phase-two requests may see a phase-one request (at c in the figure) or another phase-two request (at d in the figure) in front of it. In this case, the interlocking effects on the queueing delay of phase-two requests can be corrected using (3.28).

On the other hand, if the phase-two service demand is small, then the probability

for other phase-one requests (from any other client chains) arriving at the server before the corresponding phase-two request is released, is also small, which is shown in Figure 3.21. This case means that phase-two requests have a little chance to wait for the completion of a phase-one request. One extreme example of this kind of behaviour is in LQN device submodels. Since all the service demands (excludes nested services) of a task, regardless of being phase-one or phase-two, eventually consume time at its device (a LQN processor). In device submodels, phase-two requests are always released before phase-one requests from other client chains since there is only one customer from the common source. Therefore, all the phase-two requests do not have any queueing delays at their processor. The probability of a phase-one request from a client chain k arriving at the server before the phase-two request of the client chain j is released is estimated by the fraction of the second-phase client service demand and the think time of the arriving phase-one customers.

3.4.4 Results

In this section, all the models in Section 3.4.2 are re-solved by using the improved solution. To verify the effectiveness of the improved solution, all the results were compared with the corresponding results found by the previous solutions with and without interlocking corrections. The comparisons of these three analytic solutions are against the simulation results (with a confidence interval of 95%) found by LQSIM.

3.4.4.1 Multi-Phase Interlocked Server

The model used in this section is the same model shown in Figure 3.16. The model was re-solved by the new solution by varying the number of customers at the common source client $c0$ from 1 to 10. The set of figures, Figure 3.22a $\sim$ 3.22c show the results. The maximum error rates of the queueing delays (at both a lower level and an upper level) by the new solution are around 5 $\sim$ 8%, which is only one-third of the error

Figure 3.22: Results from new solution and previous solutions for the Figure 3.16 model.

rates by the previous solutions.

Since the ratio of the second-phase services (α) has an influence on the interlock effects, the model was also re-solved by varying the ratio of the second-phase service α from 0 to 1 by a step of 0.1, with one customer and two customers only. Figure 3.23a $\sim$ 3.23c show the case of only one customer at the common source, and Figure 3.24a $\sim$ 3.24c, demonstrate the case of two customers. The results by the new solution in both cases are much better than the previous solutions. For the one customer case, results of the new solution are almost perfect.

Figure 3.23: Results from the new solution and the previous solutions for the Figure 3.16 model with one customer.

Figure 3.24: Results from the new solution and the previous solutions for the model in Figure 3.16 with two customers.

Figure 3.25: Results for the multi-phase intermediate server model by varying the number of customers.

3.4.4.2 Multi-Phase Intermediate Server

The model in Section 3.4.2.3 is re-solved by the new solution. Figures 3.25a ~ 3.25c show the results from changing the number of customers at the common client from 1 to 10. Figures 3.26a ~ 3.26f show the case of two customers and varying the ratio of the second-phase services.

3.4.4.3 Randomized Model with Second-Phase Services

First, the experiment model used in this section is a model where all the tasks have second-phase services, shown in Figure 3.27. This model can demonstrate the interlocking effects for combining the multi-phase client, intermediate server and inter-

Figure 3.26: Results comparison for the multi-phase intermediate server model by varying the ratio of the second-phase services at the populations of one customer and two customers.

Figure 3.27: A simplified LQN model with second-phase client and servers.

locked server. Again, the most severe cases of interlocking effects are when there are one customer and two customers at the common client. Figures 3.28a ~ 3.28f show the results of these two cases when compared to simulations, the figures on the left for the one customer case. These results show that the queueing delays at the interlocked server and intermediate servers are all improved.

Experiments were also performed on this model with one hundred sets of randomized parameters, on the number of customers at the common source, the service demands at intermediate task $s0$ and $s1$, the number of visits from the common source $c0$ to the intermediate tasks $s0$ and $s1$, and the ratio α . The average errors from solving the model using the previous approximations with and without interlocking corrections, and the new approach, are shown in Table 3.8. All the results by the improved solution are much better than the previous results.

3.5 Verification

This section presents two relatively more complex interlocking models to verify the improved solution (integrating all the modifications) discussed throughout this chapter. All the models are solved analytically by using LQNS without any interlock corrections, with the previous interlock adjustment, and the new interlock adjustments. All

Figure 3.28: Results comparison for the model in Figure 3.27

Table 3.8: Results from solving 100 different versions of the model in Figure 3.29.

Methods	$\% \Delta$	λ_{c0}	$S_{c0,ph1}$	$S_{c0,ph2}$	$W_{c0,s0,ph1}$	$W_{c0,s0,ph2}$
None	<i>mean</i>	12.16	-12.67	-11.80	-7.84	-5.73
	<i>sdev</i>	3.13	2.49	2.38	1.22	1.79
Lqns	<i>mean</i>	17.34	-17.29	-16.54	-15.86	-13.93
	<i>sdev</i>	2.67	1.71	1.56	1.01	0.59
New	<i>mean</i>	5.76	-6.92	-5.51	3.91	7.49
	<i>sdev</i>	2.02	1.98	1.79	1.02	0.98
Methods	$\% \Delta$	$S_{s0,ph1}$	$S_{s0,ph2}$	$W_{s0,e1,ph1}$	$W_{s0,e1,ph2}$	U_{e1}
None	<i>mean</i>	-11.65	-9.79	-19.13	-16.64	10.67
	<i>sdev</i>	3.25	2.61	4.61	3.77	2.08
Lqns	<i>mean</i>	-16.85	-14.59	-27.76	-25.00	15.63
	<i>sdev</i>	2.00	1.49	2.14	1.54	2.04
New	<i>mean</i>	-7.09	-6.37	-11.69	-10.78	5.20
	<i>sdev</i>	2.45	1.89	3.60	2.77	1.63

the results were compared by using their error rates, which are against simulations. Again, all the simulations were run with confidence intervals of $95\% \pm 1.0\%$.

3.5.1 The Combined Model

The model in Figure 3.2 (Section 3.1) combines most of the identified forms of interlocking effects but excludes *Type2* and *Type3* sending interlocks. The model is a system where a number of tasks ($s0$, $s1$, $s2$, $s3$, $s4$ and $c4$) are executed on a processor ($p0$). In which tasks $s4$ and $c4$ are invoked internally, are background services. Task $c4$ has included service $s3$. The tasks $s0$, $s1$ and $s2$ are invoked by the client tasks $c0$, $c1$ and $c2$, which are executed on their own processor.

Table 3.9 shows the mean RMS error rates (compared to LQSIM results) of the new solution and the previous solution with and without corrections. The mean RMS error rates of the improved solution are around 1-2%, except call waiting. The RMS error rates of call waiting are reduced by 40%.

Table 3.9: Results of the Figure 3.2 model.

RMS error	none	lqns	new
Call Waiting	15.85	14.04	8.19
Entry Service	5.75	8.03	2.39
Throughput	7.36	10.38	1.09
Processor Util	4.95	8.63	0.90
Processor Waiting	5.94	9.36	1.16

Figure 3.29: A Complex Model with Interlocking Effects.

3.5.2 Deeply Layered Interlocking Model

This model in Figure 3.29 appeared in [72]. It is a synthetic model which emphasizes a complicated case of split interlocking impacts. The difference between this model and the previous model in Figure 3.2 is that this model is deeply layered, which complicates the process of recognizing interlocking relationships and the calculation of interlocking probabilities.

Table 3.10, shows the mean RMS error rates (compared to LQSIM results) of the new solution and the previous solution with and without corrections. All the RMS

Table 3.10: Results of the Figure 3.29 model.

RMS error	none	lqns	new
Call Waiting	34.03	26.55	7.19
Entry Service	12.81	8.82	3.83
Throughput	12.02	8.58	4.04
Processor Util.	12.04	8.58	4.11

error rates are reduced by 50~80% by the improved solution.

Table 3.11 shows the execution time of solving these two models using the previous solver (with and without corrections) and the improved solution. The results show that the improved solution did not increase the execution time significantly.

Table 3.11: The execution time of two models in this section.

Model	Execution Time (S)		
	None	Previous	New
The combined model (in Section 3.5.1)	0.028	0.031	0.032
The deeply layered interlocking model (in Section 3.5.2)	0.022	0.024	0.026

3.6 Summary

This chapter has shown that solving a Layered Queueing Network using the method of decomposition can lead to large errors if the effect of interlocking is not considered. The previous solution of the Layered Queueing Network Solver has handled some cases, but also can suffer from large errors due to the following:

1. The distribution of the waiting time from contention at the interlocked server in the lowest submodel was not distributed properly among the layers of the input model.
2. The calculation of $\text{Pr}(\text{IL})$ was naïve in that it was a simple ratio based on the number of customers in an interlocked path.

3. The interlock adjustment did not properly account for *external contention*.

The new approximation presented in this chapter reduces the amount of error in each of these three cases.

1. The interlock adjustments are performed both at the lowest level submodel and the upper level submodels.
2. The new approximation corrects for interlock when there is one less customer j in the queue ($L_{mk}(\mathbf{N} - \mathbf{e}_j)$) rather than at the full customer population ($L_{mk}(\mathbf{N})$).
3. The new approximation takes into account of the impacts of external contentions and mixed flows.
4. The new approach estimates the interlock probability ($\text{Pr}(\text{IL})$) far more accurately than the previous approach used by LQNS.

In this section, addition to improving some cases, such as split interlocks and sending interlocks, which are handled previously, some interlocking effects cases are newly identified and handled in the improved solution. Furthermore, the impacts of the second-phase services on the interlocking effects were also reduced.

All the models presented in this chapter are all severely affected by interlocking dependencies. All the identified interlocking effects are covered (enumerated) by these models. The results from solving these models using the new algorithm are always superior to the results found using the previous approach. Furthermore, the new calculation does not produce infeasible utilizations, unlike the previous approach.

The improved solution was tested by regression tests in order to prevent the occurrence of undesired side effects. Furthermore, the improved solution does not adversely affect the efficiency of the LQN analytic solution.

Chapter 4

Sub-Chain Dependencies

This chapter describes the phenomenon of sub-chain dependencies in which traffic from multiple independent clients share a common intermediate server then diverges to lower level servers. The existing LQN solution often gets the queueing delays at the servers in the model wrong. The problem is corrected in this work by limiting the maximum possible number of customers issued by each client sub-chain in the MVA solutions.

4.1 Sub-Chain Dependencies

In many systems, multi-threaded tasks are used to run services concurrently, to improve the responsiveness of systems by reducing clients' response time. A multi-threaded server task may serve customers from different independent sources, such as regular user requests or administration operation requests. These services can be implemented by separate functions within a task or module, rather than by individual tasks or modules. In LQN, these services are modelled as entries within a server task. Administration operations, such as book-keeping or backups, generally only have a few customers, which are much less than the regular system users.

Sub-chain dependent problems arise when solving an LQN model using the method

Figure 4.1: A LQN model with a sub-chain dependency.

of decomposition when the number of customers from one source is less than the number of server threads, while the number of customers from another source is not. Sub-chain dependencies occur in two situations: one is when the administration operations make further requests to some other server tasks to complete the entire process; the other situation is in the device submodels of multi-threaded server tasks with multiple services.

For example, Figure 4.1 shows a LQN model with a sub-chain dependency in which two customer classes, $c0$ and $c1$ are interacting with a common intermediate server $t0$ which, in turn, makes requests to two lower-level server tasks $t1$ and $t2$. Client $c0$ only introduce one customer into the system, while there might be a maximum of five customers from client $c1$ in the system simultaneously. The intermediate multi-server task $t0$ has two entries which segregate the calls from $c0$ and $c1$. The number of clients that are making requests to these two entries in `Submodel1` are imbalanced.

4.1.1 Impacts of Sub-Chain Dependencies

Sub-chain dependency occurs when customers from independent routing chains at upper submodels are blended at a common entity when this entity is acting as a source of customers to lower layer submodels. Without accounting for this problem, significant errors can appear, especially when the number of clients from different

Figure 4.2: Submodels from Figure 4.1.

sub-chains are imbalanced.

Figure 4.2 shows why a sub-chain dependency occurs by showing the two submodels used to solve the overall model using the method of decomposition. Task $t0$ is acting as a server in **Submodel1** (Figure 4.2a) and as a client in **Submodel2** (Figure 4.2b). When $t0$ is acting as a client, it is represented by exactly one chain in the corresponding queueing model. Although the number of threads of $t0$ is six (five for entry $e0b$, and one for entry $e0a$), there can only be exactly one customer making a request from entry $e0a$ to the entry $e1$ (task $t1$). The reason is that entry $e0a$ only accepts requests from the reference task $c0$ in the **Submodel1**, which can only ever have one customer. Therefore, in **Submodel2**, the requests from $c0$ ‘bleed into’ the call from $e0a$ to $e1$ due to the common routing chain.

This phenomenon has a significant impact on the accuracy of the LQN analytic solution especially with models like the model in Figure 4.1 when the “lightly-loaded” entry ($e0a$) of the common routing chain only has one customer. Figure 4.3 shows the results from solving the model in Figure 4.1 using the current analytic solution (LQNS) and simulation. The number of customers from client $c1$ varies from 1 to 10 while the number of customers from client $c0$ is fixed at 1. The problems that exist in LQNS indicated by the results are as follows.

1. The queueing delay of the entry $e0a$ at the server task $t1$, which is labelled as $e0a(old)$ in Figure 4.3a, is between 0.2~0.15 from using the analytic solution while they are zeros as found by simulation ($e0a(sim)$ is overlapped with the

Figure 4.3: Results from solving the model in Figure 4.1.

X axis in the figure). So, this queueing delay is completely extraneous.

2. The queueing delay of the lightly-loaded client entry $e0a$ at the processor $p0$ is problematic too. LQNS produced the same queueing delays (with values between 4.90~13.76 in Figure 4.3b) for both entries $e0a$ and $e0b$ at processor $p0$. For entry $e0b$, these results are quite accurate. However, for entry $e0a$, the difference between LQNS and simulation (with values only between 0.365~4.80) are very significant. The error rates of the mean queueing delay of entry $e0a$ at the processor ($\frac{\text{sim}-\text{lqns}}{\text{sim}} \times 100$) are greater than 200%, (shown in Figure 4.3d on the left Y axis). The error rate is inversely proportional to the number of customers at the lightly-loaded entry.
3. The throughput of the top level client task $c0$, which is related to the sub-chain of entry $e0a$, also has rather large errors, shown in Figure 4.3c. Its error rates are around 40%, shown in Figure 4.3d using the right Y axis.

The reason that the current LQN analytic solver is failing to handle this case is that the current solver is using the multiplicity of the sub-chain dependent client task as N_k , for the number of customers in chain k in the population vector ($\mathbf{N}$), during the MVA solution. The fraction of $\frac{N_k-1}{N_k}$ is used to estimate the mean queue length with one customer removed from chain k , $L(\mathbf{N} - \mathbf{e}_k)$, in (2.4), based on Bard-Schweitzer AMVA [15, 186]. Bard-Schweitzer AMVA assumes that the mean queueing delay is proportion to the total mean queue length with one customer removed. The number of customers N_k from a client chain k has a direct impact on the queueing delay. To correct the number of customers, especially for lightly-loaded sub-chains, the maximum number of customers of an entry, denoted as MC , is used in the approximate MVA solutions and is explained in the next section.

Using MC instead of N_k , the extraneous queueing delay can be eliminated, since fewer customers are waiting in a queue before they are served. The extraneous queue-

ing delay of a lightly-loaded sub-chain can be estimated by a ratio of queue lengths, $(\frac{Q_{lqns} - Q_{new}}{Q_{lqns}})$. Since the queue length of one customer removed is estimated by using $\frac{N_k - 1}{N_k} L(\mathbf{N})$ previously, and by $\frac{MC - 1}{MC} L(\mathbf{N})$ in the new solution. The ratio $\mathbf{r}$ is the fraction of the extraneous queueing delay among the total queueing delay produced by the current LQNS, and it is estimated by (4.1).

$$\mathbf{r} \approx \frac{(1 - \frac{1}{N_k}) - (1 - \frac{1}{MC})}{1 - \frac{1}{N_k}} = \frac{N_k - MC}{(N_k - 1) \times MC}; (N_k > 1) \quad (4.1)$$

Equation (4.1) can be used only when the number of customers from a client chain is more than one customer, $(N_k > 1)$. If there is only one customer from that client chain, then no queueing delay occurs, since there is no contention among the requests from that particular client chain. When MC is close to N_k , the difference between $(\frac{1}{MC} - \frac{1}{N_k})$ in (4.1) is small, so the current solver can produce results with good accuracy. However, when MC is much smaller than N_k , the ratio $\mathbf{r}$ becomes inversely proportional to MC . The latter case indicates that the queueing delay produced by the current LQNS solver contains a significant portion of the extraneous queueing delays in this case. When MC is one, the ratio of the extraneous queueing delay is one too.

4.2 Improved Chain Calculation

In this section, the maximum possible number of customers at an entry, MC , is introduced into the approximate MVA solution as the population of a sub-chain, to handle sub-chain dependencies.

4.2.1 The Maximum Number of Customers

In order to correct the number of customers that arrive from a sub-chain, the maximum possible number of customers of an entry is used to replace its multiplicity in

the analytic solver. The $MC_{c,e}$ of any client entry e and $MC_{s,e'}$ of any server entry e' are constants among solution iterations. Therefore, they need to be calculated only once during the entire solution.

In the submodels, the maximum possible customers MC of a client entry is found by (4.2), while (4.3) and (4.4) are used for server entries.

$$MC_{c,e} = \begin{cases} \min(MC_{s,e}, n_i) & \text{For a normal client entry } e; \\ n_i & \text{For a reference client task } i \text{ entry } e; \end{cases} \quad (4.2)$$

$$MC_{s,e,i} = \min\left(\sum_{e' \in \mathcal{P}} MC_{c,e'}, n_i\right); \quad (4.3)$$

$$MC_{s,e} = \sum_{e' \in \mathcal{Q}} MC_{s,e,i}; \quad (4.4)$$

where:

$MC_{c,e}$: the maximum number of customers of a client entry e ;

n_i : the multiplicity of the owner task i ;

$MC_{s,e}$: the maximum number of customers that can come to a server entry e from all sources;

$MC_{s,e,i}$: the maximum number of customers of a server entry e from a client task i ;

$\mathcal{P}$: is the set of entries that issue requests to the server entry e and owned by a client task i ;

$\mathcal{Q}$: is the set of client tasks that issue requests to the server entry e .

If a client entry e belongs to a reference task i , its $MC_{c,e}$ is the multiplicity n_i of its owner task i . Otherwise, for a normal client entry e , (both acting as a server entry in a submodel and as clients in some lower layer submodels), its $MC_{c,e}$ is the minimum number between the multiplicity n_i of its owner client task i and $MC_{s,e}$ of the very same entry e when it acts as a server.

For a server entry e , $MC_{s,e,i}$ is the total number of customers from a client task i that can arrive at this server entry e . So, $MC_{s,e,i}$ includes all customers from all the

client entries in the set of $\mathcal{P}$ to the server entry e . The maximum value of $MC_{s,e,i}$ is the multiplicity n_i of the client task i , shown in (4.3). Finally, the maximum possible customers of the server entry e , $MC_{s,e}$ is the sum of $MC_{s,e,i}$ from all client tasks in the set of $\mathcal{Q}$ by (4.4).

Finding $MC_{c,e}$ is trivial after $MC_{s,e}$ is found. Algorithm 1 is used for finding the total number of possible customers to every server entry owned by a server task or processor (**aServer**). Table 4.1 shows the MCs of every client and server entry appearing in the model in Figure 4.4, found by Algorithm 1. For example, the server entry $s1_e2$ in Figure 4.4 accepts requests from client tasks $c0$ and $s0$. From task $s0$, client entries $s0_e1$ and $s0_e2$ both issue requests to the server entry $s1_e2$. $MC_{c,s0_e2}$ of the client entry $s0_e2$ is one because it accepts requests from the reference task $c1$ with only ONE customer all the time. $MC_{c,s0_e1}$ of the **client** entry $s0_e1$ is three since task $s0$ is not able to generate more than three customers although the **server** entry $s0_e1$ has four customers when it is a server. Consequently, the server entry $s1_e2$ has eight customers at most, in which three customers are from the client task $s0$, and the other five customers are from the client task $c0$. On the other hand, server entries $s0_e2$, $s1_e3$ and $s3_e1$ have only ONE customer at most.

4.2.2 Applying MC to Approximate MVA Solutions

After obtaining MC for every client and server entry in all the LQN submodels, the maximum number of customers $MC_{s,e,i}$, at a server entry e from a client task i , is passed into the corresponding queueing model and is known as $C_{m,se,k}$. In MVA solutions, $C_{m,se,k}$ denotes the maximum number of customers from a client chain k to a server entry se owned by a server station m , and was set by Algorithm 1 (*line 17*).

Algorithm 1 Finding the maximum customers of $aServer$

```

1: ServerStation  $aStation = aServer.serverStation()$ ;
2: for each  $se$  in  $aServer.entries()$  do
3:   unsigned totalCustomers=0;
4:   for each  $task_j$  in  $clients()$  do
5:     unsigned total=0;
6:     if  $aStation.hasServerChain(task_j)$  then
7:       unsigned  $k=task_j.getChain()$ ;
8:       for each client entry  $ce$  of  $task_j$  do
9:         if  $se$  is called by  $ce$  then
10:          total+=  $ce.getMaxCustomer()$ ;
11:        end if
12:      end for
13:      if total>  $task_j.population()$  then
14:        total=  $task_j.population()$  ;
15:      end if
16:      totalCustomers += total ;
17:       $aStation.setMaxCustomer(se, k, total)$ ;
18:    end if
19:  end for
20:   $se.setMaxCustomer( totalCustomers )$  ;
21: end for

```

Figure 4.4: An example model for MC calculations.

Table 4.1: MC s of all client and server entries in the model in Figure 4.4.

Maxmium number of Customers		
Entries	Client Entry ($MC_{c,e}$)	Server Entry ($MC_{s,e}$)
c0	5	n/a
c1	1	n/a
c2	4	n/a
c3	1	n/a
s0_e1	3	4
s0_e2	1	1
s1_e1	2	5
s1_e2	2	8
s1_e3	1	1
s2_e1	1	2
s3_e1	1	1
s4_e1	1	7
p0_se1*	n/a	3
p0_se2*	n/a	1
p1_se1*	n/a	2
p1_se2*	n/a	2
p1_se3*	n/a	1

4.2.2.1 Applying MC to Bard-Schweitzer AMVA Solutions

Based on the Arrival Theorem [189], when a customer arrives at a server station m , it sees customers from its own client chain j , and from all other client chains. The mean number of these customers is the mean queue length with one customer removed from chain j , $L_{m,k}(\mathbf{N} - \mathbf{e}_j)$ ($1 \leq k, j \leq K$), in Bard-Schweitzer AMVA [15, 186]. The estimation of $L_{m,k}(\mathbf{N} - \mathbf{e}_j)$ is based on the assumption that $L_{m,k}(\mathbf{N} - \mathbf{e}_j)$ and $L_{m,k}(N)$ are proportional to the number of customers.

The improvements to the Bard-Schweitzer AMVA solution by applying MC are:

1. $C_{m,se,k}$ is used to replace N_k in the population vector $\mathbf{N}$ where $C_{m,se,k}$ and N_k are different;
2. Distinguishing the sub-chain that has the incoming customer from the other sub-chains owned by the client chain j , which is the client chain having the

incoming customer;

3. The calculation of queue lengths with one customer removed, $L_{m,e,k}(\mathbf{N} - \mathbf{e}_j)$ is now done at the sub-chain level. So the customers from a particular sub-chain (with fewer customers) can be differentiated.

In the case of sub-chain dependencies, if the client chain j with the incoming customer has sub-chains, the arriving customer at a server entry $se1$ sees:

1. at most $C_{m,se1,j} - 1$ customers from its own sub-chain;
2. *either* at most $C_{m,se2,j} - 1$ customers from another sub-chain if $C_{m,se2,j} = N_j$.
3. *or* at most $C_{m,se2,j}$ customers from another sub-chain, if $C_{m,se2,j} < N_j$;
4. at most N_k customers from any other client chain k ($k \neq j$).

Based on these observations, the estimation of the mean queue length with one customer less at a sub-chain level, $L_{m,e,k}(\mathbf{N} - \mathbf{e}_j, je)$ is found by (4.5). $L_{m,e,k}(\mathbf{N} - \mathbf{e}_j, je)$ denotes a queue length with one customer less from the client chain k at the server entry e (owned by a station m) when the incoming customer is from the sub-chain js (of the client chain j) arriving at the server entry je .

$$L_{m,e,k}(\mathbf{N} - \mathbf{e}_j, je) = \begin{cases} \frac{C_{m,e,k}-1}{C_{m,e,k}} L_{m,e,k}(N); & (k = j) \text{ and } (e = je) \\ \frac{C_{m,e,k}-1}{C_{m,e,k}} L_{m,se,k}(N); & (k = j), (e \neq je) \text{ and } (C_{m,e,k} = N_k) \\ L_{m,e,k}(N); & (k = j), (e \neq je) \text{ and } (C_{m,e,k} < N_k) \\ L_{m,e,k}(N); & (k \neq j) \end{cases} \quad (4.5)$$

where: j is the chain with incoming customer;

je is the server entry who accepts the incoming customer from the sub-chain js .

In order to demonstrate how the estimation using 4.5 improves the MVA solution compared to the previous LQNS, two sub-chain dependent cases are presented. Figure 4.5 shows the submodels of these two cases. The sub-chain dependent client $t0$

Figure 4.5: Two types of sub-chain dependent submodels.

in both the submodels in the figure have *THREE* customers, and *MCs* of sub-chains are given as 1 and 2 for **Type 1** and 1, 2, and 3 for **Type 2** respectively.

Case 1: Server entry *se1* is the only entry of the server station *t1* accepting requests from the client chain *t0*. When a customer from chain *j* arrives at *se1*, it sees at most $C_{t1,se1,j} - 1$ customers from its own sub-chain *e1* in front of itself. Since $C_{t1,se1,j}$ is one, the arriving customer will see nothing in front of it. The mean queue length, $L_{t1,se1,j}(\mathbf{N} - \mathbf{e}_j, je = se1)$, shown in (4.6), for server entries *se1* found by (4.5-1), shows the fact.

Similarly, the arriving customer at server entry *se2* only sees customers from its own sub-chain *e2*, which has a maximum two customers. Therefore, $L_{t2,se2,j}(\mathbf{N} - \mathbf{e}_j, je = se2)$ is found by (4.5-3) and shown in (4.6). On the other hand, server entry *se3* receives requests from both sub-chains *e1* and *e2*, so the total number of customers might arrive at *se3* is three, which also is N_j . Then the queue length $L_{t3,se3,j}(\mathbf{N} - \mathbf{e}_j, je)$ is found by $\frac{N_j-1}{N_j}$.

$$\begin{aligned}
 C_{t1,se1,j} = 1 : \quad & L_{t1,se1,j}(\mathbf{N} - \mathbf{e}_j, se1) = 0 \times L_{t1,se1,j}(N); by(4.5 - 1) \\
 C_{t2,se2,j} = 2 : \quad & L_{t2,se2,j}(\mathbf{N} - \mathbf{e}_j, se2) = \frac{1}{2} \times L_{t2,se2,j}(N); by(4.5 - 3); \\
 C_{t3,se3,j} = 3 : \quad & L_{t3,se3,j}(\mathbf{N} - \mathbf{e}_j, se3) = \frac{2}{3} \times L_{t3,se3,j}(N); by(4.5 - 2);
 \end{aligned} \tag{4.6}$$

In contrast, the old solution for $L_{m,e,j}(\mathbf{N} - \mathbf{e}_j)$ is shown in (4.7), where the factor was

found to be $\frac{2}{3}$.

$$\begin{aligned} L_{t1,se1,j}(\mathbf{N} - \mathbf{e}_j) &= \frac{N_j-1}{N_j} \times L_{t1,se1,j}(N) = \frac{2}{3} \times L_{t1,se1,j}(N); \\ L_{t2,se2,j}(\mathbf{N} - \mathbf{e}_j) &= \frac{N_j-1}{N_j} \times L_{t2,se2,j}(N) = \frac{2}{3} \times L_{t2,se2,j}(N); \\ L_{t3,se3,j}(\mathbf{N} - \mathbf{e}_j) &= \frac{N_j-1}{N_j} \times L_{t3,se3,j}(N) = \frac{2}{3} \times L_{t3,se3,j}(N); \end{aligned} \quad (4.7)$$

Case 2: Multiple server entries are accepting requests from multiple sub-chains of a sub-chain dependent client chain. This case is a typical model for devices. In LQN models, every task has a device server which represents the physical devices (CPU or disks). A device server has pseudo entries corresponding to all the entries which physically executing at this device. In Figure 4.5b, suppose a customer from a sub-chain $e1$ arrives at the server entry $se1$:

1. It sees at most $C_{m,se1,k}(< N_k) - 1$ customers from its own sub-chain $e1$.
2. It may see at most $C_{m,se2,k}(< N_k)$ customers from another sub-chain $e2$; since this client chain could have three customers, apart from the arriving customer, the other two customers may stay in the server queue on behalf of the sub-chain $e2$.
3. It may see at most $C_{m,se3,k}(= N_k) - 1$ customers from another sub-chain $e3$.

Therefore, the mean queue lengths with one customer removed are found using (4.8).

$$L_{m,se1,j}(\mathbf{N} - \mathbf{e}_j, se1) = \frac{C_{m,se1,j} - 1}{C_{m,se1,j}} \times L_{m,se1,j}(N) = 0; \quad by(4.5 - 1) \quad (4.8)$$

$$L_{m,se2,j}(\mathbf{N} - \mathbf{e}_j, se1) = L_{m,se2,j}(N); \quad by(4.5 - 3) \quad (4.9)$$

$$\begin{aligned} L_{m,se3,j}(\mathbf{N} - \mathbf{e}_j, se1) &= \frac{N_k - 1}{N_k} \times L_{m,se3,j}(N) \\ &= \frac{2}{3} \times L_{m,se3,j}(N); \quad by(4.5 - 2) \end{aligned} \quad (4.10)$$

In contrast, the old solution for $L_{t1,e1,j}(\mathbf{N} - \mathbf{e}_j)$ is shown in (4.11).

$$L_{t1,se1,j}(\mathbf{N} - \mathbf{e}_j) = \frac{N_j - 1}{N_j} \times (L_{t1,se1,j}(N) + L_{t1,se2,j}(N) + L_{t1,se3,j}(N)) \quad (4.11)$$

Finally, the total queue length with one customer removed at the server entry e is the sum of $L_{m,e,k}(\mathbf{N} - \mathbf{e}_j)$ from all client chains, which is found by (4.12) and (4.13).

$$\begin{aligned} L_{m,e,j}(\mathbf{N} - \mathbf{e}_j) &= \sum_{je} L_{m,je,j}(\mathbf{N} - \mathbf{e}_j, je) \\ &= L_{m,je,j}(\mathbf{N} - \mathbf{e}_j, je) + \sum_{e \neq je} L_{m,e,j}(\mathbf{N} - \mathbf{e}_j, je) \end{aligned} \quad (4.12)$$

$$\sum_k L_{m,e,k}(\mathbf{N} - \mathbf{e}_j) = L_{m,e,j}(\mathbf{N} - \mathbf{e}_j) + \sum_{k \neq j} L_{m,e,k}(\mathbf{N} - \mathbf{e}_j) \quad (4.13)$$

The residence time $W_{m,e,k}(N)$ can be found by (4.14) for a FCFS server queue or by (4.15), for a PS (processor sharing) queue respectively.

$$\begin{aligned} W_{m,e,k}(N) = S_{e,k} &+ S_{e,k=j} \times \left(L_{m,e,j}(\mathbf{N} - \mathbf{e}_j, je) + \sum_{e \neq je} L_{m,e,j}(\mathbf{N} - \mathbf{e}_j, je) \right) \\ &+ S_{e,k} \times \sum_{k \neq j} \sum_e L_{m,e,k}(\mathbf{N} - \mathbf{e}_j, je); \text{ FCFS server} \end{aligned} \quad (4.14)$$

$$\begin{aligned} W_{m,e,k}(N) = S_{e,k} \times \left(1 &+ L_{m,e,j}(\mathbf{N} - \mathbf{e}_j, je) + \sum_{e \neq je} L_{m,e,j}(\mathbf{N} - \mathbf{e}_j, je) \right. \\ &\left. + \sum_{k \neq j} \sum_e L_{m,e,k}(\mathbf{N} - \mathbf{e}_j, je) \right); \text{ PS server} \end{aligned} \quad (4.15)$$

Equations (4.5) and (4.12) for the estimation of queue lengths with one customer removed and (4.14)~(4.15) for residence times $W_{m,e,k}(N)$ are used in the *Schweitzer Core* algorithm, shown in Listing 2.1 in Section 2.3.2.

4.2.2.2 Linearizer AMVA Solution

Linearizer MVA [43] is generally more accurate than Bard-Schweitzer MVA [186]. Linearizer needs to run the *Schweitzer Core* algorithm multiple times with full client populations $\mathbf{N}$ and populations with one customer less ($\mathbf{N} - \mathbf{e}_j$) from a client chain

j ($1 \leq j \leq K$, and K is the number of client chains). The aim of using a population with one less customer is to find the mean queue length $L_{m,e,k}(N)$ at a population $(\mathbf{N} - \mathbf{e}_j)$ to replace the $L_{m,e,k}(\mathbf{N} - \mathbf{e}_j)$ at a full population $\mathbf{N}$.

In the execution of the *Schweitzer Core* algorithm at a full population, $L_{mk}(\mathbf{N} - \mathbf{e}_j)$ is estimated using (4.5). However, for the solution of a population of one customer less, with the existence of a sub-chain dependent client, the number of possible customers may not be unique. In fact, the possible number of customers from a sub-chain might be $C_{m,se1,j}$ or $C_{m,se1,j} - 1$. For example, the server entry *se1*, in Figure 4.5a, accepts requests from the sub-chain *e1*, which is the only client chain (task *t0*). When the full population is $\mathbf{N} = \{1\}$, and the population with one less customer is not just $(\mathbf{N} - \mathbf{e}_j) = \{0\}$. In this case, the server entry still might have customers. Therefore, by considering probabilities and the estimation of all the possible combinations of queue length $L_{mk}(\mathbf{N} - \mathbf{e}_j)$, the computational costs are increased depending on the number of sub-chains with dependencies and the number of client chains.

Using Bard-Schweitzer AMVA, the accuracy of the new solution is significantly better. Therefore, Bard-Schweitzer AMVA is used to solve the submodels with the sub-chain dependencies or interlocks in the LQNS, since Linearizer also has problems in handling interlocks with external contentions. In order to maintain the accuracy of an entire LQN model, Linearizer is used to solve all other submodels without traffic dependencies. Therefore, the corrections made to Bard-Schweitzer AMVA are sufficient to improve the accuracy of the entire solution.

4.2.3 Verification

To verify the improved solution, two experiments [134] were used to demonstrate the improvements. First, the model in Figure 4.1 with same parameters used in Section 4.1 was re-solved by using the improved solution. Figures 4.6a ~4.6c show the new results compared with the previous results from LQNS and simulation. The new

(a) Queueing Time

(b) Processor Waiting

(c) Throughput

Figure 4.6: New results from solving the model in Figure 4.1 (varying c_1).

solution produces correct queueing delays (zeros) of the lightly-loaded entry $e0a$ at the server entry $e1$ (task $t1$), shown in Figure 4.6a (in the figure, both lines for ‘ $e0a(sim)$ ’ and ‘ $e0a(new)$ ’ overlap the X axis). Also, the throughput of top layer client task $c0$ (in Figure 4.6c) and the processor waiting time of the entry $e0a$ (in Figure 4.6b) are greatly improved. The error rate of entry $e0a$ ’s processor waiting time from the new solution is around 10%, which is a significant improvement compared with the previous results by LQNS. For throughput, the error rates of the new solution are around 5%, while the error rates of the previous solution are around 25~35%. Since this is the most severe sub-chain dependency case when a sub-chain only has one customer while the others have multiple customers, the new approach handles the case well.

Table 4.2 shows the results of the same model, but with randomized parameters. The number of customers of `client` and `client1` are uniformly distributed between 2~4 and 2~6 respectively. (Since the previous LQNS solution produces non-zero queueing delays when `client` and/or `client1` have one customer, the error rates in these cases are infinity and make the averages infeasible). The error rates rapidly decrease as the number of customers increases. Finally, the mean service demands of entries $e0a$, $e0b$, $e1$, and $e2$ are random values with normal distributions, as are the mean number of visits to entries $e1$ and $e2$ from entries $e0a$ and $e0b$.

Table 4.2: Results for the Figure 4.1 model with randomized parameters.

$\% \Delta$	λ_{client}	$\lambda_{client1}$	S_{e0a}	S_{e0b}	$W_{e0a,e1}$	$W_{e0b,e2}$	PW_{e0a}	PW_{e0b}	U_{proc}
Method lqns									
mean	-0.07	-5.52	0.16	6.72	6.51	23.84	10.28	10.28	-4.08
sdev	2.02	3.41	2.21	4.74	21.21	21.72	10.23	10.23	2.54
Method new									
mean	2.65	-0.02	-2.72	-0.00	-33.96	-7.39	0.38	0.38	0.66
sdev	2.17	1.03	2.27	1.13	9.58	5.02	2.21	2.21	1.11

The second experiment was done using the same model in Figure 4.1, except the sub-chain dependent client task $t0$ has a finite number of threads, and the number

(a) Queueing Time (e0a(sim) and e0a(new) overlap with X axis)

(b) Processor Waiting

(c) Throughput

Figure 4.7: New results from solving the model in Figure 4.1 (varying t_0 and c_1).

of customers of client *c1* varies from 1 to 7. Figures 4.7a ~4.7c show the new results compared with the results found by LQNS and simulation. Again, the new solution removes the extraneous queueing delay of the lightly-loaded entry *e0a* at server entry *e1*, shown in Figure 4.7a. The maximum error rates of the entry *e0a*'s processor waiting were reduced from 700% to less than 20%, and the error rates of *c0* throughput were reduced from 40% to 20%.

Similarly, Table 4.3 shows the results of the second experiment model with randomized parameters. The mean service demands of the entries and the mean number of visits were randomized in the same way. The number of threads for task *t0* was randomly chosen between 2 and 4. **Client1** has 2-6 customers and the number of customers at **client** was fixed at two. The results of both experiments show that

Table 4.3: Results for the randomized model with a finite number threads.

$\% \Delta$	λ_{client}	$\lambda_{client1}$	S_{e0a}	S_{e0b}	$W_{e0a,e1}$	$W_{e0b,e2}$	PW_{e0a}	PW_{e0b}	U_{proc}
Method lqns									
mean	-1.15	-7.57	-7.20	-1.16	99999.00	2.01	-1.16	-1.16	-6.87
sdev	2.95	3.64	6.06	5.14	0.00	14.73	12.06	12.06	3.32
Method new									
mean	1.49	-6.74	-11.70	-1.57	0.00	-1.21	-1.86	-1.86	-5.85
sdev	3.50	4.41	6.16	3.52	0.00	7.40	9.23	9.23	4.15

the improved solution has a much better accuracy than the previous solution. The improved solution correctly removed the extraneous queueing delays from the lightly-loaded client entries, consequently, the throughput related to these lightly-loaded entries are also improved significantly.

Table 4.4 shows the mean execution times of solving these two sets of randomized models by using the previous solver and the improved solution. The results show that the improved solution increases the execution time by 10%.

Table 4.4: The execution time of two randomized models.

Model	Mean Execution Time (s)	
	Previous	New
The first set	0.0073	0.0085
The second set	0.0076	0.0086

Figure 4.8: An LQN activity model with a sub-chain dependency.

4.3 Sub-Chain Dependency in Parallel Executions

In addition to multi-threading, parallel executions are also modelled in LQN models by *activities* within tasks and second phases within an entry. LQN Activities are used to model internal concurrency and synchronization while second phases are used to increase concurrency and reduce the blocking time of senders.

An imbalanced number of customers arriving at a multi-threaded task (referred to as a multi-server task in LQN models) with activities, also results in sub-chain dependencies, which shown in Figure 4.8. Reducing the impacts of sub-chain dependencies to parallel executions in LQN models is described in this section. First, the use of

Figure 4.9: Precedence Schema of Activities. From LQN user manual [77]

LQN activities is described; then, how to correct the number of maximum possible customers in the improved solution is explained. Finally, the improved results are presented.

4.3.1 LQN Activities and Their Precedences

LQN tasks may have activities which model the sub-events inside tasks, for example, the model in Figure 4.8. LQN activities [78] are used to model internal concurrency with behaviours of fork-join, or internal loops within a task. Activities may happen sequentially or concurrently. The order of the execution of activities is modeled by ‘precedences’ in LQN models, which are used to connect LQN activities.

The precedences in LQN models, shown in Figure 4.9 from [77], are:

1. Sequential precedences are defined as **pre** and **post**, which are used to link the

activities in a sequential execution order.

2. **AND** (&) precedences are defined as **ANDFork** (&) and **ANDJoin** (&). **ANDFork** (&) leads to multiple concurrent activities. **ANDJoin** (&) synchronizes two or more activities before the successor activity starts.
3. **OR** (+) precedences are defined as **ORFork** (+) and **ORJoin** (+). **ORFork** (+) branches executions with probability. The successive activity after **ORJoin** (+), starts when any of these forked activities completes.
4. Loop precedence **Loop** (*) repeats activities for a given times.

Loop and **OR** do not appear in the model in Figure 4.8. The dashed line with an arrow in the figure is a reply. The details about activities and their precedences can be found in [77].

In the model of Figure 4.8, the intermediate server task **t1** has two entries and four activities. Entry **e2** is a regular entry like the entries shown in the figures in previous sections, while the entry **e1** is dedicated for starting activity **a1**. Activity **a1** is followed by two activities **b1** and **b2** which are running concurrently. This is defined by the **ANDFork** (&) precedence between activity **a1**, **b1** and **b2**. Finally, the precedence **ANDJoin** (&) makes sure that both **b1** and **b2** have to be completed before activity **c1** executes.

A sub-chain dependency exists in this model because the server task **t1** is multi-server with a multiplicity of 2, and both its client tasks **client** and **client1** have exactly one customer. Therefore, the intermediate task **t1** is a sub-chain dependent client task when task **t1** makes further requests to the lower level servers, e.g. tasks **d1**, **d2**, **d3** and **d4**. Consequently, inside task **t1**, activities **b1** and **b2** are sub-chain dependent too. The requests from activity **b1** to entry **d1**, from **b2** to entry **d2** and from entry **e2** to entry **d4** do not have any queueing delays. However, the results are found by the existing LQN analytic solver, shown in Figure 4.8, present non-zero

values on these three calls (the numbers without brackets cross the arcs) and indicate that the existing solver fails to handle this situation.

4.3.2 Improvements and Results

As discussed previously, the key point of solving the sub-chain dependency with the existence of activities, is correcting the number of customers of a sub-chain dependent client. First, *MCs* for every client entries, which include the activity starting entry (e.g. entry **e1** in Figure 4.8) and all regular entries (e.g. entry **e2**), need to be calculated using (4.2)~(4.4) in Section 4.2.1.

Then, the maximum number of customers arriving at all of the activities within the sub-chain dependent task need to be corrected. Corrections start from the activity starting entry, then follow the sequence of precedences to change *MCs* of all activities if necessary. All activities have a same number of customers with their starting entry except those activities after **join** precedences. The steps are:

1. For the activity right after its starting entry, assign its *MC* with the same value of the starting entry's *MC*;
2. For a sequenced activity, assign its *MC* with the value of its predecessor activity's *MC*;
3. For the activities after a fork (either **ANDFork** or **ORFork**), their *MCs* all have same value of their predecessor activity before the fork;
4. For an activity after a join, for example, activity *a*, if all the predecessor activities joining into activity *a* are originated from the same starting entry, then activity *a* also has the same *MC* as the starting entry. Otherwise, the total *MC* for the multiple starting entries, which are involved in this join, need to be found first. Then the smaller value between this total and the task's multiplicity, is assigned to the activity *a* as its *MC*.

Table 4.5: Results comparison for the model in Figure 4.8.

\$N	Extraneous waiting by LQNS		Percent relative error			
			e2 to d4 waiting		client throughput	
	b1-d1	b2-d2	LQNS	new	LQNS	new
1	0.103	0.156	∞	0	-9.51	-1.05
2	0.081	0.122	43.55	11.05	-9.96	-3.95
3	0.061	0.091	17.67	8.17	-10.26	-6.25
4	0.047	0.069	9.55	5.95	-11.46	-8.77
5	0.038	0.038	6.0	4.44	-12.86	-10.99

To verify this correction, the model in Figure 4.8 was re-solved by varying the number of customer of `client1` (one of the reference tasks) from 1 to 5. All of the results found by the improved solver were compared to the results found from using the previous solver and simulations with a 99% confidence interval. Table 4.5 shows the comparison of the queueing delays at server entries $d1$, $d2$ and $d4$.

The non-zero queueing delays at server entries $d1$ and $d2$ produced using the existing LQNS solver in columns 2 and 3, and are completely removed in the new solution. The error rates of the queueing delay at server entry $d4$ from the previous and new solutions are shown in columns 3 and 4. Finally, the error rates of client throughput for both solutions are shown in the last two columns. The results in this table show that the new solution is able to produce much better results than the previous one.

4.4 Summary

The sub-chain dependent problem is another type of traffic dependencies in layered queueing networks and occurs when a multi-service server has an imbalanced number of customers arriving at its provided services and the server makes further requests to some lower layer servers. The decomposition method used by the LQN analytic solver makes this kind of traffic dependency inevitable. The phenomena of sub-chain dependent problems result in extra queueing delays to the sub-chains with the least

number of customers.

The key point of removing the impacts caused by sub-chain dependent clients, is finding the correct number of customers from a sub-chain dependent client, referred to as the maximum possible customer, MC . The corrected MC s are used to estimate the queue length with one customer removed in Bard-Schweitzer AMVA. Errors are substantially reduced and in some cases eliminated altogether.

Chapter 5

Case Study: An Online Bookstore

The purpose of this case study is to evaluate the improved solution in handling traffic dependencies using an online bookstore application system named RADSBKStore. RADSBKStore has both interlocking and sub-chain dependencies, and the existence of these traffic dependencies has severe impacts on the accuracy of performance evaluation.

5.1 RADSBKStore

RADSBKStore is a hypothetical online bookstore used to study the creation of performance models from requirements [168] and from traces of execution of a prototype [105]. RADSBKStore is a typical three-tier application. The top tier has two classes of clients, the users who are visiting and interacting with the book store, and the administrator who performs maintenance, for example, backing up all the transactions periodically. The intermediate tier is the application tier, and implements the business model and includes typical e-commerce functionality. The bottom tier is the database tier, which stores all the information of the bookstore, including product categories, inventory, user accounts, and order transactions.

Figure 5.1 shows the LQN model of RADSBKStore. Clients are modelled as tasks

(**Customer** and **Administrator**) running on their own processors generating requests to the application via a load balancer labelled **RADSbookstore**. The application tier runs on the processor **BookstoreProc** and consists of the following operations:

1. general processing of all customer requests, done by task **Server**;
2. checkout and billing, fulfilled by task **ShoppingCart**;
3. user account management, performed by task **CustAccMgr**;
4. inventory management, managed by task **InventoryMgr**; and
5. back order management, achieved by task **BackorderMgr**,

The database tier consists of two databases, inventory and customer account database, which are running on processor **DatabaseProc**.

5.1.1 Traffic Dependencies in **RADSbookstore**

5.1.1.1 Sub-chain dependency

The load balancer, **RADSbookstore**, located between the client and application servers is a common strategy in designing web services to distribute incoming requests to a number of application servers. It is implemented as an infinite server with two entries, **BStore_E1** and **BStore_E2**, to forward user requests and administration requests respectively. However, incorporating the load balancer into the model introduced sub-chain dependencies into the model, because there is only one administrator while there are multiple ordinary customers. So the load balance becomes a sub-chain dependent client. When using the existing solver, extraneous queueing delay occurs at server entry **BOMgr_E2** (task **BackorderMgr**) from entry **BStore_E2** due to sub-chain dependencies. Since the entire back order operation has a total service demand with a mean of 5250 ms, the extraneous queueing delay is significant, which is around 500 ms. This

Figure 5.1: An online bookstore model From [105].

Figure 5.2: The submodel of server **InventoryMgr** (the 4th layer).

extraneous queueing delay seriously affects the accuracy of the prediction of the server which processes the user requests.

5.1.1.2 Interlocks

The RADShookstore model is heavily interlocked. First, there exists a deeply layered sending interlock in the system. Figure 5.2 show the submodel of the fourth layer. At this submodel, clients **BStore_E2** and **BOMgr_C1** and server entries **IVMgr_E7** and **IVMgr_E8** construct a sending interlock. Entry **BStore_E2** is the common entry and the parent of **BOMgr_C1** through **BOMgr_E2**. Since **BStore_E2** has only one customer, the requests arrive at server entry **IVMgr_E7** and **IVMgr_E8** cannot see each other in the server queue. Meanwhile, in the lower submodel at the DB server, all the client tasks **IVMgr_C1~IVMgr_C5** originating from task **InventoryMgr**, and going through the server entries, **DB_E1~DB_E6** (exclude **DB_E2**) construct a split interlock. The client entry **IVMgr_E2** and server entry **DB_E1** is a Type3 sending interlock. Since the common source task **InventoryMgr** has only one thread, all the requests arrive at the

Table 5.1: The error rates of the queueing delays at server `InventoryMgr`

Client entry	Server entry	Percent relative error	
		old	new
<code>BStore_E2</code>	<code>IVMgr_E7</code>	215.77	-33.56
<code>SE1</code>	<code>IVMgr_E1</code>	90.25	-5.26
<code>SE2</code>	<code>IVMgr_E2</code>	180.46	39.15
<code>SE3</code>	<code>IVMgr_E3</code>	108.01	3.34
<code>SE4</code>	<code>IVMgr_E4</code>	105.84	2.26
<code>BOMgr_C1_E1</code>	<code>IVMgr_E8</code>	1122.82	40.08
<code>SC_E3</code>	<code>IVMgr_E5</code>	529.41	69.63
<code>SC_E3</code>	<code>IVMgr_E6</code>	528.76	69.46

interlocked server, `Database`, are not able to see each other. There is no queueing delay at all for any requests.

Since the entire application tier is running on the processor `BookstoreProc`, and the load balancer `RADSbookstore` originates all the requests, the processor `BookstoreProc` is also an interlocked server.

5.1.2 Evaluation

The improvements first focus on queueing delays at the servers involved in the back order operations, e.g. `BOMgr_E2`, `BOMgr_C1_E1` `IVMgr_E7` and `IVMgr_E8`. The extraneous queueing time at server entry `BOMgr_E2` is reduced from 489 to 2.93 [134]. The queueing delay of `BOMgr_E2` and `BOMgr_C1_E1` at server `InventoryMgr` are improved. Furthermore, all the queueing delays at server task `InventoryMgr` are improved; the relative errors compared to simulation, $\frac{(t-t_{\text{sim}})}{t_{\text{sim}}} \times 100$, for the old and new algorithms are shown in Table 5.1. In all cases, the new algorithm is superior.

Table 5.2 shows the relative error of the service time of the entries that serve administrator or bookstore customers and Table 5.3 shows the relative error of the throughput for the two customer classes. Finally, Table 5.4 shows the root of the mean of the difference squared (RMS) for all of the performance results of the model. These tables indicate that the results found using the new algorithm are superior to

old algorithm.

Table 5.2: The error rates of service time

Entries	Percent relative error	
	old	new
Administrator_R	3.10	0.42
BStore_E2	61.13	7.67
BOMgr_E2	100.38	5.59
BOMgr_C1_E1	100.33	5.62
Customer_RefE	2.00	0.34
BStore_E1	60.30	10.09
SE1	25.75	2.42
SE2	119.66	27.68
SE3	101.85	3.14
SE4	99.82	2.11
SE5	145.12	4.87
SE6	0.55	0.61

Table 5.3: The error rates of throughput

Entries	Percent relative error	
	old	new
Administrator	-3.01	-0.42
Customer	-1.97	-0.34

Table 5.4: RMS error compared to simulation of the RADSbookstore model.

Result	Old	New
Queueing Time	665.41	32.44
Service Time	6.78	0.80
Throughput	2.16	0.34
Processor Utilization	2.01	0.34

5.2 Conclusions

In LQN networks, a sub-chain dependency occurs when an intermediate server both serves more than one class of customers, has more processing threads than potential customers arriving at one of its service entry points and makes further requests to some lower layer servers. The sub-chain dependent problem causes extra queueing

delays to the sub-chains with the lower number of customers, and in some cases results in queueing delays where none actually exists. RADSBookstore is a hypothetical model of an e-commerce book store. The sub-chain dependency in the RADSBookstore model is the most significant source of errors which severely affect the accuracy of the analytic solver. The response times of the regular users and the administrator client task were overestimated by up to 60%, due to the load balancer. The reason is that the load balancer is a sub-chain dependent task. Applying the improved solution, the extraneous queueing delay that is hidden in the response time of the administrator client task is eliminated up to 99% (at server entry `BOMgr_E2`, the time is reduced from 489 to 2.93).

Interlocks also cause problems to RADSBookstore with the solution. In the model, the server tasks `InventoryMgr` and `Database` and the processor `BookstoreProc` are all interlocked servers. Split interlocks and sending interlocks occur at several locations in the RADSBookstore system. The sending interlocks cross multiple layers and also include Type3 sending interlocks. After applying the improved solution, the error rates of the queueing delays at the interlocked server `InventoryMgr` are substantially reduced by a factor of 20 times. Overall, all the results found by the new approach are much better than the previous solution.

Chapter 6

Modelling Decisions in LQN

This chapter describes how the LQN model was extended to accommodate decisions by simulation and the method of dynamic parameter substitutions (DPS). Four non-trivial decision patterns are described, and methods to solve them analytically using LQX are demonstrated. In addition, the four LQN decision patterns are implemented in the LQN simulator, LQSIM. The results produced by LQSIM were verified by GreatSPN [7, 47, 48], a Petri net solver. As a result, the simulator, LQSIM, is able to provide results which are used to verify those results found analytically by the method of DPS.

6.1 Decisions in LQN

The decisions described here are restricted to exceptions handling within the context of resource allocations in distributed systems. A LQN decision is made at a branching point, with an unknown probability to each of multiple execution paths, which is to be determined based on dynamic execution performance results.

Figure 6.1 shows a general scenario of a decision in resource allocation, using a UML [158] sequence diagram, which is widely used to represent scenarios of dynamic behaviours of systems. This scenario is that a resource-acquiring request is sent

Figure 6.1: A decision scenario of an exception handling in resource allocations.

to `aResource`, if `aResource` is available, the resource is allocated to the request; otherwise, this request fails. Exceptions, which arise when resource allocations fail, can be handled in several ways:

1. discarding messages or requests immediately. This is common in communication networks. The probabilities of dropping messages for open arrivals can be handled by the current LQN solution.
2. wait for some time and then give up.
3. wait for some time and then make another attempt. The number of retries may be finite or infinite.
4. perform some necessary clean-up operations before leaving. These clean-up operations may be required, for example, to release resources a request is holding. These operations are modelled as extra delays.

The scenario in Figure 6.1 can be modelled by the LQN model shown in Figure 6.2a if the probabilities of successful resource acquisition, P_{succ} , and unsuccessful resource acquisition, P_{fail} , are known. In Figure 6.1, `resource` is a queueing server, while the

Figure 6.2: LQN model for a decision scenario.

task **abort** is a delay server. The client entry *requester* makes requests to server entries **aResource** and **delay** with routing probabilities of P_{succ} and P_{abort} .

Using the current LQN solver, input parameters are not allowed to be changed as the solver runs. However, both the probabilities of success P_{succ} and failure P_{fail} depend on the dynamic execution of a system and are unknown initially. To accommodate decisions with unknown branching probabilities, a new LQN model element, *Decision*, is defined. The newly defined notation of a decision is a shaded parallelogram which is placed on the arc of the success path of a decision and is shown in Figure 6.2b. The letter in the shaded parallelogram is one of ‘A’, ‘T’, ‘I’, and ‘R’, which stand for Abort, Timeout, infinite-retry and finite-retry decisions, respectively. The number of execution paths of a decision may be two or three, depending on the type of decision. In the notation, P_{succ} and P_{abort} are dynamic parameters instead of pre-defined inputs. The LQN decision patterns are discussed in the following section.

6.2 LQN Decision Patterns

This section defines five LQN decision patterns; four of them are from Wu’s decision patterns in [224], and the fifth is a new pattern, *timeout*. The four patterns defined by Wu are:

1. The *queue* pattern is essentially “no exception”, because the requester waits forever until it is successful.
2. The *abort* pattern ends a caller’s request prematurely.
3. The *infinite-retry* pattern keeps trying until the resource request is successful.
4. The *finite-retry* pattern makes a fixed number of retries before giving up.

The fifth decision pattern, *timeout*, is a common decision widely used in distributed systems. A request may time out either on its queueing delay or on the response time. The timeout pattern considered here is confined to the timeouts on the queueing delays. A timeout on response time should be defined as another individual decision pattern, because what happens when a timeout occurs in the middle of execution is quite a system specific decision.

In each of the following sections, a LQN decision pattern is defined as well as the input file template.

6.2.1 Queue Pattern

Wu’s *queue* pattern is not considered further here, because no decision needs to be made. LQN, as well as other queueing-based formalisms, are able to handle this case without any difficulties.

6.2.2 Abort Decision Pattern

Scenario Description.

The *abort* pattern arises when a request for a resource fails, in which case the requester simply gives up. Failure may require some necessary cleanup operations during the procedure of exception handling. The scenario of a LQN abort pattern is shown in Figure 6.3.

Figure 6.3: Abort decision scenario.

Use of Pattern.

This pattern is used for handling the requests for locks or some resources with a limited queue size when some cleanup operations are required along the exception path.

Input Parameters.

1. $T_{abort}(\geq 0)$ is the mean service demand required by the clean-up operations.
2. The location of the decision is specified by the associating entry.
3. Two execution paths, the success path and the abort path, are specified by the calls, which are bound to the decision paths.

Output Parameters.

P_{succ} and P_{abort} are the probabilities that requests are succeed or aborted, with $P_{succ} + P_{abort} = 1$.

6.2.3 Infinite-Retry Decision Pattern**Scenario Description.**

The *infinite-retry* pattern arises when an application keeps trying to request a resource until it succeeds. If a request fails, the application will sleep (or suspended) for T_{sleep} units of time before it tries again. Figure 6.4 shows the sequence of events for this pattern.

Figure 6.4: Infinite-retry decision scenario.

Input Parameters.

1. T_{sleep} (> 0) is a fix amount of delay before retrying. If $T_{sleep} = 0$, it is a normal queue model without any decision.
2. The location of the decision: the associating entry.

3. The calls are bound to the two execution paths, the success path and the retry path.

Output Parameters.

n_{retry} (≥ 0) is the mean number of retries before a resource is allocated successfully. Since a request keeps retrying until allocation is successful, the probability of success P_{succ} always is one.

6.2.4 Finite-Retry Decision Pattern

Scenario Description.

The *finite-retry* pattern arises when an application only makes a finite number of requests for a resource. If a request fails, the application will sleep for T_{sleep} units of time before it tries again. If the maximum number of retries is reached and the allocation is not successful, the abort path is taken and the request fails. Figure 6.5 shows the sequence of events for this pattern.

Figure 6.5: Finite-retry decision scenario.

Input Parameters.

1. T_{sleep} (> 0) is a fixed amount of delays before retrying. If $T_{sleep} = 0$, it is an abort decision pattern. Since all the retry request retries arrive at the same moment when the resource is busy, the request is definitely aborted.
2. M_{retry} is the maximum number of retries allowed before a request is aborted.
3. T_{abort} (≥ 0) is the mean service demand required by the clean-up operations.
4. The location of the decision is specified by the associating entry.
5. Three execution paths, the success path, fail path, and the retry path are specified by the calls bound to the paths.

Output Parameters.

1. P_{succ} and P_{abort} are the probabilities of the success path and the abort path, with $P_{succ} + P_{abort} = 1$;
2. n_{retry} (≥ 0) is the mean number of retries for all requests either succeed or aborted.
3. n'_{retry} (≥ 0) is the mean number of retries before it succeeds.

6.2.5 Timeout Decision Pattern**Scenario Description.**

The timeout pattern arises when the response time for a service exceeds some value, $T_{timeout}$. The scenario of a LQN timeout decision pattern is shown in Figure 6.6.

Use of Pattern.

This case is widely used in web-based systems and communication protocols, like TCP.

Figure 6.6: Timeout decision scenario.

Input Parameters.

1. $T_{timeout}$ (> 0) is the time limit for a timeout decision.
2. T_{abort} (≥ 0) is the mean service demand required by those clean-up operations.
If $T_{abort} = 0$, a timed out request is simply discarded.
3. The location of the decision is specified by the associating entry.
4. Two execution paths, the success path and the timeout path, are specified by the calls, which are bound to decision paths.

Output Parameters.

P_{succ} and $P_{timeout}$ are the probabilities of normal executions and timeouts;

6.2.6 LQN Decision Schema

LQN models are composed using three different schemas, which describe LQN models, LQN submodels, and LQN components. To accommodate decisions in LQN models, the top-level schema was modified by adding a decision element, shown in Figure 6.7. LQN decisions are defined as a primary LQN model element.

Figure 6.7: Top level schema of LQN models.

Figure 6.8 shows the schema of the LQN Decision element. Based on the definitions of LQN decision patterns, LQN Decisions have four types (*PatternType*), which are abort, timeout, infinite-retry, and finite-retry. In addition to the name and type, a LQN DecisionType may have attributes: abort (for clean-ups), timeout, sleep, and max-retries, depending on its *PatternType*. Each decision has two or three execution paths, namely success, fail (timeout or abort) and retry, which are defined by *DecisionPathType* in Figure 6.8. Currently, each *DecisionPath* is bound to a pair of entries, **fromEntry** and **toEntry**. With this definition, decisions are able to access all the performance parameters of the related tasks via their entries.

Figure 6.8: Schema of the LQN Decision.

6.2.6.1 Pattern Template in LQN Input Model

Any of LQN decisions can be defined using this template in LQN input files by specifying the necessary parameters. For example, Listing 6.1 shows a finite-retry decision. It specifies that the location of the decision (by the attribute `fromEntry`) and its three execution paths, as well as the input parameters, T_{sleep} , T_{abort} , and M_{retry} .

Listing 6.1: XML file template for a *finite-retries* decision.

```

1  <decision name="aFiniteRetry" type="retry"
2      sleep="0.5" abort="0.1" max-retries="3"
3      fromEntry="requesterEntry"
4      decision-entry = "finiteRetryEntry">
5      <path type="success" toEntry="resource"/>
6      <path type="retry" toEntry="retryDelay"/>
7      <path type="abort" toEntry="abortDelay"/>
8  </decision>

```

Abort, timeout and infinite retries decisions use similar templates but with only two execution paths. The attribute `timeout` with a positive value is required for a timeout decision, and a negative value is used for the attribute `max-retries` for an infinite-retry decision.

6.3 Decisions in LQSIM

In this section, the four LQN decision patterns are generalized into two decision patterns, timeout/abort decisions and retry decisions. An abort decision is regarded as a special case of timeout decision, where the timeout threshold is zero. An infinite-retry decision is also a special case of finite-retry, which the max-retries is infinite. Therefore, only these two general decisions are implemented in the simulator by two newly created LQSIM tasks, `Timeout_Decision` task and `Retry_Decision` task [137]. The results of LQSIM are verified by the results of GreatSPN, a Petri net solver, and some other existing simulation results from [224].

6.3.1 Timeout/Abort Decisions

Both the timeout and abort decision patterns were implemented by the `Timeout_Decision` task in LQSIM. A `Timeout_Decision` task consists of a queue task and timeout worker tasks. A `Timeout_Decision` task itself is a queue task (described in Section 2.4.4.1), which receives requests and dispatches these requests to the associated (resource) server task whenever the server is available; otherwise, the request at the front of the queue waits at the `Timeout_Decision` task. If the waiting time of the request at the `Timeout_Decision` task reaches the timeout threshold $T_{timeout}$, it is dispatched to one of the available timeout workers to perform the clean-up operations required by handling exceptions.

If the timeout threshold $T_{timeout}$ is zero, the timeout decision is an abort decision. All requests arriving when the resource server is busy are forwarded to the timeout workers immediately. These timeout workers are used to consume the time T_{abort} , which simulates the cleanup operations of timed-out requests; as many of the workers are created as needed. Figure 6.9 shows the translated model of a `Timeout_Decision` task in LQSIM. The reason for using forwarding calls is that replies are sent to the original sender of requests instead of the `Timeout_Decision` task after the services of the associated (resource) server or timeout workers are completed.

Figure 6.9: Translated model of `Timeout_Decision` in LQSIM

A `Timeout_Decision` task has three ports, `std_port`, `ready_port` and `worker_port`,

Figure 6.10: Message movements of a Timeout_Decision task.

which are shown in Figure 6.10. Figure 6.10 also shows how messages move in and out of a `Timeout_Decision` task through these ports. The requests from a sender task are delivered to `std_port` to enter the server queue. Both `ready_port` and `worker_port` are used to receive ‘ready’ messages from the associated server and timeout workers. Ready messages indicate the server or timeout workers are available and ready to receive requests. The algorithm of the `Timeout_Decision` task is shown in Listing 6.2.

For all successful requests, the waiting times which occur at the `Timeout_Decision` task are accumulated as the mean queueing delay of the requests on the success path of the decision. The waiting times of any timed-out requests at the `Timeout_Decision` task are $T_{timeout}$ and they are counted as an extra service demand in addition to T_{abort} for the clean-up operations. The probability of taking the success path, P_{succ} , is the fraction of the number of the requests successfully dispatched to the resource server, N_{succ} , against to the total number of requests N_{req} arriving at the `Timeout_Decision`

Listing 6.2: Timeout_Decision task routine.

```

1  while (true) {
2      msg msg_in = receive(std_port, -1);
3      wait= ps_now -msg_in.timestamp;
4      timeout=Ttimeout-wait;
5      if (timeout<0) timeout=0;
6      if (receive(ready_port, timeout)){
7          send(server.std_port, msg_in);
8      }else{
9          msg worker_ready=receive(worker_port,0);
10         send(worker_ready.reply_port, msg_in);
11     }
12 }

```

task, by (6.1), in which both N_{succ} and N_{req} are accumulated during simulation.

$$\begin{aligned}
 P_{succ} &= \frac{N_{succ}}{N_{req}}; \\
 P_{abort} &= 1 - P_{succ}
 \end{aligned}
 \tag{6.1}$$

6.3.2 Retry Decisions

Both infinite-retry and finite-retry decisions were implemented by a **Retry_Decision** task. A **Retry_Decision** task has a queue task and an infinite number of retry workers. It is also paired with an associated resource server task and an abort server task (an infinite queue task), shown in Figure 6.11.

Unlike a **Timeout_Decision** task, a **Retry_Decision** task does not queue any requests. A request arriving at the **Retry_Decision** task may be dispatched to the resource server task if the resource is available; otherwise, it is either dispatched to one of retry workers or is dispatched to the abort server if the number of retries exceeds the maximum number (M_{retry}) allowed to retry.

Retry workers are used for holding unsuccessful requests for a fixed amount of time. Requests that come back from any retry worker are retry requests and are delivered to the **std_port** of the **Retry_Decision** task to again make another attempt.

Figure 6.11: Message movements in a `Retry_Decision` task.

Figure 6.11 also shows how messages move in and out of a `Retry_Decision` task. All messages in the simulation carry a time stamp when they are sent. Retry requests are distinguished from the initial requests (from the sender) through the time stamps. If a time stamp does not equal the current time when the message arrives, this message must be a retry request because no queueing delay occurs at the `Retry_Decision` task. Like `Timeout_Decision` tasks, successful requests and aborted requests are dispatched using forwarding, so that replies are sent to the sender of requests directly. The algorithm for the `Retry_Decision` task is shown in Listing 6.3.

For a finite-retry decision, P_{succ} is the fraction of the number of successful requests (N_{succ}) against the total number of initial requests (N_{req}), which includes both

Listing 6.3: Retry_Decision task routine.

```

1 while (true)
2 {
3     msg msg_in = receive(std_port, -1);
4     wait= ps_now -msg_in.timestamp;
5     nRetries=wait/T_sleep;
6     if (receive(ready_port, 0)){
7         send(server.std_port, msg_in);
8     }else{
9         nRetries +=1;
10        if (max_retry!=-1) && (nRetries>max_retry){
11            send(abort.std_port, msg_in)
12        }else{
13            msg worker_ready=receive(worker_port,0);
14            send(worker_ready.reply_port, msg_in);
15        }
16    }
17 }

```

successful and abort requests. The probabilities P_{succ} and P_{abort} are found by (6.2).

$$\begin{aligned}
 N_{req} &= N_{succ} + N_{abort}; \\
 P_{succ} &= \frac{N_{succ}}{N_{req}}; \\
 P_{abort} &= 1 - P_{succ} = \frac{N_{abort}}{N_{req}};
 \end{aligned} \tag{6.2}$$

Figure 6.12: A LQN models for a finite-retry decision.

A LQN model of a finite-retry decision is shown in Figure 6.12. In the figure, n_{retry} is the mean number of retries of all requests, found by (6.4), and it is the fraction of the total number of retry requests N_{retry} over the total number of initial requests N_{req} . In (6.3), N_{retry} consists of the retry requests by successful requests and abort requests. In a finite-retry decision, every aborted request has the number of M_{retry} attempts before it aborts while a successful request may have the number of N_i retries before it succeeds, which N_i has a range from zero to M_{retry} , inclusive. By combining (6.3) and (6.4), n_{retry} can be split into two parts, which are caused by successful requests or aborted requests in (6.5). n'_{retry} explicitly denotes the mean number of retries by all the successful requests, found by (6.8). Therefore, the mean number of retries, n_{retry} and n'_{retry} can be calculated from each other using the probabilities P_{succ} and P_{abort} based on (6.7).

$$N_{retry} = \sum_{i=0}^{M_{retry}} i \times N_i + M_{retry} \times N_{abort}; \quad (6.3)$$

$$n_{retry} = \frac{N_{retry}}{N_{req}}; \quad (6.4)$$

$$= \frac{\sum_{i=0}^{M_{retry}} i \times N_i}{N_{req}} + M_{retry} \times \frac{N_{abort}}{N_{req}}; \quad (6.5)$$

$$= \frac{\sum_{i=0}^{M_{retry}} i \times N_i}{N_{succ}} \times \frac{N_{succ}}{N_{req}} + \frac{M_{retry} \times N_{abort}}{N_{req}};$$

$$= \frac{\sum_{i=0}^{M_{retry}} i \times N_i}{N_{succ}} \times P_{succ} + M_{retry} \times P_{abort}; \quad (6.6)$$

$$n_{retry} = n'_{retry} \times P_{succ} + M_{retry} \times P_{abort}; \quad (6.7)$$

$$n'_{retry} = \frac{\sum_{i=0}^{M_{retry}} i \times N_i}{N_{succ}} \quad (6.8)$$

where: N_i is the number of requests that have retried i times.

The mean residence time of successful requests W_{succ} and aborted requests at the

decision are found by (6.9) and (6.10).

$$W_{succ} = S_{succ} + n_{retry} \times T_{sleep}; \quad (6.9)$$

$$W_{abort} = T_{abort} + M_{retry} \times T_{sleep}; \quad (6.10)$$

A negative M_{retry} (*line 10* in Listing 6.3) indicates that an infinite number of retries are permitted. Therefore, all the requests will succeed eventually and no requests will be aborted, so P_{succ} is 1 and P_{abort} is 0. Equations (6.3) ~ (6.8) become (6.11), (6.12) and (6.13). The mean residence time of successful requests W_{succ} at the decision is also found by (6.9).

$$N_{retry} = \sum_{i=1}^{\infty} i \times N_i; \quad (6.11)$$

$$n_{retry} = \frac{N_{retry}}{N_{req}} = \frac{\sum_{i=0}^{\infty} i \times N_i}{N_{req}}; \quad (6.12)$$

$$n_{retry} = n'_{retry} \quad (6.13)$$

6.3.3 LQSIM Results Verification

In this section, first, the results produced by LQSIM were verified by GreatSPN [7, 47, 48], a Petri net solver. Then, the results of LQSIM were compared with some other existing results of simulations which were from Wu [224].

6.3.3.1 Verification Through GreatSPN

The verification through GreatSPN is using a simplified web server model with an abort, timeout, infinite-retry and finite-retry decisions respectively. The LQN model of this web server model with an abort decision is shown in Figure 6.13, and the other three decision models are very similar. These LQN models with decisions were solved by the simulator LQSIM and their corresponding Petri net models were solved by GreatSPN.

Figure 6.13: The LQN model of the simplified web server model with an abort decision.

1. Timeout_Decision Task

LQSIM `Timeout_Decision` tasks are used to model both abort decisions and timeout decisions. First, the Petri net model of this abort decision is shown in Figure 6.14. This GSPN model was analytically solved using GreatSPN. The number of customers was varied from 1 to 10 by a step of 1. Figures 6.15 show the probability of abort and the throughput of users from GreatSPN and the simulator LQSIM. These figures show that LQSIM produces results comparable to those from GreatSPN.

Figure 6.14: The Petri net model of an abort pattern.

Figure 6.15: Results from LQSIM and GreatSPN of a simple abort pattern.

Figure 6.16: The Petri net model of a timeout pattern.

Next, Figure 6.16 shows the corresponding Petri net model of this simplified web server model with a timeout decision. Similarly, the number of customers is also varied from 1 to 10 by a step of 1. Figure 6.17 shows the probabilities of abort and the throughput of users found by using GreatSPN and LQSIM. Again, the results found by LQSIM compare quite favourably with GreatSPN.

Figure 6.17: Results from LQSIM and GreatSPN of a timeout pattern.

2. Retry_Decision Task

Both the infinite retry decision and finite retry decision were verified by GreatSPN [7, 47, 48], also through this simplified web server model.

First, the Petri net model of this web server model with an infinite-retry decision is shown in Figure 6.18, and this Petri net model was analytically solved using GreatSPN. In this Petri net model, transition t_3 is fired when a request succeeds while t_6 is fired when this request has to retry. Therefore, the mean numbers of retries n_{retry}^{pn} were found by the throughput ratio of transitions t_6 and t_3 , using (6.14).

$$n_{retry}^{pn} = \frac{\lambda_{t6}}{\lambda_{t3}} \quad (6.14)$$

Figure 6.19 shows the results from GreatSPN and the simulator LQSIM, in terms of the mean number of retries and the throughput of users. LQSIM produced the

results comparable to those found using GreatSPN.

Figure 6.18: A Petri net model of an infinite-retry decision.

Figure 6.19: Results from LQSIM and GreatSPN for the infinite-retry decision pattern.

The Petri net model of the web server model with a finite-retry decision is shown in Figure 6.20; this model was also solved using GreatSPN.

Similarly, the mean number of retries and the probabilities of success and abort had to be calculated manually through the throughput of transitions. The probability of success P_{succ}^{pn} was found by the ratio of transitions $T4$ and $T3$ using (6.15). The probability of abort P_{abort}^{pn} can be found by P_{succ}^{pn} or the ratio of transitions $T5$ and $T3$ using (6.15).

$$P_{succ}^{pn} = \frac{\lambda_{T4}}{\lambda_{T3}} \quad (6.15)$$

Figure 6.20: The Petri net model of a finite-retry decision. ($M_{retry} = 3$)

$$\begin{aligned}
 P_{abort}^{pn} &= 1 - P_{succ}^{pn} \\
 P_{abort}^{pn} &= \frac{\lambda_{T5}}{\lambda_{T3}}
 \end{aligned} \tag{6.16}$$

The mean number of retries, n_{retry}^{pn} , requires the throughput of transitions t_7 , t_{12} , t_{14} , and t_{10} . The transitions t_7 , t_{12} , or t_{14} are fired when the server is available after a request has made 0, 1 or 2 retries, respectively. Transition t_{10} is fired when the server is still not available after the request has made a maximum number of retries; this request then moves to the abort path. The mean number of retries n_{retry}^{pn} was found by (6.17).

$$n_{retry}^{pn} = \frac{\lambda_{t7} \times 0 + \lambda_{t12} \times 1 + \lambda_{t14} \times 2 + \lambda_{t10} \times 2}{\lambda_{T3}} \tag{6.17}$$

Figures 6.21 show the results found using GreatSPN and the simulator LQSIM in terms of the mean number of retries, the probability of abort and the throughput of users. Again, LQSIM produces results that compare favourably with GreatSPN.

Figure 6.21: Results from LQSIM and GreatSPN of a finite-retry decision.

6.3.3.2 Verification Through CSIM

Because of the state explosion problem of Petri net solutions, GreatSPN has difficulties in solving moderately sized systems. In order to test the capability of LQSIM for solving moderate or large-scale systems and verify the accuracy of the LQSIM results, Wu's database model [224] was used here. This model is a web server system with a database, which is shown in Figure 6.22a, and has an emphasis on the database lock.

In this model, the web server task (10 threads) and database server task (5 threads)

are deployed on two separate server machines. The services **query** and **update** provided by the database are modelled by a LQN task **dbWorker** with two entries. A single task **dblock** models the lock of the database with a zero service time. The task **dbControl** in the figure acts as a pseudo source of all the requests arriving at the database. All the update requests from **dbControl** have to get the lock first in order to update the database while the query requests are served by the entry **query** immediately. All the disk operations required by reading or writing the database are modelled by a task **disk** with an entry **diskop**.

This database model with an abort decision (Figure 6.22a), infinite-retry (excluding the abort task in Figure 6.22b) and finite-retry (Figure 6.22b) decisions were solved by Wu using his hybrid approach HPMM [224] and simulations (CSIM). In this section, this database model was re-solved using LQSIM, and the results from LQSIM were compared with Wu's simulation results. Since Wu did not handle time-out decisions, only the results of abort, infinite-retry and finite-retry decisions were compared here.

1. Abort decision pattern

The model with an abort decision shown in Figure 6.22a, in which, the entries **dbReq**, **dblock** and **abort** form an abort decision. The experiments were performed by varying the number of customers (the reference task **user**) from 2 to 30 by a step of 2.

Figure 6.23 shows the results from LQSIM and CSIM from [224], which include the probabilities of lock acquiring, P_{succ} , and the mean holding time of **dblock** in the abort decision model.

Figure 6.22: LQN models with decisions.

Figure 6.23: Results from LQSIM and CSIM [224] for an abort pattern.

2. Infinite-retry decision pattern

Similarly, this web-based database model with an infinite-retry decision shown in Figure 6.22b (excluding the abort task), and the entries `dbReq`, `dblock` and `sleep` form the infinite-retry decision. The experiments were also performed by varying the number of customers (the reference task `user`) from 2 to 30 by a step of 2.

Figure 6.24 shows the results from LQSIM and CSIM regarding the mean number of retries n_{retry} and the mean holding time of `dblock` in this infinite-retry decision.

Figure 6.24: Results from LQSIM and CSIM [224] for the infinite-retry pattern.

For both abort and infinite-retry decisions, the differences between these two simulators are quite small.

Figure 6.25: Results from LQSIM and CSIM [224] for a finite-retry pattern.

3. Finite-retry decision pattern

This web-based database model with a finite-retry pattern has both **sleep** and **abort** tasks along with **dbReq** and **dblock** tasks to construct the finite-retry decision, which is shown in Figure 6.22b. The maximum number of retries M_{retry} in this decision is set to 3. The number of customers of task **user** was also varied from 2 to 30 by a step of 2 when this model was solved by LQSIM.

Figure 6.25 shows the results from LQSIM and CSIM, which include the probabilities of lock acquiring P_{succ} , the mean number of retries n_{retry} , and the mean holding time of **dblock** in this finite-retry decision. Note that results from CSIM for more than 24 customers were not shown in [224].

Both GreatSPN and CSIM validate the accuracy of LQSIM. Therefore, LQSIM results are used to verify the results found by the analytic solution (*DPS*) in the following section.

6.4 Dynamic Parameter Substitutions Using LQX

Dynamic parameter substitution (*DPS*) is the process of mapping dynamic parameters to certain performance results, which are obtained by MVA solutions. This section explains how to substitute dynamic parameters for each decision pattern using LQX. DPS functions have been developed only for three patterns: abort, timeout and infinite-retry, so far.

6.4.1 Method of Dynamic Parameter Substitutions

The probabilities of all the execution paths of decisions or the number of visits of an execution path in retry decision patterns are modelled as dynamic parameters in LQN models. Thus, these parameters can be changed based on the performance results as a model is being solved.

The Layered Queueing eXperiment language, LQX [150], presents a way to perform dynamic parameter substitutions outside the LQNS solution iterations. DPS happens in the outermost iterations (the four iterations of LQX were described on Page 39) because the value stored in any dynamic parameter does not change inside `solve()`. A complete model is solved, then its results are used to compute new values for dynamic parameters after which the model is then solved again. This is equivalent to hybrid performance modelling since this approach solves the model to completion each time `solve()` is invoked. However, with LQX, there is no requirement to solve a model to completion before changing parameters. Therefore, compared to hybrid approaches, DPS through LQX has reduced I/O processing and execution time due

to only using a single solution space.

The DPS algorithms used in LQX for these three LQN decision patterns are described in the following sections; Table 6.1 lists the parameters used in these sections.

Variables	Descriptions
w_q	The mean queueing delay at resource , excludes the service demand and nested delays at any lower layer resources required by resource .
L_q	The mean queue length of requests on the success path, excludes the one in service. $L_q = w_q \lambda_{succ}$.
L	The mean queue length of requests at resource . $L = (S_{succ} + w_q) \lambda_{succ}$.
S_{succ}	The mean service time on the success path, includes all the queueing delay at lower levels.
λ_{succ}	The throughput of resource .
$P_{succ}^{(n)}$	The probability of going to the success path in the n^{th} iteration.

Table 6.1: The performance parameters used in DPS algorithms.

6.4.2 Abort Decision Pattern

Figure 6.26 shows a LQN model of an abort decision pattern. A request from the task **users** to **resource** aborts if it has to queue at the resource, and then goes to the task **Abort_Delay**. The probabilities of success and abort, P_{succ} and P_{abort} are determined iteratively through the dynamic parameters, $\$y_{succ}$ and $\$y_{abort}$.

Assume that any requests going to the abort path would wait in the queue of **aResource** if they were not aborted immediately. Therefore, the probability of abort, P_{abort} , is related to the likelihood of requests staying in the queue. The value of the dynamic parameter $\$y_{abort}$ is found from the ratio of the mean service demand of the resource entry **aResource** (S_{succ}) and the mean queueing delays (w_q) at the resource by using (6.18).

$$\begin{aligned} \$y_{abort} &= \frac{L_q}{L} = \frac{w_q}{S + w_q} \\ \$y_{succ} &= 1 - \$y_{abort} \end{aligned} \tag{6.18}$$

Listing 6.4 shows the LQX algorithm for calculating the probabilities of LQN

Figure 6.26: A LQN model for an abort decision pattern.

abort decision patterns. In Listing 6.4, the dynamic parameters y_{succ} and y_{abort} start with initial values of 1 and 0, respectively. They are recalculated using (6.18) in each LQX iteration, and the new values are fed to the next invocation of the function `solve()` at line 14. During the iterations, as y_{abort} increases, the mean queueing delay w_q at **aResource** decreases, which also results in the reduction of y_{abort} in the next iteration.

The `for` loop controls convergence, which the criteria of convergence is that the square of the difference of y_{abort} between two successive iterations is less than 5×10^{-5} . `call()`, `phase()`, `entry()`, and `task()` are LQX built-in functions to access the attributes and results of these LQN elements. The purpose of line 19 is to prevent y_{abort} from making large oscillations, especially in the first several iterations.

The abort decision model in Figure 6.13 was re-solved by using this algorithm. The results of DPS, shown in Figures 6.27, were compared with the results by LQSIM and GreatSPN. In Figure 6.27a, the abort probabilities are very accurate. The execution times of DPS, GreatSPN and simulation are shown in Figure 6.27b, the execution

Listing 6.4: LQX-DPS algorithm of a LQN abort decision

```

1 function abort_prob_calculation(
2     request_name, /* string */
3     resource_name /* string */ )
4 {
5     $y_succ = 1.0;
6     $y_abort = 0.0;
7     last_y1 = 2;
8     last_y2 = 2;
9     succ_path_call = call(phase(entry(requester_name),1),
10                           resource_name);
11     resource_phase = phase(entry(resource_name),1);
12
13     for ( delta = 1; delta >= 0.00005; ){
14         solve(); /* invoke the LQNS solver */
15         w_succ = succ_path_call.waiting;
16         s_succ = resource_phase.service_time;
17         prob = w_succ / (w_succ + s_succ);
18
19         $y_abort = ($y_abort + 4.0 * prob) / 5.0; /*favour to prob*/
20         $y_succ = 1 - $y_abort;
21         delta = (last_y2 - $y_abort) ** 2;
22         last_y1 = $y_succ;
23         last_y2 = $y_abort;
24     }
25 }

```

times of DPS are less than GreatSPN in most of the cases. The average execution time of DPS is 8 *ms* and the maximum execution time is less than 20 *ms*. However, the maximum execution time of GreatSPN is 36 *ms* when there are 30 customers in the model.

Figure 6.27: Results of DPS comparing with LQSIM and GreatSPN of an abort pattern.

6.4.3 Timeout Decision Pattern

A timeout occurs when the queueing delay at a resource server is too long. The timeout decision pattern is solved by DPS in two ways, referred as **Algorithm 1** and **Algorithm 2**.

6.4.3.1 Algorithm 1

Figure 6.28 shows a general LQN model for a timeout decision pattern. At resource server **aResource**, requests have a mean queueing delay w_q , and mean queue lengths L_q (in the queue, but not in service), defined in Table 6.1 on Page 165. The mean service demand of the task **Timeout_Delay** is the sum of $T_{timeout}$ and T_{abort} because all timed out requests have spent $T_{timeout}$ in the queue before going to the timeout path to do the clean-up operations.

Figure 6.28: LQN model for the timeout decision pattern.

When $w_q > T_{timeout}$, those requests with queueing delays larger than the timeout threshold have to go to the timeout path. So, the value of $L_q - T_{timeout} \times \lambda$ is the mean number of requests that have wait longer than $T_{timeout}$ in the queue of **resource**. Therefore, the fraction of requests being moved from the queue of **resource** to the timeout path is:

$$f = \frac{L_q - T_{timeout} \times \lambda}{L} = \frac{w_q - T_{timeout}}{w_q + S_{succ}}. \quad (6.19)$$

After some requests are moved to the timeout path, the total number of requests arriving at **aResource** is reduced. Consequently, the mean queueing time w_q at **aResource** is also reduced. If $w_q < T_{timeout}$ in any of the iterations after the first iteration, it indicates that too many requests have been moved to the timeout path, so some requests have to be moved back to **aResource**. Finally, w_q reaches $T_{timeout}$ after the algorithm converges.

Similarly, the probability of timeout $P_{timeout}$ is calculated iteratively through the dynamic parameter $\$y_{timeout}$. The process of finding $\$y_{timeout}$ is as follows. In the solution, the lower bound (B_l) and the upper bound (B_u) of $\$y_{timeout}$ are used to reduce the number of iterations by shrinking the bounds. Initially, $\$y_{succ}^{(1)}$ and $\$y_{timeout}^{(1)}$

are set to 1 and 0, and the lower bound and the upper bound of $y_{timeout}$ are set to 0 and 1 respectively. The evolution of the two bounds, B_l and B_u , is shown by (6.20). The dynamic parameter $y_{timeout}$ at the $n + 1^{th}$ iteration is found by (6.21), in which, f is the fraction of requests to be moved to the timeout path at the current iteration, which is found by (6.19). The dynamic parameter y_{succ} for the probability of P_{succ} can be found easily by (6.21) at the $n + 1^{th}$ iteration.

At the $n + 1^{th}$ iteration:

$$\begin{cases} B_l = y_{timeout}^{(n)}; & w_q > T_{timeout} \\ B_u = y_{timeout}^{(n)}; & 0 < w_q < T_{timeout} \text{ and } y_{timeout}^{(n)} > 0 \end{cases} \quad (6.20)$$

$$\begin{aligned} y_{timeout}^{(n+1)} &= \begin{cases} y_{timeout}^{(n)} + f; & w_q > T_{timeout} \\ \frac{B_u + B_l}{2.0}; & 0 < w_q < T_{timeout} \text{ and } y_{timeout}^{(n)} > 0 \\ 0; & 0 \leq w_q \leq T_{timeout} \text{ and } y_{timeout}^{(n)} = 0 \end{cases} \\ y_{succ}^{(n+1)} &= 1 - y_{timeout}^{(n+1)} \end{aligned} \quad (6.21)$$

In (6.21), if $w_q=0$, no requests wait in the queue of `aResource`, so no requests will time out. If $0 \leq w_q \leq T_{timeout}$ in an iteration and $y_{timeout}$ is zero, then no requests will time out. When any of these two cases occurs in the first LQX iteration, there is no need to iterate further. Otherwise, the value of w_q may approach $T_{timeout}$ from both sides of $T_{timeout}$ during the iterations, and the two bounds of $y_{timeout}$ approach towards each other at the same time. When the two bounds meet, the solution stops. Therefore, dynamic parameters $y_{timeout}$ and y_{succ} at the final iteration are the desired probabilities $P_{timeout}$ and P_{succ} of the timeout decision.

Listing 6.5 shows the LQX function for calculating the dynamic parameter $y_{timeout}$ of LQN timeout decisions using the **Algorithm 1**. This algorithm loses accuracy when $P_{timeout}$ is less than 0.1 because the queueing delay w_q is a mean value and can be

Listing 6.5: LQX function of Algorithm 1 for timeout decisions

```

1 function timeout_prob_calculation(
2     requester_name, /* string */
3     resource_name,  /* string */
4     T_timeout       /* double */ )
5 {
6     $y_succ=1.0;
7     $y_timeout=0.0;
8     lowerBound=0.0;
9     upperBound=1.0
10    last_y1=2;
11    last_y2=2;
12    succ_path_call=call(phase(entry(requester_name),1),
13                        resource_name);
14    resource_phase=phase(entry(resource_name),1);
15
16    for ( delta=1; delta>=0.00005; ){
17        solve();
18        w_succ= succ_path_call.waiting;
19        s_succ= resource_phase.service_time;
20
21        if (w_succ>T_timeout){
22            lowerBound= $y_timeout;
23            prob=(w_succ-T_timeout)/(w_succ+ s_succ);
24            $y_timeout=$y_timeout +prob;
25        }else if ($y_timeout>0.){
26            upperBound=$y_timeout;
27            $y_timeout=(upperBound+lowerBound)/2.0;
28        }else{
29            last_y1=$y_succ;
30            last_y2=$y_timeout;
31        }
32        $y_succ=1- $y_timeout;
33        delta=(last_y1-$y_succ)**2
34              +(last_y2-$y_timeout)**2;
35        last_y1=$y_succ;
36        last_y2=$y_timeout;
37    }
38 }

```

smaller than $T_{timeout}$. Nevertheless, a small number of timed out requests may exist but are drowned out by the majority of non-timeout requests. Therefore, **Algorithm 2** for timeout decisions was developed, shown in the following section.

6.4.3.2 Algorithm 2

A timeout decision also can be modelled in another way, shown in Figure 6.29. This transformed model uses an abort decision to model a timeout decision. To construct the abort decision, an intermediate multi-threaded task is added between the source entry **requester** and the destination entry **aResource** on the success path of the timeout decision. The intermediate task, referred to as **non_timeout_client**, represents a pseudo source to issue all the success requests to the resource.

Figure 6.29: Another LQN model for LQN timeout decisions.

This pseudo source **non_timeout_client** has zero service demand. Its multiplicity is a dynamic parameter m , which is the ceiling of $T_{timeout}$ divided by the total service time of the resources S_{succ} , shown in (6.22). In (6.22), the capacity of the resource

m_{res} , for example, the size of buffer, is also taken into account.

$$m = \lceil \frac{T_{timeout}}{S_{succ}} \times m_{res} \rceil \quad (6.22)$$

During the LQX iterations, m may either be a constant or a variable, which depends on whether the particular resource has nested queueing delays or not. If the value of m is properly chosen, it ensures that the queueing delays of the requests from the pseudo source `non_timeout_client` must be less than $T_{timeout}$. Any requests queued at `non_timeout_client` should be timed out. Therefore, the original requester, the pseudo source `non_timeout_client`, and task `timeout_Delay` form an abort decision. Consequently, the probability of abort at `non_timeout_client` becomes the probability of timeout at the resource. Therefore, the dynamic parameter $y_{timeout}$, for the probability of timeout $P_{timeout}$, can be found by (6.18) in Section 6.4.2, but w_q in (6.18) is the queueing delay at the pseudo source `non_timeout_client` rather than the delay at the resource.

Both m and $y_{timeout}$ have to be calculated iteratively using *Algorithm 2* in Listing 6.6. Initially, m has the same value as the multiplicity of the requester, so no requests queue at the pseudo task. The calculation of $y_{timeout}$ starts from the second iteration after m is recalculated until the algorithm converges. Finally, the probabilities $P_{timeout}$ and P_{succ} are the values of $y_{timeout}$ and y_{succ} at the last iteration.

6.4.3.3 Discussions

A web server system with a timeout decision, shown in Figure 6.30, is used to compare these two algorithms. The results, shown in Figure 6.31 were found by these two algorithms as well as by the LQN simulator, LQSIM. Both Figure 6.31a and Figure 6.31b show the probabilities of timeout, $P_{timeout}$ of the system. The latter is

Listing 6.6: LQX function of Algorithm 2 for the timeout decision in Figure 6.29.

```

1  function timeout_prob_calculation2 (
2      requester_name,      /* string */
3      resource_name,       /* string */
4      pseudo_task_name,    /* string */
5      T_timeout            /* double */
6  {
7      $y_succ=1.0;
8      $y_timeout=0.0;
9      lowerBound=0.0;
10     upperBound=1.0;
11     last_y2=2;
12     $m=$n;
13     i=1;
14     succ_path_call=call(phase(entry(requester_name),1),
15                          pseudo_task_name);
16     pseudo_task_phase=phase(entry(pseudo_task_name),1);
17     resource_phase=phase(entry(resource_name),1);
18
19     for (delta=2; delta>0.000005 or i<=2;
20         delta=(last_y2-$y_timeout)**2)
21     {
22         last_y2=$y_timeout;
23         solve();
24         s_resource= resource_phase.service_time;
25         w_succ=succ_path_call.waiting;
26         s_succ=pseudo_task_phase.service_time;
27         $m=ceil(T_timeout/s_resource);
28
29         if (w_succ>0.0){
30             lowerBound=$y_timeout;
31             $y_timeout+=w_succ/(w_succ+s_succ);
32         } else if(i>1) {
33             upperBound=$y_timeout;
34             $y_timeout=(upperBound+lowerBound)/2.0;
35         }
36         $y_succ=1-$y_timeout;
37         i+=1;
38     }
39 }

```

Figure 6.30: A webserver system with timeout.

an enlarged view of Figure 6.31a with the number of customers between 400 and 600. Figure 6.31c shows the computation costs required by both algorithms, in terms of the number of invoking the function `wait()` (in *SchweitzerCore*).

Overall, *Algorithm 1* has the same level of accuracy compared with *Algorithm 2*, except when the number of customers is between 400 and 600. However, *Algorithm 2* incurred more computational effort (around $2 \sim 4$ times) than *Algorithm 1*, shown in Figure 6.31c. Therefore, a trade-off has to be considered.

6.4.4 Infinite-Retry Decision Pattern

An *infinite-retry* pattern occurs when requests for a resource fail then make retry attempts until the requests succeed. No unsuccessful requests leave the system, so they may visit the retry path several times before the resource is successfully allocated to them. Therefore, the mean number of requests visiting the retry path, y_{retry} , depends on w_q , and can be found by (6.23).

$$y_{retry} = \frac{w_q}{T_{sleep}} \quad (6.23)$$

$$n_{retry} = y_{retry} \times \left(1 + \frac{S}{S + w_q}\right) = \frac{w_q}{T_{sleep}} \times \left(1 + \frac{S}{S + w_q}\right) \quad (6.24)$$

(a) Probability of Timeout.

(b) Probability of Timeout (between 400 and 600 customers).

(c) Effort of Computation

Figure 6.31: Results from the two DPS algorithms for a timeout decision.

Listing 6.7: LQX function for infinite-retry decisions.

```

1  function infRetry_prob_calculation(requester_name ,
2                                     resource_name , T_sleep)
3  {
4      last_y2=2;
5      $y_succ=1.0;
6      $y_retry=0.;
7      succ_path_call=call(phase(entry(requester_name),1),
8                           resource_name);
9      resource_phase=phase(entry(resource_name),1);
10
11     for (delta=1; delta>=0.00005; delta=(last_y3-$y_retry)**2)
12     {
13         solve();
14         w_succ= succ_path_call.waiting;
15         s_succ= resource_phase.service_time;
16         last_y3=$y_retry;
17         $y_retry= w_succ /T_sleep;
18         n_retry= $y_retry *(1+ s_succ/(w_succ+s_succ));
19     }
20     return (n_retry);
21 }

```

Because of the nature of queueing models, there are queueing delays at *resource*, which are caused by the successful requests. Therefore, y_{retry} is not same as the mean number of retries n_{retry} in DPS. The mean number of retries, n_{retry} , is a product of y_{retry} and a factor of $(1 + \frac{S}{S+w_q})$, which is given by (6.24).

Listing 6.7 shows the LQX function for calculating the dynamic parameters, n_{retry} and y_{retry} , of LQN infinite-retry patterns.

The infinite-retry decision model in Figure 6.13 was re-solved by using this algorithm. The results from DPS and LQSIM and GreatSPN, are shown in Figures 6.32. The mean numbers of retries are very accurate. The execution time of DPS, GreatSPN and simulation are shown in Figure 6.32b.

As for the execution time, the algorithms of DPS in LQX are quite efficient. The execution times of DPS, GreatSPN and LQSIM for the abort decision are shown in Figure 6.27b while execution times for the infinite-retry decision are in Figure 6.32b.

Figure 6.32: Results from DPS, LQSIM and GreatSPN for an infinite-retry pattern.

For the infinite-retry decision, in Figure 6.32b, the execution times of the simulations are from 0.3 to 1.0 seconds as the number of customers increases from 1 to 10. The solution times of GreatSPN are between 6~16 *ms*. In the figure, the curve of the execution times of DPS is accumulated execution time. In most of the cases, the accumulated execution times of DPS are less than GreatSPN. The accumulated execution time of DPS at the last (the tenth) experiment is 17 *ms*. Therefore, the average execution time for these ten experiments is 1.7 *ms*. For the abort decision, the average execution time of DPS is less than 8 *ms*. At the last experiment, when there are 30 customers, DPS used less than 10 *ms* to solve the model while GreatSPN used 36 *ms*. Most important thing is that the speed of increases for DPS is much slower than GreatSPN as systems grow larger. Therefore, DPS is far more scalable than GreatSPN and simulation.

6.4.5 DPS Demonstration

Again, Wu's database model [224] was used to demonstrate the correctness of these DPS algorithms in this section. The models and the processes of verifying the abort pattern and the infinite-retry pattern were exactly the same with the corresponding

LQSIM verification in Section 6.3.3.2.

6.4.5.1 Abort Pattern

The model with an abort decision shown in Figure 6.22a, in which, the entries **dbReq**, **dblock** and **abort** are formed an abort decision. The dynamic parameters $y1$ and $y2$ are the probabilities, P_{succ} and P_{abort} .

The results found by the DPS method were compared with both Wu's HPMM and simulations (CSIM) results from [224], which are presented in Figure 6.33. This set of figures include the probabilities of lock acquiring P_{succ} , the mean holding time of **dblock**, the utilization of the **disk**, the mean response time of web users, the 'useful throughput' (excluding the abort requests), and the computation costs of DPS through LQX. Compared with Wu's results, overall the method of DPS has a similar accuracy of HPMM, but more accurate than HPMM for finding response time.

6.4.5.2 Infinite-retry Pattern

The model with an infinite-retry decision is shown in Figure 6.22b, but excludes the abort task. In this model, the entries **dbReq**, **dblock** and **retry** are formed an infinite-retry decision. The dynamic parameters $y3$ is the mean number of visiting the retry loop, y_{retry} . The set up of the experiments are same with the abort decision model.

The performance results presented in Figure 6.34 are also similar to the abort model, except, the mean number of retries, n_{retry} (in Figure 6.34a) instead of the probability P_{succ} in the abort model since requests never abort in an infinite-retry decision. Compared with Wu' HPMM, the method of DPS overall has a better accuracy than the hybrid approach HPMM.

(a) Probability of lock acquiring.

(b) Dblock holding time.

(c) Utilization of Disk.

(d) Response Time of web users.

(e) Useful throughput (excludes abort requests).

(f) Computation Effort.

Figure 6.33: Results of the model in Figure 6.22a with the abort decision.

Figure 6.34: Results of the model with the infinite retries decision.

6.4.5.3 Timeout Pattern

The model used to verify the DPS algorithms of timeout decisions, was the web server model shown in Figure 6.30. This model has a large number of customers (from 100 to 1000), so it cannot be solved by GreatSPN. (GreatSPN only can solve this model with the number of customers up to 50.) This web server system has a performance requirement that a request for retrieving a web page times out if it has a queueing delay longer than one second at the task *webserver*. In this model, the entries `requestPage`, `retrievePage` and `timeout` are formed a timeout decision. The dynamic parameters $\$y1$ and $\$y2$ are used to calculate the probabilities P_{succ} and $P_{timeout}$.

The experiments of this timeout model were conducted by varying the number of customers from 100 to 1000 using a step of 100. The results in Figure 6.35, include the probabilities of timeout, $P_{timeout}$, the mean response time of web users (the reference task `requestPage`), the throughput of the web server, and the computation costs of DPS through LQX. Since timeout decisions were not considered by Wu's HPMM [224], the result comparison of timeout decisions was only between DPS and LQSIM.

Overall, the DPS approach has a high accuracy when compared with LQSIM. The maximum error rates ($\frac{DPS-LQSIM}{LQSIM} \times 100$) of response times of web users are less than 0.04% for both the algorithms of DPS. The maximum error rates of useful throughput are less than 0.001% (Algorithm 1) and 0.0001% (Algorithm 2). The maximum error rates of $P_{timeout}$ of the Algorithm 1 is 49%, occurs in a population of 525, while for the Algorithm 2, it is 34% with a population of 500. They are in the area of small timeout probabilities, which is mentioned in Section 6.4.3.3. Beyond this area, the maximum error rates of $P_{timeout}$ are 9% (Algorithm 1) and 5% (Algorithm 2).

Figure 6.35: Results of the model with the timeout decision.

6.5 Summary

This chapter explained how to handle decisions in the LQN models through both simulations and dynamic parameter substitutions. The decisions can be handled in LQN models were defined using four LQN decision patterns: abort, timeout, infinite-retry and finite-retry.

In LQN simulator, LQSIM, the `Timeout_Decision` server task was developed to handle both abort and timeout decisions while the `Retry_Decision` server task was for both infinite, finite retry decisions. The accuracy of LQSIM was verified first using

GreatSPN, a Petri net solver, using a small size model (a simplified web server model). Then, LQSIM was also verified by using an existing model borrowed from [224], which was solved by simulation (CSIM). LQSIM produced results quite similar to GreatSPN and CSIM. Therefore, LQSIM can provide results, which can be used to verify the results found by the LQN analytic solution.

Dynamic Parameter substitution takes advantage of the very method used to solve a layered queueing network analytically, which is an iterative, fast and scalable solution. Dynamic parameter substitution extends this method to allow arbitrary output to be fed back as input for subsequent iterations of the model. This method was implemented using the LQX language. The DPS mapping functions using LQX were developed for the three LQN decision patterns, abort, timeout, and infinite-retry decisions.

The results of DPS were compared with GreatSPN and LQSIM, as well as Wu's HPMM [224] and simulation results. DPS is able to produce comparable results. The DPS solution is quite efficient. Since all computation is performed within the context of a single operating system process, no file input/output or process switching is required. The execution time of DPS is less than GreatSPN. The scalability of the LQN analytic solution makes the method of dynamic Parameter substitution superior to state-based methods and hybrid methods like Wu's HPMM.

Chapter 7

Analytic Solution of Fair Share Scheduling

In this chapter, the DPS approach was adopted into LQN models to solving fair share scheduling [116] analytically. First, the DPS approach is implemented by LQX, then DPS is integrated into the LQN analytic solver LQNS.

7.1 Fair Share Scheduling

Fair Share Scheduling (FS) was introduced by Kay [116], to prevent allocating resources inappropriately in a student computing center environment. In FS scheduling, a value is assigned to each group of users who have the right to consume a resource. This value specifies how much resource a group is allowed to have; an entitlement of resources is referred to as a share. A FS scheduler distributes the usage of a resource to each task or group based on its share. FS scheduling is intended to prevent tasks or processes from consuming more resources than that they are supposed to, or even to monopolize some resources [218].

Many algorithms to achieve the goal of the fairly sharing resource have been developed. Fair queueing [62, 152] and its variants [87, 156, 194, 201] are used in all

sorts of communication networks (packets [29, 62, 152, 201] or wireless [176]). Proportional share scheduling [211, 212], includes lottery scheduling [60, 212], stride scheduling [6, 213] (deterministic allocation), Surplus Fair Share (SFS) [42], and Completely Fair Share (CFS) [164]. Some other algorithms exist, such as the Borrowed Virtual Time (BVT) [66] in virtual machine environments, and a discrete time-limit vocation model [2].

FS scheduling has been supported in some operating systems, for example, Unix type operating systems [97] supported it as early as 1984, and the CFS [164] scheduling was adopted into the Linux kernel in the release of 2.6.23. The impact of FS scheduling in virtual machine environments have been studied in [46, 66, 68, 216, 217].

Accommodating FS scheduling into LQN models was studied in [132], in which CFS [164] was implemented in the PARASOL Simulation Engine [153] and the LQN simulator, LQSIM. The principle of this implementation is:

1. Every group has a priority based on its share and how long its tasks have been waiting already.
2. If a group occupies resources more than it is entitled to, all the tasks within the group were suspended (forced to sleep) for an amount of time, in order to let other groups consume some of the resources;
3. The amount of sleep time was proportional to the amount of the excessive resources it consumed. Tasks in a suspended group re-enter the ready-to-run-queue of the FS processor after the sleep time of the group is exhausted;

The solution in [132] was a simulation solution, so scalability is the main concern. The study of a hybrid solution in the LQN solver [133] was initially motivated to solve FS scheduling in the LQN solver faster than the simulation solution in [132]. In the hybrid solution, simulation was used to solve the particular submodel only, where the processor with a FS scheduler resides; the rest of submodels are solved by

the analytic solver LQNS. This approach works well if the FS processor is not an interlocked server. Unfortunately, this is not a common case.

This chapter applies the DPS approach to handling FS scheduling analytically. The DPS functions for a FS scheduler are developed based on the ideas of the CFS scheduler used in [132], which were enumerated above. The probabilities of groups going to sleep are dynamic parameters that will be solved by using the DPS substitution functions.

7.1.1 Group Shares

A share [116] is the entitlement of each task or group to use a resource, which aims to prevent the group from monopolizing resources (channel or processor). In LQN models, all tasks using a processor with a FS scheduler are grouped into LQN groups [132]. Then, a group share (GS_i) is assigned to each group i ; and all the tasks within the group i share or consume CPU time evenly.

Guarantee and cap [29, 30] are two kinds of group shares. A guarantee share $GS_{(g,i)}$ defines the minimal amount of resources that a group i is allowed to use while a cap share $GS_{(c,i)}$ specifies the maximum amount. If there are some extra resources, the group with a guarantee share could consume more, but a group with a cap share cannot.

In LQN models, group shares are specified as a percentage of a resource which groups are allowed to use; the value of a GS is between 0 and 1 (inclusive). The total share of all groups, $\sum_{i=1}^G GS_i$ is 1, (G is the total number of groups scheduled by a FS scheduler). Therefore, group shares may need to be normalized if the sum of all group shares specified in an input file does not equal to 1. The normalization follows the rules from [132]:

1. If the total group share, $\sum_{i=1}^G GS_i$ from an input file is greater than 1, then all

group shares scale down by multiplying a factor $\frac{1}{\sum_{i=1}^G GS_i}$, shown in (7.1). Guarantee shares and cap shares are scaled in the same rate. This case is treated as an error in an input file. Instead of stopping, the model is solved with a warning message.

$$GS'_i = \frac{GS_i}{\sum GS_i}, \sum GS_i > 1 \quad (7.1)$$

where: GS'_i is the group share of the group i after normalization.

2. If the total group share, $\sum_{i=1}^G GS_i$ from an input file is less than 1, all the cap shares $GS_{(c,i)}$ are not changed. Only the guarantee shares $GS_{(g,i)}$ are scaled up by (7.2). Scaling guarantee shares in this way indicates that all the groups with guarantee shares have rights to consume more resources, but the rights are proportional to their original shares.

$$\begin{aligned} GS'_{(g,i)} &= GS_{(g,i)} \times \frac{1 - \sum GS_{(c,i)}}{\sum GS_{(g,i)}}, \sum GS_i < 1 \\ GS'_{(c,i)} &= GS_{(c,i)}, \end{aligned} \quad (7.2)$$

where: $GS'_{(g,i)}$ and $GS'_{(c,i)}$ are group shares of the group i after normalization.

7.2 Dynamic Parameter Substitution Using LQX

In this section, the DPS approach was implemented by using LQX [150] functions, which are referred to as LQX-DPS functions. The implementation of these LQX-DPS functions demonstrates the principles of analytically solving fair share scheduling in LQN models, and is the foundation of the implementation in LQNS.

7.2.1 FS Processor Transformation

The pre-condition of using the LQX-DPS functions to solve LQN models with FS processors is transforming a FS processor into a *normal* processor. Here, a normal processor is a processor with FCFS or PS scheduling, which has a product-form solution and can be solved by MVA.

Figure 7.1: The model Transformation of a FS processor.

First, a FS processor (**ApplicCPU** in Figure 7.1a) is transformed into a normal processor with a shadow task, shown in Figure 7.1b. This shadow task (**delaytask** in Figure 7.1b) is a delay server and used to simulate the extra delays of the groups that are forced to wait. It has the same number of service entries as the FS processor, and each entry has the same service demand with its corresponding device entry of the FS processor. The shadow task runs on its own processor, an infinite server. The scheduling policy of the transformed processor can be FCFS or PS.

Dynamic parameters ($y1$ and $y2$ in Figure 7.1b), which represent the mean number of visits to the entries of the shadow task, are assigned to the transformed model. All calls originating from the tasks within the same group use the same dynamic parameter. The mean number of visits is changed based on their group shares

during the solver's iterations. Therefore, the extra delays at the shadow task for each group reflects the relationships between its group share and the actual consumption of resources.

7.2.2 Mappings to Dynamic Parameters

Based on Little's Law [139], the relationship between the FS processor and its transformed processor can be expressed by (7.3), in which the subscripts 'F', 'T' and 'D' stand for the FS processor, transformed processor and the delay task respectively. With the existence of the shadow delay task, the mean queue length $L_{(F,i)}$ of a group i at the FS processor is the sum of the mean queue length at the transformed processor, $L_{(T,i)}$, and the utilization of the shadow delay task by the group i . Since the service demand of an entry of the delay task is same as the corresponding device entry of the FS processor, the utilization $U_{(D,i)}$ of the delay server task is the product of the utilization of the FS processor $U_{(F,i)}$ and $\$y_i$. The mean residence time $W_{(F,i)}$ of the group i at the FS processor is the sum of the mean residence times at the transformed processor, $W_{(T,i)}$, and all the delays at the delay task.

$$\begin{aligned}
 L_{F,i} &= L_{T,i} + U_{D,i}; \\
 W_{F,i} &= W_{T,i} + \$y_i \times S_{D,i}; \\
 U_{F,i} &= U_{T,i}; \\
 U_{D,i} &= \$y_i \times U_{T,i};
 \end{aligned} \tag{7.3}$$

When the utilization of the transformed processor from a group i , $U_{(T,i)}$ is greater than its group share GS_i , the processor usage of this group has to be reduced by increasing $\$y_i$, which increases the time the group spends at the shadow task. The fraction $\frac{U_{(T,i)} - GS_i}{U_{(T,i)}}$, is the ratio of the processor usage being reduced. Therefore, a number of customers from the group i being moved to the shadow task is determined by (7.4), and the increment of $\$y_i$, Δy_i is found by (7.5). Then, the new value of

$\$y_i^{n+1}$ in the next iteration is updated by (7.6).

$$\Delta L_{(T,i)} = \frac{(U_{(T,i)} - GS_{gi})}{U_{(T,i)}} \times L_{(T,i)} \quad (7.4)$$

$$\Delta y_i = \frac{\Delta L_{(T,i)}}{U_{(T,i)}} = \frac{U_{(T,i)} - GS_i}{U_{(T,i)}^2} \times L_{(T,i)} \quad (7.5)$$

$$\$y_i^{(n+1)} = \$y_i^{(n)} + \Delta y_i \quad (7.6)$$

The utilization $U_{(T,i)}$ of a group i may be smaller than its group share GS_i . This may be caused by one of the following reasons:

1. Group i does not get enough usage because other groups have consumed more;
2. Group i does not have enough customers to consume the resource. This case is the reason why surplus resource exists;
3. The oscillations of a group utilization as it approaches its group share in the solution process.

The common characteristic of the first two cases is $\$y_i = 0$, which means they do not spend any time at the delay server. For Case 1, the group i may get more usage after other groups are limited by their shares in later iterations. For Case 2, the spare share may be assigned to some other groups with guarantee shares. For Case 3, the group utilization oscillates as it approaches its group share. For this case, $\$y_i$ of the next iteration is updated by (7.7), which calculates the decrement $\Delta \$y_i$.

$$\begin{aligned} \Delta L_{(T,i)} &= \frac{GS_i - U_{(T,i)}}{U_{(T,i)}} \times L_{(T,i)}; \\ \Delta y_i &= \min\left(\frac{\Delta L_{(T,i)}}{U_{(T,i)}}, \$y_i^{(n)}\right); \\ \$y_i^{(n+1)} &= \$y_i^{(n)} - \frac{\Delta y_i}{2}; \end{aligned} \quad (7.7)$$

The LQX-DPS algorithm is shown in Listing 7.1 (excluding line 30 and line 31). This algorithm performs the substitutions of $\$y_i$ without considering the surplus share

Listing 7.1: LQX function of number of calls to the shadow task

```

1 function calls_to_shadow_task(
2     ngroups, /* int */
3     groups[] /* Group */ )
4 {
5     double $y[ngroups], diff, ql, last_y, delta;
6     double proc_utilization;
7     for(int g = 0; g < ngroups; g++){
8         $y[g]=0.0;
9         group[g].currentShare = groups[g].share;
10    }
11    for ( delta = 1; delta >= 0.00005; ){
12        solve();
13        delta = 0.0;
14        proc_utilization = group[0].processor().utilization();
15        for(int g = 0; g < ngroups; g++){
16            diff=1.0-group[g].currentShare/groups[g].utilization;
17            ql = groups[g].waiting * groups[g].throughput;
18            ql += groups[g].utilization;
19            diff *= ql / groups[g].utilization;
20            last_y = $y[g];
21
22            if(diff > 0)
23                $y[g] += diff;
24            else {
25                diff = min((-1)*diff, $y[g]);
26                $y[g] -= diff / 2.0;
27            }
28            delta += ($y[g] - last_y)** 2;
29        }
30        if (delta < 0.005 && proc_utilization < 0.95)
31            new_share = share_redistribution(ngroups, groups, y);
32    }
33 }

```

adjustment. In the algorithm, the model is first solved without any requests to the shadow task (all y_i s are set to zero). Then, y_i increases when $U_{(T,i)} > GS_{gi}$ and y_i decreases when $U_{(T,i)} < GS_{gi}$ and if $y_i > 0$. After new values of y_i for all the groups are calculated, the model is re-solved using these new values in the next iteration by invoking `solve()` function. When the total change of y_i for all groups between two successive iterations, is substantially small, the algorithm stops. The criteria of convergence of the LQX-DPS algorithm is the sum of the squares of Δy_i for all groups between two successive iterations, Δ in (7.8), is less than 5×10^{-5} .

$$\Delta = \sum_i^G (\Delta y_i)^2 \quad (7.8)$$

Using this algorithm, the utilization of all the groups are eventually less than their group shares when the algorithm stops. Therefore, this algorithm is sufficient in solving the models:

1. which have cap shares only; or,
2. which have guarantee shares and the FS processor is fully utilized.

If the FS processor is fully utilized, the utilization of all groups cannot exceed their shares even though they have guarantee shares. Therefore, guarantees have the same effects as caps when the FS processor is fully utilized.

7.2.3 Redistributing Surplus Shares

If the FS processor has some spare resources, a group with a guarantee share may have a chance to consume some extra resource. How many extra shares a group can

get, is proportional to its original share using (7.9).

$$GS^- = \sum_i^{i \in \mathcal{P}} (GS_i - U_i) \quad (7.9)$$

$$GS^+ = \sum_i^{i \in \mathcal{Q}} GS_{(g,i)} \quad (7.10)$$

$$GS'_{(g,i)} = GS_{(g,i)} \times \left(1 + \frac{GS^-}{GS^+}\right) \quad (7.11)$$

where: GS^- is the total surplus shares that can be redistributed;

GS^+ is the total guarantee shares that can be increased;

$\mathcal{P}$ is a set of groups that are able to contribute their spare shares;

$\mathcal{Q}$ is a set of groups that may receive additional shares;

For any groups in the set $\mathcal{P}$, their group shares can be either guarantees or caps while all groups in the set $\mathcal{Q}$ have guarantee shares only. Also, all the groups in set $\mathcal{P}$ have some spare shares and must not have any delay at the delay server ($y_{gi \in \mathcal{P}} = 0$). Conversely, all the groups in set $\mathcal{Q}$ must have some delays at the delay server, which indicates these groups definitely need more resources.

(7.9) gives the total surplus shares that can be redistributed, which is the total of the differences between group utilization and group shares of all the groups in the set $\mathcal{P}$, while (7.10) gives the total guarantee shares of all the groups in set $\mathcal{Q}$. Therefore, the fraction of $\frac{GS^-}{GS^+}$ in (7.11) defines how many extra shares a group may receive.

The surplus share is redistributed by using Listing 7.2, and it is invoked through line 30 and line 31 in Listing 7.1. The condition at line 22 in Listing 7.2, and the second condition at line 30 in Listing 7.1 are aimed to remove unnecessary executions and reduce the total number of iterations. The condition at line 30 in Listing 7.1 also makes sure that a converging trend appears in the execution of Listing 7.1 before it enters the execution of Listing 7.2 to adjust some guarantee shares. This will prevent

Listing 7.2: LQX function for surplus share redistribution

```

1 function share_redistribution(
2     ngroups, /* int */
3     groups[] /* Group */
4     y[]      /* double */)
5 {
6     new_share[ngroups];
7     share_can_increase = 0.0;
8     surplus_share = 0.0 ;
9     can_increase[ngroups]; /* Boolean */
10    for(int g=0; g<ngroups; g++){
11        can_increase[g]=False;
12        new_share[g]=groups[g].share;
13        if ( (groups[g].utilization < groups[g].share)
14             && (y[g]<=0) ){
15            surplus_share += groups[g].share-groups[g].utilization;
16        }else if ((groups[g].utilization>groups[g].share)
17                 && !(groups[g].isCap) ){
18            share_can_increase += groups.share;
19            can_increase[g] = True;
20        }
21    }
22    if (share_can_increase > 0.05 ){
23        unit_share = surplus_share /share_can_increase;
24        for(int g=0; g<ngroups; g++){
25            if( can_increase[g] ){
26                new_share[g] = groups[g].share *(1+ unit_share);
27                if (new_share >1)
28                    new_share = 1.0;
29            }
30        }
31    }
32    return (new_share);
33 }

```

the process from oscillation in large steps, which may lead to no convergence.

7.2.4 Verification LQX-DPS

This section presents three test cases to verify the two algorithms presented in the previous section. The results found by DPS are compared with the results of simulations, which run in the confidence interval of 95%, $\pm 2.0\%$.

All the test cases in this section use the same model. This model contains three classes of clients, $c1$, $c2$ and $c3$, which make requests to one of server tasks $s1$, $s2$ and $s3$. These server tasks are running at a FS processor *ApplicCPU*. Task $s1$ belongs to *Group1* while tasks $s2$ and $s3$ belong to the other group, *Group2*. The transformed model of this model is shown in Figure 7.2.

Figure 7.2: A test model with 3 tasks in two groups.

7.2.4.1 Case 1: Cap Shares

In *Case1*, all group shares are cap shares. The experiments are performed by varying group shares GS_1 and GS_2 from $(0.2, 0.8)$ to $(0.8, 0.2)$ by a step of $(0.2, -0.2)$. The results of LQX-DPS are presented in Table 7.1. In the table, GU_1 and GU_2 are group

utilizations of *Group1* and *Group2*; PU_{s1} , PU_{s2} and PU_{s3} are processor utilizations of the server tasks $s1$, $s2$, and $s3$; and U_{proc} is the total utilization of the FS processor, *ApplicCPU*. In Table 7.1, all the group utilizations are very close to their cap shares, with a difference rate $(\frac{|GS_i - GU_i|}{GS_i} \times 100)\%$ less than 0.1%. Table 7.2 shows the relative error rates $(\frac{DPS - SIM}{SIM} \times 100)\%$ of the throughput of the three client tasks against simulations. All the relative error rates are less than 2%.

Table 7.1: Case 1: Group utilization.

(GS_1, GS_2)	GU_1	GU_2	U_{Proc}	PU_{s1}	PU_{s2}	PU_{s3}
(0.20, 0.80)	0.1977	0.7957	0.9934	0.1977	0.4058	0.3898
(0.40, 0.60)	0.3969	0.5941	0.9910	0.3969	0.3007	0.2934
(0.60, 0.40)	0.6003	0.4007	1.0010	0.6003	0.2014	0.1993
(0.80, 0.20)	0.8001	0.2008	1.0009	0.8001	0.1006	0.1002

Table 7.2: Case1: Relative errors (%) compared to simulation.

(GS_1, GS_2)	λ_{c0}	λ_{c1}	λ_{c2}
(0.20, 0.80)	-1.34	-1.76	0.82
(0.40, 0.60)	-0.84	-2.14	0.88
(0.60, 0.40)	0.08	-0.10	0.99
(0.80, 0.20)	0.11	1.27	-0.27

The results of *Case1* in Table 7.1, clearly state the basic properties about FS scheduling with cap shares:

1. The utilization of a group cannot exceed the cap share of this group.
2. Tasks within a group evenly divide the group share.

7.2.4.2 Case 2: Guarantee Shares Without Spare Resources

In *Case2*, all group shares are guarantees, and the experiments are carried out in the same way as *Case1*. Table 7.3 presents a set of results which are exactly the same as the results in Table 7.1 in the previous *Case1*. This is not surprising because Listing 7.1 calculates and adjusts the number of visits to the shadow task to fit their

group shares, and stops due to the processor is fully utilized. This test case shows another fact that guarantees and caps have the same effects on a fully utilized FS processor.

Table 7.3: *Case2*: Group utilization.

(GS_1, GS_2)	GU_{s1}	GU_{s2}	U_{Proc}	PU_{s1}	PU_{s2}	PU_{s3}
(0.20, 0.80)	0.1977	0.7957	0.9934	0.1977	0.4058	0.3898
(0.40, 0.60)	0.3969	0.5941	0.9910	0.3969	0.3007	0.2934
(0.60, 0.40)	0.6003	0.4007	1.0010	0.6003	0.2014	0.1993
(0.80, 0.20)	0.8001	0.2008	1.0009	0.8001	0.1006	0.1002

7.2.4.3 Case 3: Guarantee Shares with the Existence of Surplus Resources

In order to make some spare resource appear, the test model is changed slightly from *Case2*. The number of customers of client $c1$ is reduced to 1, so Group1 is not able to consume all the resources it is entitled to in some experiments. When $GS_1 \geq 0.4$, some spare resources appear due to the low request rate to *Group1*. *Group2* consumes these surplus resources, and the two tasks $s2$ and $s3$ in *Group2* consume the same amount of the extra resource. Figure 7.3 shows group utilization found by DPS and simulation. In the figure, there are quite a significant difference between DPS and simulations after surplus shares appear.

The results in Figure 7.3 not only show the problem in the redistribution of surplus shares but also shows the hint to overcome the problem. The redistribution of the surplus shares not only results in some groups receiving extra shares but also increases the priorities of these groups. Consequently, the groups contributing surplus share are not able to obtain as many shares as before. For example, when *Group1* has a share of 0.6, the simulation results show that *Group1* can get a utilization of about 0.5, but the DPS results shows that *Group1* only can get a utilization of about 0.4. These results indicate that priority scheduling may be applied to the transformed processor

Figure 7.3: Results of shares redistribution.

to maintain a higher priority of the contributing groups after they contribute surplus shares.

7.3 Integrating DPS into LQNS

As stated previously, LQX-DPS iterations take place at the outermost iterations of the LQNS analytic solution. In each LQX-DPS iteration, the entire model is (re)solved by LQNS. By integrating DPS functions into LQNS, (referred to as LQNS-DPS [136] functions), the LQNS-DPS functions are located in a certain LQN submodel. Therefore, the LQNS-DPS iterations only take place when solving this particular submodel.

7.3.1 Mapping Dynamic Parameters in LQNS

The key point of handling FS processors is calculating the *extra* delays of the groups because they have insufficient group shares. Inside the LQNS solver, the extra delays of groups at a FS processor are modelled by surrogate delays [108] attached to LQN *phases*, which belong to the client tasks in groups. The reason is that in LQN models,

the behaviour of a task consuming a physical resource is modelled by its phases making requests to the device entries.

All these extra delays are consumed at a special ‘think’ server, which is commonly used by any client tasks who have think time or some surrogate delays within the same submodel. This *think* server is equivalent to the shadow task in the LQX-DPS approach in term of the FS processor. Therefore, the step of transformation of a FS processor is not required in the LQNS solver.

The extra delay of each group becomes a dynamic parameter, $\$D_i$, whose value equals the second term, $\$y_i \times S_{D,i}$, in (7.3.b), which is used in the previous LQX-DPS approach. The mean residence time of a group at the FS processor plus its extra delay, then becomes the true mean residence time at the FS processor.

Based on (7.4) $\sim$ (7.7), used in the previous DPS-LQX approach previously, the substitution function for the dynamic parameter $\$D_i$ then becomes (7.12) when U_i is greater than GS_i , and (7.13) when U_i is smaller than GS_i (Case 3, Page 191).

$$\begin{aligned} \Delta W_i &= \frac{(U_i - GS_i)}{U_i} \times W_i \\ \$D_i^{(n+1)} &= \$D_i^{(n)} + \Delta W_i \end{aligned} \quad (U_i > GS_i) \quad (7.12)$$

Where: W_i and U_i are the mean residence time and processor utilization of a group i at the FS processor.

$$\begin{aligned} \Delta W_i &= \min\left(\frac{GS_i - U_i}{U_i} \times W_i, \$D_i^{(n)}\right) \\ \$D_i^{(n+1)} &= \$D_i^{(n)} - \frac{\Delta W_i}{2}; \end{aligned} \quad (U_i < GS_i) \quad (7.13)$$

7.3.2 The Implementation in LQNS

To perform the substitutions of dynamic parameters in LQNS, two extensions are made to the LQN analytic solution.

1. A new type of LQN submodel, *FairShareSubmodel* is created as a subclass of

MVASubmodel and used to handle the DPS iterations in LQNS.

2. A new type of queueing station, *FairShareServer* is created as a subclass of FCFS queue, and used to handle the priority issue of groups.

7.3.2.1 FairShareSubmodel

A *FairShareSubmodel* contains, at least, one FS processor. The algorithm for solving a *FairShareSubmodel* is shown in Listing 7.3. The solution of *FairShareSubmodel* consists of two separate iterations: DPS iterations and MVA iterations. The MVA iterations are the inner iterations and are invoked in every DPS iteration. Therefore, *FairShareSubmodel* is eventually solved by the MVA iterations, which it inherits from MVASubmodel.

In each DPS iteration, which is shown in the lines 3~7 in Listing 7.3, *FairShareSubmodel* performs the substitutions of dynamic parameters using the substitution functions (7.12) and (7.13). These substitution functions are similar to (7.4) ~ (7.7), used in the previous LQX-DPS approach, except the dynamic parameters for groups are the extra delays instead of the mean number of visits to the shadow task. The new values are used in the following MVA iterations. The results of an MVA solution, including the utilization of all the groups, are saved and are fed into the substitution functions to find out the extra delays of the groups in the next iteration. The DPS iterations stop when the DPS convergence test is satisfied.

Since the values of the dynamic parameters are varied in every iteration during the solution, the LQNS-DPS approach may have a convergence problem. In the LQNS solver, the DPS iterations are controlled by `LQNS-DPS()` in line 4 in Listing 7.3. The algorithm of controlling the DPS iterations is shown in Listing 7.4, which is a detailed algorithm for Listing 7.3 from line 2 to line 7.

In the algorithm, the DPS iterations can be grouped into five steps:

1. Step 1 is the first DPS iteration. In this step, the submodel is solved like a

Listing 7.3: FairShareSubmodel Solution Algorithm.

```

1  begin
2      reset groups;
3      do:
4          LQNS-DPS();
5          Solve FairShareSubmodel s using MVA;
6          Save group utilization;
7      until{ DPS convergence}
8
9      Save waiting times for FairShareSubmodel s;
10     Set think times for submodel s+1;
11 end

```

regular MVASubmodel without any added-on extra delays. The intermediate results of this step, i.e., the utilization of groups (`first_step_util`), determine which groups need to be adjusted because they used more resource than they are entitled.

2. Step 2 is the substitution step. The DPS substitution functions are invoked prior to the MVA iterations. This step takes the most number of the DPS iterations. This step is finished when the utilizations of all the groups, `second_step_util`, are not greater than their group share (regardless of guarantees or caps). The results of this step give the group utilizations when all groups are capped. On the other hand, the results also show whether surplus shares exist. If no surplus share exists, the DPS iterations stop. All guarantee shares have the same effect as cap shares on a fully utilized resource. Otherwise, DPS iterations enter Step 3.
3. Step 3 is for redistributing surplus shares. The redistribution algorithm for surplus shares is the same as the one used in the previous DPS-LQX approach. According to the amount of surplus shares and the weights of their original group shares, some groups will be assigned an increased group share. The results of step 3 determine how much extra resource a group with a guarantee

Listing 7.4: LQNS-DPS algorithm

```

1  begin
2      long dps_iter = 0;  /* initialize*/
3      double dps_delta = 0.;
4      bool _dps_converage = false;
5
6      if (step != 5):
7          reset_groups();
8      do:
9          dps_iter ++;
10
11         if (step == 2 or step == 4):
12             dps_delta += group_adjust() ;
13         else if (step == 3):
14             redistribute_share();
15             group_reset();
16             dps_delta += group_adjust() ;
17         endif
18
19         MVA_solve(); /* solve the submodel by the MVA iterations*/
20
21         if (step == 1):
22             save first_step_util;
23         else if (step == 5):
24             _dps_converage = true; /* stops the DPS iteration */
25         else:
26             _dps_converage = dps_converge_test( dps_delta) ;
27         endif
28
29         step = check_dps_step(); /* update step */
30     until{ _dps_converge}
31     save MVA results;
32     save group utilization;
33 end

```

share may obtain. Then DPS iterations enter Step 4.

4. Step 4 is also a substitution step, which is almost same as Step 2, except the enlarged group shares are used in the substitution functions (7.12)~ (7.13) instead of their original group shares. Step 4 also ends when all group utilizations are not greater than their increased group share. The results of Step 4 provide the group utilizations which satisfy the increased group shares. At the end of Step 4, if there still are some surplus shares remaining, the DPS iterations go back Step 3 again. Otherwise, the DPS iterations enter Step 5.
5. Step 5 is the final step. In this step, all the DPS operations are disabled, and all the dynamic parameters are not changed. *FairShareSubmodel* is solved as a regular MVASubmodel. The purpose of this step is to assure the convergence of the entire model.

7.3.2.2 FairShareServer

FairShareServer is a new type of queue which is specially used to handle the priority issue of ‘contributing’ groups at a FS processor. It is a subclass of *FCFSServer* (a FCFS queue) in the LQNS solver. The characteristics of a *FairShareServer* are:

1. *FairShareServer* is a FCFS queue when *NO* group is contributing its extra share.
2. *FairShareServer* is a priority queue with preemption when one or more groups are contributing their unused shares, *AND* some groups are consuming these extra shares.

In the first case, all groups have the same priority when they queue at the resource, therefore, tasks in any groups are served as a FCFS order.

In the second case, some surplus shares exist. The groups that are consuming the extra share must have extra arrivals at the FS processor to use the remaining available

resource. The extra arrivals may increase the queueing delays of the requests from the contributing groups. Therefore, to eliminate extra queueing delays, the requests from the groups contributing the extra share must have a higher priority than the requests from the groups receiving the extra share. If a request (from the contributing groups) arrives at the resource after these extra arrivals (from the consuming groups), the request will preempt these customers that are in front of it.

In the LQN analytic solution, handling group priority is realized by applying the ‘maximum queueing delays’ to the contributing groups. In fact, the queueing delays occurred when all groups are capped are the maximum queueing delays of all the contributing groups. These maximum queueing delays can be found at the end of Step 2 of the DPS iterations. These maximum queueing delays are passed into MVA iterations in Step 4 of the DPS iterations as the bounds of the queueing delays to limit the increment of the residence time of the requests from the contribution groups.

7.3.2.3 Convergence Control

DPS-LQNS may fail the convergence test or need a long time to converge if the dynamic parameters are varied in every DPS iteration. The following measures for convergence control prevent the failure of the convergence test and reduce the number of iterations at different levels.

1. The convergence criteria for both Step 2 and Step 4 is that the sum of the squares of ΔU_i for all groups between two successive **DPS iterations**, Δ in (7.14), is less than 5×10^{-5} .

$$\Delta = \sum_i^G (\Delta U_i)^2 \leq 5 \times 10^{-5} \quad (7.14)$$

This is the convergence test which is performed at each end of DPS iteration at line 7 in Listing 7.3.

2. Adding Step 5 ensures the convergence of the entire model. All the dynamic parameters are using finalized values and all DPS operations are ignored. This step can be reached from Step 2 and Step 4 after they converge. The condition for entering Step 5 is that the sum of the squares of ΔU_i for all groups between two successive **submodel iterations**, Δ in (7.14), is less than 5×10^{-5} .

7.3.3 Verification of the LQNS-DPS Approach

In order to verify the correctness of the LQN-DPS approach, two sets of models are used in the section. First, the models in the previous section are resolved to demonstrate that the LQNS-DPS approach overcomes the LQX-DPS approach to solving the group priority issues when surplus shares exist. The second set of models are randomly generated models, which represents a larger-scaled and more complex case. Both sets of models show the correctness of the LQNS-DPS approach.

7.3.3.1 Resolving the Previous Model

In order to verify the correctness of a FairShareServer queue, the model in Section 7.2.4.3 is resolved. The group shares vary from 0.1 to 0.9 (Group 1) and from 0.9 to 0.1 (Group 2). The model was solved multiple times under the following four configurations:

- (a) both groups have cap shares;
- (b) both groups have guarantee shares;
- (c) Group 1 has a cap share and Group 2 has a guarantee share;
- (d) opposite to (c) only Group 2 is capped.

In this model, the FS processor has spare resources when Group 1 has a share more than 0.5 because Group 1 does not have enough incoming traffic to consume the entire share it is entitled to. On the contrary, Group 2 always consumes all the available resource. In configurations (a) and (d), Group 2 is prevented from consuming the

spare resource since it is capped. Therefore, a larger group share for Group 1 starves the FS processor. On the other hand, in configurations (b) and (c), the FS processor has surplus shares when Group 1 has a share more than 0.5. These surplus shares are consumed by Group 2, then the FS processor becomes fully utilized again. The experiments for this model are able to cover all the situations of the competitions between these two groups at the FS processor.

Figure 7.4: Results from LQNS-DPS and LQSIM.

The results for these four configurations by solving from the LQNS-DPS approach are shown in Figure 7.4, as well as the results from the LQX-DPS approach and simulation. The two analytic DPS approaches have the same accuracy when all groups are capped or there is no surplus share existing for the groups with guarantee

shares. Since LQX-DPS has a priority issue when surplus shares exist, which are the cases when the share of Group 1 is more than 50% in the configurations of (b) and (c), the results of the previous LQX-DPS are incorrect. The LQNS-DPS approach overcomes the priority issue and produces much better results than LQX-DPS.

Table 7.4: The execution time of the LQNS-DPS and simulations.

Type of group shares	LQNS-DPS execution (s)	LQSIM execution (s)
both cap shares	0.012	4.236
both guarantee	0.010	3.667
group 1 capped	0.011	3.346
group 2 capped	0.009	3.787

The execution times of the LQNS-DPS and simulation are shown in Table 7.4, which clearly state that the analytic method, LQNS-DPS, is significantly faster than simulations.

7.3.3.2 Randomized Model

In this section, a randomly generated LQN model is used to verify the LQNS-DPS approach. This random model was generated by `lqngen` [77], which is one of the LQN tools. Using `lqngen`, random models can be generated by specifying the number of layers, number of reference tasks, total number of tasks or processors, as well as the shape of system models, such as pyramid (more servers at lower layers), funnel, etc.

The random model, which is shown in Figure 7.5, has three reference tasks ($c0$, $c1$ and $c2$) and eight server tasks ($t0 \sim t7$), which are located in three layers. These server tasks are hosted by two processors, $p0$ and $p1$. All the input parameters are also randomized, the input file of this random model (`random_model.lqnx`) is in Appendix 1 A.

The processor $p1$ is a FS processor and hosts six server tasks, $t0$, $t3$, $t5$, $t7$, and $t1$, $t2$, which are in the two groups, *Group1* and *Group2*. Two sets of experiments were performed using the pairs of group shares (0.6, 0.4) and (0.2, 0.8). Each experiment

Figure 7.5: A randomly generated LQN model.

has 50 sets of random parameters.

The first set of experiments verifies the correctness of the LQNS-DPS approach with all cap shares. In this set of experiments, two groups are capped and 50 different parameters were used. Tables 7.5 and 7.6 show the difference of the results of LQNS-DPS compared to the simulation results with group shares (0.6, 0.4) and (0.2, 0.8) respectively. All the simulation results found by LQSIM, are with a confidence interval of $95\% \pm 2.0\%$. The values in the row labelled “mean” are the mean values of the difference between LQNS-DPS and LQSIM; while the values in the row of “sdev” are the standard deviations of these differences. The mean error rates on average are no worse than 5% of the results found from simulation over 100 randomly parameterized

models.

Table 7.5: Results for the random model with ($CS_{Group1} = 0.6$, $CS_{Group2} = 0.4$).

% Δ	U_{Group1}	U_{Group2}	$Proc_Util$
Group Share: (0.60,0.40)			
mean	-1.61	4.91	0.84
sdev	3.07	7.41	3.52

	$Group1$				$Group2$	
% Δ	U_{t0}	U_{t3}	U_{t5}	U_{t7}	U_{t1}	U_{t2}
Group Share: (0.60,0.40)						
mean	-0.20	-2.16	-2.15	-2.16	3.41	3.49
sdev	3.55	3.18	3.15	3.19	13.20	6.39

% Δ	λ_{C0}	λ_{C1}	λ_{C2}
Group Share: (0.60,0.40)			
mean	-0.15	3.29	3.53
sdev	3.57	13.21	6.33

Table 7.6: Results for the random model with ($CS_{Group1} = 0.2$, $CS_{Group2} = 0.8$).

%Δ	U_{Group1}	U_{Group2}	$Proc_Util$
Group Share: (0.20,0.80)			
mean	2.65	2.80	2.72
sdev	5.10	0.59	1.10

	$Group1$				$Group2$	
%Δ	U_{t0}	U_{t3}	U_{t5}	U_{t7}	U_{t1}	U_{t2}
Group Share: (0.20,0.80)						
mean	3.92	2.11	2.24	2.16	3.23	2.53
sdev	5.13	5.22	5.19	5.25	1.41	0.87

%Δ	λ_{C0}	λ_{C1}	λ_{C2}
Group Share: (0.20,0.80)			
mean	3.98	3.22	2.55
sdev	5.11	1.35	0.91

The second set of experiments verify the LQNS-DPS approach with the existence of spare shares, particularly for verifying the correctness of FairShareServer queues. In this set of experiments, only *Group2* is capped at 0.8, so *Group1* has a guarantee share of 0.2; the mean number of customers in *Group2* is reduced. Therefore, *Group2* may not have enough workload to consume its large share of 0.8, so a spare share appears and can be consumed by *Group1*.

Table 7.7: Results for the random model with ($GS_{Group1} = 0.2$, $CS_{Group2} = 0.8$).

$\% \Delta$	U_{Group1}	U_{Group2}	$Proc.Util$
Group Share: (0.20,0.80)			
mean	-1.93	2.25	1.17
sdev	6.85	0.95	1.68

	<i>Group1</i>				<i>Group2</i>	
$\% \Delta$	U_{t0}	U_{t3}	U_{t5}	U_{t7}	U_{t1}	U_{t2}
Group Share: (0.20,0.80)						
mean	-0.47	-2.49	-2.38	-2.45	5.87	1.31
sdev	7.07	6.85	6.85	6.84	3.09	1.11

$\% \Delta$	λ_{C0}	λ_{C1}	λ_{C2}
Group Share: (0.20,0.80)			
mean	-0.56	5.86	1.31
sdev	7.10	3.05	1.12

Table 7.7 shows the mean error and standard deviation for comparing LQNS-DPS and simulation over 50 different sets of parameters. The mean error rates of the group utilizations and the processor utilization are around 2%. The maximum mean error rate of task utilizations is 5.79 only at task $t1$, and for all the other tasks, the mean error rates are around 2~3%.

Table 7.8: The execution time of the LQNS-DPS and simulations.

Group shares	LQNS-DPS (Sec.)	LQSIM (Sec.)
$(CS_{g1} = 0.6, CS_{g2} = 0.4)$	0.1061	33.827
$(CS_{g1} = 0.2, CS_{g2} = 0.8)$	0.1178	47.969
$(CS_{g1} = 0.2, GS_{g2} = 0.8)$	0.1173	43.689

As for the computation effort, the average execution times of these experiments on randomized models for LQNS-DPS and simulation are shown in Table 7.8. Again, the analytic method, LQNS-DPS, is 400 times faster than simulations.

7.4 Summary

LQN models with processors with FS scheduling were handled only by the LQN simulator previously. This chapter revisits these LQN models and solves them analytically

by adopting the method of dynamic parameters substitution into LQN models by two implementations. First, in order to apply the DPS method by LQX functions, a LQN model with a FS processor has to be transformed into a model containing a ‘normal’ processor (FCFS or PS scheduling) and a shadow delay task. The transformed processor was solved by MVA (inside LQNS). The shadow task was used to simulate the suspension of the groups after they exhaust their shares. The numbers of visits to the shadow task were dynamic parameters, which were solved by DPS substitution functions using LQX. This LQX-DPS algorithm produces accurate results when all group shares are caps, or where a FS processor is fully utilized. However, when some surplus shares appear, LQX-DPS has a group priority issue.

Next, the DPS method was integrated into the LQNS solver directly. The implementation was realized by a newly created *FairShareSubmodel* and *FairShareServer* queue. This LQNS-DPS solution produced the same accuracy compared to the LQX-DPS approach, when no surplus share exists. The LQNS-DPS solution solved the group priority issue by using a *FairShareServer* queue, and produced accurate results during the verification experiments.

The advantages of LQNS-DPS over the LQX-DPS are: first, the LQN-DPS iterations are inside the submodel iterations. Therefore, the LQN-DPS iterations only take place at certain submodels rather than over the entire model, like the LQX-DPS functions do. Next, LQNS-DPS does not transform the FS processor, so LQNS-DPS does not increase the number of tasks. Finally, LQNS-DPS is able to solve models with surplus share redistribution.

Chapter 8

Conclusions

This research focused on improving the accuracy of the Layered Queueing Network solver (LQNS) and enhancing the modelling power of LQN models to handle decisions. In this section, conclusions of these two research activities are given in the order of the contributions, listed in Chapter 1. Future work also is addressed.

8.1 Handling Traffic Dependencies in LQN

In general, the LQN analytic solution, LQNS, is an accurate, fast and scalable solution to solve many distributed systems with a client-server architecture. However, with the existence of traffic dependencies, the accuracy of LQNS is affected. The main reason is that LQNS uses the method of decomposition to decompose the entire LQN network into a set of submodels, then solve each of these submodels individually and independently using MVA solutions. During the iterations, the results of the upper-level submodels affect the lower-level model in terms of thinking time of the clients, while the lower-level queueing delays are counted as a part of service demand of the upper-level servers.

There are two major traffic dependencies, *Interlocks* and *Sub-chain dependencies*, which may occur in LQN models. The follow sections conclude how these two kinds

of dependencies were handled in LQN models to improve the accuracy of the LQN analytic solution.

8.1.1 Interlocking

The previous analytic solution, LQNS, handled some forms of interlocking dependencies. However, it can suffer from large errors due to the following:

1. The waiting time from contention at the interlocked server in the lowest submodel was not distributed properly among the layers; too much queueing delay was introduced at the immediate servers, where the interlocked flows occur, and insufficient delay to the interlocked servers.
2. The calculation of the Interlocking probability $\text{Pr}(\text{IL})$ was a simple ratio based on the number of customers in an interlocked path.
3. The interlock adjustment did not properly account for *external contention* at either the interlocked server (at the lowest level), or at any intermediate server, referred to as *mixed flows*.

The new approximation presented in this work reduces the amount of errors in all the cases, which were recognized and handled previously. The new approximation is achieved by the following:

1. Interlock adjustments are performed both at the lower-layer submodel where the interlock flows cause false contention and in the upper-layer submodels where contention at lower-level submodels has too large an effect on the contention at higher levels. This improvement is the essential way to correct the misdistribution of queueing delays among the layers.
2. The new approximation corrects for interlock when there is one less customer j in the queue ($L_{mk}(\mathbf{N} - \mathbf{e}_j)$) rather than at the full customer population

$$(L_{mk}(\mathbf{N})).$$

3. The new approach estimates the interlock probability far more accurately than the naïve approach used previously.

In addition to the impacts of external contentions, this research also recognized more cases of interlocking dependencies, for example, the *Type2* and *Type3* sending interlocking effects, which are hidden more deeply. The new approximation is also able to correct these interlocking effects after they are properly recognized by the solver.

Furthermore, interlocking effects in the models with second-phase services were also addressed. *Second-phase* is used to model the aggressive servers with early replies in LQN models. Although the existence of second-phase service, in general, dilutes the interlocking effects, it also brought some new concerns, for example,

1. the influence of the ratio (α) of the second-phase service on interlocking effects;
2. the models with one customer at the common source have an additional customer at their interlocked servers;
3. phase-two services may behave specially at their device submodel.

These matters were addressed in this research, during the explanations of these two situations: the interlocked server has second-phase service and the intermediate tasks have second-phased service. The calculations of interlock probabilities ($\text{Pr}(\text{IL})_{ph1}$ or $\text{Pr}(\text{IL})_{ph2}$) were re-evaluated depending on phases, but they are based on the interlocking probabilities ($\text{Pr}(\text{IL})_s$) used in single-phase models.

The new solution was extended to incorporate these new approximations of interlocking probabilities at a multi-phased server. The integrated new approximation is used to handle all cases of interlocking dependencies. The computational complexity of the new approach does not increase much, so it does not adversely affect the solution speed of the solver.

To verify this integrated solution, this research enumerated all the identified interlocking dependencies and discussed in the groups of split interlocks, mixed flows and external contentions, sending interlocks and the interlocking impacts on the LQN models with aggressive servers with early replies (second phase servers).

In the test cases presented in this research, the results from solving the model using the new algorithm are always superior to the results found using the previous approach. Furthermore, the new calculation does not produce infeasible utilizations, unlike the previous approach.

8.1.2 Sub-chain Dependencies

The sub-chain dependency is another kind of traffic dependencies in layered queueing networks, which arises when a multi-entry server has imbalanced loads to each of its provided services, and the server has to make further requests to some lower-layer servers to perform these services. The phenomenon of a sub-chain dependency results in extra queueing delays to the sub-chains with fewer customers. The most significant case is when a sub-chain, which belongs to a multi-server, has one customer only. The impact of sub-chain dependencies decreases as the maximum number of customers of a sub-chain is approaching to the multiplicity of its owner task.

The impacts of sub-chain dependencies were removed by finding the correct maximum possible number of customers from a sub-chain dependent client arriving at a server station. The corrected maximum number of customers is used to estimate the queue length with one customer removed in Bard-Schweitzer AMVA [186]. The new approach for correcting sub-chain dependencies improves the accuracy of the mean queueing delay at both task servers and processors. Furthermore, since LQN Activities are used particularly to model internal concurrency, an imbalanced number of customers arriving at a multi-server task with activities, were also corrected.

Since the maximum number of customers for any sub-chain is calculated once,

so the computational complexity of the new approach does not adversely affect the solution speed of the solver either. All the results produced by the improved solution had a better accuracy than the previous solution.

8.1.3 Case study

The case study presented in Chapter 5 is an online bookstore. This bookstore model has many occurrences of both interlocking and sub-chain dependencies. The improved sub-chain calculation and the new approximation were integrated and applied in this bookstore model. The improved solution produced much better results.

8.2 Handling Decisions in LQN

Pure queueing-based solutions to handle the state-dependent decisions are not possible because all of the parameters to a queueing network must be known before the model is solved. Consequently, state-based solutions, simulation, or combinations of these two techniques with a queueing model have been employed in the past. Layered Queueing Networks as an extended queueing network also have difficulties in handling decisions.

The significance of addressing decisions in LQN is that allowing the LQN technique to model and solve the performance models with state-based behaviour. Also, solving decisions analytically using the DPS method, overcomes to many of other techniques in terms of a fast and scalable solution. This research has shown how to handle decisions in LQN models through dynamic parameter substitution, and simulation.

8.2.1 LQN Decision Patterns

This research was focused on the decisions that commonly made in exception handling. These decisions are defined as five LQN decision patterns: queue, abort,

infinite-retries, finite-retries and timeout patterns. The first four decisions were based on Wu's pattern [224]. Wu's 'queue' pattern is a decision of waiting forever; therefore, it does not make any decision during executions. 'Queue' pattern is not considered further in this research. Additionally, the 'timeout' decision is defined as the fifth LQN decision pattern. In reality, timeout decisions may be the most important and used most commonly.

A template that is used in LQN input files was developed for all the LQN decision patterns. Modellers use this template to specify LQN decisions by simply modifying the values of the input parameters and names of the entries related to decisions.

8.2.2 Handle Decisions in the LQN Simulator

In the simulator LQSIM, these four non-trivial decisions were generalized further into two decision patterns: *Timeout* and *Retry*. The reason is that an abort pattern can be treated as a special case of a timeout pattern when the timeout limit is zero while the infinite retry is a special case of finite retries patterns. Therefore, only two generalized decisions were implemented in the simulator LQSIM.

The accuracy of LQSIM results was verified by a Petri net tool, GreatSPN [47, 48], using a small sized web server model, first. Then, LQSIM was also verified by some existing results of a larger web server database model from [224], which were produced by simulation (CSIM). All the verification show that LQSIM can produce accurate results on decision models with different sizes.

The implementation in LQSIM does not only enhance LQN models being able to handle decisions, but also, it is able to provide accurate results which can be used to verify the analytic solutions using DPS.

8.2.3 Applying DPS Method to Handle Decisions

Dynamic Parameter substitution takes advantage of the very method used to typically solve a layered queueing network analytically: iterative solution among multiple submodels. With this approach, the results of the solution of a submodel are used as input to other submodels. Dynamic parameter substitution extends this method to allow arbitrary output to be fed back as input for subsequent iterations of the model. This method was implemented using the LQX [150] language for the three non-trivial LQN decision patterns (exclude the finite-retries decisions).

Similarly, the results of the DPS method were verified by the results of the Great-SPN using the same small web server model, first. Then, the DPS method was compared against the existing results found using the HPMM [224] and simulation (CSIM) for the exception patterns abort and infinite retry, upon the model which emphasized the exceptions while acquiring a lock on a web database. From the results here, DPS performs as well as HPMM in solving the model. As for the timeout pattern, DPS was compared to the simulation results from LQSIM. Again, the analytic results compare favourably to simulation.

The DPS solution is quite efficient. Since all computation is performed within the context of a single operating system process, no file input/output or process switching is required. This could be of tremendous benefit if the hybrid approach's solution is dominated by overhead because the infinite-retry model required a substantial number of 'feedback' iterations to reach convergence. DPS can be improved further by moving the feedback process deeper into the solution hierarchy.

8.2.4 Using DPS to handle FS Scheduling

Processors with FS scheduling [116] in LQN models were previously handled by the LQN simulator only in [132]. This research revisits these LQN models and solves them analytically by using the method of dynamic parameters substitution. The

DPS method for FS scheduling was implemented in two locations, LQX and LQNS, which are referred to as LQX-DPS and LQNS-DPS respectively.

First, in the LQX-DPS implementation, a LQN model with a FS processor has to be transformed into a model containing a ‘normal’ processor (with FCFS or PS scheduling) and a shadow delay task. The transformed processor can be solved by MVA used by the LQNS solver. The shadow task is to simulate the suspension (forced sleep) of the groups after they exhaust their group shares. The numbers of visits to the shadow task are dynamic parameters, which are solved by DPS substitution functions using LQX.

This LQX-DPS algorithm produces accurate results when all the groups’ shares are caps or guarantee shares with a fully utilized FS processor. However, with the existence of surplus shares, the results found by LQX-DPS are quite different with simulation results. The reason is that the algorithm of redistribution of surplus shares actually increases the priority of every group which receives the extra shares. As a consequence, the priorities of the groups contributing their surplus shares are implicitly lowered, so they cannot receive enough share as much as before they contribute.

The implementation inside LQNS, LQNS-DPS overcame the priority issues of the ‘contributing’ groups. In LQNS-DPS, a newly developed server queue *FairShareServer* is used to prioritize groups during the share redistribution. *FairShareServer* ensures a higher priority to the ‘contributing’ groups by removing the extra waiting time caused by the extra arrivals from the ‘consuming’ groups when they are consuming the addition resource. The dynamic parameter substitutions in LQNS-DPS are performed in the *FairShareSubmodel*, a new type of the LQN submodels. *FairShareSubmodel* has DPS iterations that locate in between the submodel iterations and MVA iterations. Therefore, DPS iterations only occur at the *FairShareSubmodel* rather than the entire model, as LQX-DPS does.

The results of LQNS-DPS were verified by LQSIM results in the cases with and

without surplus share using a random model. LQNS-DPS can produce accurate results with less execution time.

8.3 Remarks

The previous solution of LQNS was able to produce accurate results on throughput and server utilizations, and predicts the capacity of systems and quickly locates potential bottlenecks. However, there are some misdistributed queueing delays caused by traffic dependencies. Therefore, this research first focused on handling traffic dependencies to improve the accuracy of the LQN analytic solution. This research and the previous research indicate that traffic dependencies are inevitable when using the method of decomposition. Therefore, this work has presented a fundamental idea for handling traffic dependencies that may be required by other extended Queueing Networks which are solved using the method of decomposition. LQNS used MVA to solve the submodels and previously throughputs for corrections for interlocks. This research showed that removing delay is superior other solution technique that also solve models using decomposition.

This work also enhanced LON models to handle some state-based behaviours (for example decisions made in exception handling) and some special system behaviours (for example fair share scheduling policy). The significance of the research is that it presents a mechanism to handle state-based behaviour in LQN models by using the method of DPS which fully takes advantages of LQNS, i.e., it is both fast and scalable.

Layered Queueing Network models become a generalized performance prediction and evaluation technique with the capabilities of handling traffic dependencies and solving state-based behaviours.

8.4 Limitations and Future Work

8.4.1 Limitations

Regarding using the DPS method to handle decisions, the limitations are:

1. Currently, the DPS method is implemented through LQX, which is referred to as LQX-DPS. This implementation of LQX-DPS prevents further improvements to shorten execution times since the entire model has to be solved (run to completion) in every DPS iteration. LQX-DPS also may have difficulties in handling some special dynamic behaviours, for example, the priority issue arises when using LQX-DPS to solve fair share scheduling during redistributing surplus shares.
2. LQNS-DPS has a convergence control, which is described in Section 7.3.2. However, this convergence control is specially tailored for solving the FairShareSub-model. A general strategy of convergence control needs to be considered when extending LQNS-DPS to handle decisions, such as abort, timeout and retry decisions. In particular, if there are multiple decisions in a LQN model, ensuring that the solver is converging is a challenge.
3. This research only handled a subset of decisions, which are restricted in the context of exception handling during resource allocations. These decisions also have limitations, for example, timeout decisions are restricted to the timeouts on queueing delays.

As for the LQN analytic solution using MVA, the calculation of queueing delays at a server with multiple threads needs to be improved.

8.4.2 Future work

This research only handled fair share scheduling and a subset of decisions commonly made in exception handling. However, the method of DPS provides a possible way to model and solve more complex system behaviours, which cannot be solved analytically by queueing-based models at present. DPS uses feedbacks to alter intermediate results during the solution. Therefore, future work can be focused on:

Develop general LQNS-DPS architecture. First, DPS needs to be integrated into LQNS directly to handle general decisions. As the analytic solution for solving LQN models using fair share scheduling demonstrated, it is feasible to develop an analytic solution which accommodates the DPS method into LQNS directly. LQNS-DPS does not only widen the scope of LQN models but also reduce the execution time of DPS further. This method of DPS is implemented using the LQX language for now, which means that DPS occurs in the outermost iteration. LQNS extensions aim to integrate DPS into the inner iteration of the LQNS solver (between the *Steps 2* and *Step 3* on Page 39). The outermost iteration (used by LQX) will be removed. Therefore, the extra iterations used by DPS only occur in solving the submodels with dynamic parameters, not for solving the entire model, like the current LQX version of DPS does. This should reduce the computation cost compared to the LQX algorithms. Substituting dynamic parameters internally has to be done inside the LQNS solution, which requires modifying the fundamental algorithms of MVA solvers.

Develop more DPS mapping functions to handle more decisions

1. Handling Timeout Decisions on Response Times. So far, the timeout decisions handled in this research only regard queueing delays. The timeout decisions on response time are also commonly used in many web-based systems. The challenging part of timeout decisions on response time is evaluating the probability of the wasted resources by the timed out requests.

2. Handling Finite-Retrieves Decision. Finite-retrieves decisions were handled only in the simulation. The correct substitution function for finite-retrieves has not been found yet. It is necessary to find out this substitution function to solve finite-retrieves analytically in the near future.
3. More research int patterns which were not discussed here.

Bibliography

- [1] Marco Ajmone Marsan, Gianni Conte, and Gianfranco Balbo. A class of generalized stochastic petri nets for the performance evaluation of multiprocessor systems. *ACM Trans. Comput. Syst.*, 2(2):93–122, May 1984.
- [2] Attahiru Sule Alfa and Yu-Fei Shi. A discrete time-limited vacation model for the fair share scheduler. *Telecommunication Systems*, 13(2–4):167–197, July 2000.
- [3] Daniel Amyot. Use case maps as a feature description notation. In *Language Constructs for Describing Features*, pages 27–44. Springer, 2001.
- [4] Daniel Amyot, Xiangyang He, Yong He, and Dae Yong Cho. Generating scenarios from use case map specifications. In *Proceedings of the Third International Conference on Quality Software*, page 108. IEEE Computer Society, 2003.
- [5] Ermeson Andrade, Paulo Maciel, Gustavo Callou, Bruno Nogueira, and Carlos Araújo. Mapping UML sequence diagram to time petri net for requirement validation of embedded real-time systems with energy constraints. In *Proceedings of the 2009 ACM symposium on Applied Computing*, pages 377–381. ACM, 2009.
- [6] Andrea C. Arpaci-Dusseau and David E. Culler. Extending proportional-share scheduling to a network of workstations. In *Proceedings of the International*

- Conference on Parallel and Distributed Processing Techniques and Applications*, pages 1061–1070, Las Vegas, NV, 1997.
- [7] Soheib Baarir, Marco Beccuti, Davide Cerotti, Massimiliano De Pierro, Susanna Donatelli, and Giuliana Franceschinis. The GreatSPN tool: Recent enhancements. *SIGMETRICS Perform. Eval. Rev.*, 36(4):4–9, March 2009.
- [8] David A. Bacigalupo, Jano van Hemert, Xiaoyu Chen, Asif Usmani, Adam P. Chester, Ligang He, Donna N. Dillenberger, Gary B. Wills, Lester Gilbert, and Stephen A. Jarvis. Resource management of enterprise cloud systems using layered queuing and historical performance models. In *Parallel Distributed Processing, Workshops and Phd Forum (IPDPSW), 2010 IEEE International Symposium on*, pages 1–8, April 2010.
- [9] Jos CM Baeten, Twan Basten, Twan Basten, and Michel A Reniers. *Process algebra: equational theories of communicating processes*, volume 50. Cambridge university press, 2010.
- [10] Rusty O. Baldwin, Nathaniel J. Davis IV, Scott F. Midkiff, and John E. Kobza. Queueing network analysis: concepts, terminology, and methods. *Journal of Systems and Software*, 66(2):99 – 117, 2003.
- [11] Simonetta Balsamo, Antinisca Di Marco, Paola Inverardi, and Marta Simeoni. Model-based performance prediction in software development: A survey. *IEEE Transactions on Software Engineering*, 30(5):295–310, May 2004.
- [12] Simonetta Balsamo and Moreno Marzolla. Performance evaluation of UML software architectures with multiclass queueing network models. In *Proceedings of the 5th international workshop on Software and performance*, pages 37–42. ACM, 2005.

- [13] Jerry Banks. *Handbook of simulation: principles, methodology, advances, applications, and practice*. John Wiley & Sons, 1998.
- [14] Jerry Banks, John S. Carson, II, and Barry L. Nelson. *Discrete-event system simulation*. Prentice Hall, 3rd edition, 2000.
- [15] Yonathon Bard. Some extensions to multiclass queueing network analysis. In Matyas Arato, Alexandre Butrimenko, and E. Gelenbe, editors, *Proceedings of the Third International Symposium on Modelling and Performance Evaluation of Computer Systems: Performance of Computer Systems*, pages 51–62. North-Holland, Amsterdam, 1979.
- [16] Forest Baskett, K. Mani Chandy, Richard R. Muntz, and Fernando G. Palacios. Open, closed, and mixed networks of queues with different classes of customers. *Journal of the ACM*, 22(2):248 – 260, April 1975.
- [17] Falko Bause. Queueing petri nets—a formalism for the combined qualitative and quantitative analysis of systems. In *Petri Nets and Performance Models, 1993. Proceedings., 5th International Workshop on*, pages 14–23. IEEE, 1993.
- [18] Falko Bause and Pieter S Kritzinger. Queueing petri nets. In *Stochastic Petri Nets*, pages 203–219. Springer, 1996.
- [19] Matthias Becker, Markus Luckey, and Steffen Becker. Model-driven performance engineering of self-adaptive systems: a survey. In *Proceedings of the 8th international ACM SIGSOFT conference on Quality of Software Architectures*, pages 117–122. ACM, 2012.
- [20] Steffen Becker, Heiko Koziolk, and Ralf Reussner. Model-based performance prediction with the palladio component model. In *Proceedings of the 6th international workshop on Software and performance*, pages 54–65. ACM, 2007.

- [21] Steffen Becker, Heiko Koziolk, and Ralf Reussner. The Palladio component model for model-driven performance prediction. *Journal of Systems and Software*, 82(1):3–22, 2009.
- [22] Heinz Beilner, J Mäter, and C Wysocki. The hierarchical evaluation tool HIT. In *Short Papers and Tool Descript. of 7th Int. Conf. on Modelling Techniques and Tools for Computer Perf. Evaluation*. Citeseer, 1994.
- [23] Simona Bernardi, Susanna Donatelli, and José Merseguer. From UML sequence diagrams and statecharts to analysable petri net models. In *Proceedings of the 3rd international workshop on Software and performance*, pages 35–45. ACM, 2002.
- [24] Simona Bernardi, José Merseguer, and Dorina C Petriu. A dependability profile within MARTE. *Software & Systems Modeling*, 10(3):313–336, 2011.
- [25] Marco Bertoli, Giuliano Casale, and Giuseppe Serazzi. JMT: performance engineering tools for system modeling. *ACM SIGMETRICS Performance Evaluation Review*, 36(4):10–15, 2009.
- [26] Andrew D. Birrell and Bruce Jay Nelson. Implementing remote procedure calls. *ACM Transactions on Computer Systems*, 2(1):39–59, February 1984.
- [27] Alex Bobrek, Joshua J Pieper, Jeffrey E Nelson, JoAnn M Paul, and Donald E Thomas. Modeling shared resource contention using a hybrid simulation/analytical approach. In *Proc. 2004 Design, Automation and Test in Europe (DATE '04)*, volume 2, pages 1144–1149. IEEE, February 16–20 2004.
- [28] Gunter Bolch, Stefan Greiner, Hermann de Meer, and Kishor S Trivedi. *Queueing networks and Markov chains: modeling and performance evaluation with computer science applications*. John Wiley & Sons, 2006.

- [29] Ethan Bolker and Yiping Ding. On the performance impact of fair share scheduling. In *CMG Conference*, volume 1, pages 71–82, Orlando, FL, USA, December 10–15 2000. Computer Measurement Group.
- [30] Ethan Bolker, Yiping Ding, and Anatoliy Rikun. Fair share modeling for large systems: Aggregation, hierarchical decomposition and randomization. In *CMG Conference*, volume 2, pages 807–818. Computer Measurement Group, Anaheim, CA, USA 2001.
- [31] Pere Bonet, Catalina M Lladó, Ramon Puijaner, and William J Knottenbelt. PIPE v2. 5: A petri net tool for performance modelling. In *Proc. 23rd Latin American Conference on Informatics (CLEI 2007)*, San Jose, Costa Rica, October 2007.
- [32] Ray JA Buhr and Ron S Casselman. *Use Case Maps for Object-oriented Systems*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1996.
- [33] Jeffrey P. Buzen. Computational algorithms for closed queueing networks with exponential servers. *Comm. ACM*, 16(9):527–531, September 1973.
- [34] M. D. Byrne and M. Akif Bakir. Production planning using a hybrid simulation-analytical approach. *International Journal of Production Economics*, 59(1–3):305–311, March 1999.
- [35] M. D. Byrne and M. M. Hossain. Production planning: An improved hybrid approach. *International Journal of Production Economics*, 93–94:225–229, January 2005.
- [36] Giuliano Casale. An efficient algorithm for the exact analysis of multiclass queueing networks with large population sizes. *ACM SIGMETRICS Performance Evaluation Review*, 34(1):169–180, 2006.

- [37] Giuliano Casale. CoMoM: Efficient class-oriented evaluation of multiclass performance models. *Software Engineering, IEEE Transactions on*, 35(2):162–177, 2009.
- [38] Giuliano Casale. Exact analysis of performance models by the method of moments. *Performance Evaluation*, 68(6):487 – 506, 2011.
- [39] Giuliano Casale. A generalized method of moments for closed queueing networks. *Performance Evaluation*, 68(2):180 – 200, 2011. Advances in Quantitative Evaluation of Systems QEST 2009.
- [40] Giuliano Casale and Giuseppe Serazzi. Quantitative system evaluation with java modeling tools. In *Proceedings of the 2nd ACM/SPEC International Conference on Performance Engineering*, ICPE '11, pages 449–454, New York, NY, USA, 2011. ACM.
- [41] Christina Catley, Dorina C Petriu, and Monique Frize. Software performance engineering of a web service-based clinical decision support infrastructure. In *Proceedings of the 4th International Workshop on Software and Performance*, WOSP '04, pages 130–138, New York, NY, USA, 2004. ACM.
- [42] Abhishek Chandra, Micah Adler, Pawan Goyal, and Prashant Shenoy. Surplus fair scheduling: A proportional-share CPU scheduling algorithm for symmetric multiprocessors. In *Proceedings of the 4th Symposium on Operating System Design and Implementation (OSDI'00)*, pages 45–58, San Diego, California, October 23–25 2000.
- [43] K. Mani Chandy and Doug Neuse. Linearizer: A heuristic algorithm for queueing network models of computing systems. *Communications of the ACM*, 25(2):126–134, February 1982.

- [44] K Mani Chandy and Charles H Sauer. Computational algorithms for product form queueing networks. *Communications of the ACM*, 23(10):573–583, 1980.
- [45] Hong Chen and David D Yao. *Fundamentals of queueing networks: Performance, asymptotics, and optimization*, volume 46. Springer Science & Business Media, 2013.
- [46] Ludmila Cherkasova, Diwaker Gupta, and Amin Vahdat. Comparison of the three CPU schedulers in Xen. *SIGMETRICS Perform. Eval. Rev.*, 35(2):42–51, September 2007.
- [47] Giovanni Chiola. A graphical Petri Net tool for performance analysis. In *Modelling Techniques and Performance Evaluation*, volume 7, pages 337–337. Elsevier Science, Amsterdam, March 1987.
- [48] Giovanni Chiola, Giuliana Franceschinis, Rossano Gaeta, and Marina Ribaudou. GreatSPN 1.7: GGraphical Editor and analyzer for timed and stochastic Petri nets. *Performance Evaluation*, 24(1):47–68, 1995.
- [49] Cosmina Chișe and Ioan Jurca. Hybrid analytical-simulation model used to evaluate and improve system performance. In *Parallel and Distributed Computing (ISPDC), 2011 10th International Symposium on*, pages 240–246, Cluj Napoca, July 6–8 2011.
- [50] Gianfranco Ciardo and Andrew S Miner. SMART: Simulation and Markovian analyzer for reliability and timing. In *Proc. IEEE Intl. Computer Performance and Dependability Symposium (IPDS’96)*, Urbana-Champaign, IL, USA, September 4–6 1996.
- [51] Gianfranco Ciardo, Jogesh Muppala, and T Trivedi. SPNP: stochastic petri net package. In *Petri Nets and Performance Models, 1989. PNPM89., Proceedings of the Third International Workshop on*, pages 142–151. IEEE, 1989.

- [52] Allan Clark, Stephen Gilmore, Jane Hillston, and Mirco Tribastone. Stochastic process algebras. In Marco Bernardo and Jane Hillston, editors, *Formal Methods for Performance Evaluation*, volume 4486 of *Lecture Notes in Computer Science*, pages 132–179. Springer Berlin Heidelberg, 2007.
- [53] Compuware. Application performance management survey, October 2006.
- [54] Adrian E. Conway, E de Souza e Silva, and Stephen S Lavenberg. Mean value analysis by chain of product form queueing networks. *Computers, IEEE Transactions on*, 38(3):432–442, 1989.
- [55] Adrian E Conway and Nicolas D Georganas. RECAL— a new efficient algorithm for the exact analysis of multiple-chain closed queueing networks. *J. ACM*, 33(4):768–791, August 1986.
- [56] Vittorio Cortellessa, Antiniscia Di Marco, and Paola Inverardi. *Model-based software performance analysis*. Springer Science & Business Media, 2011.
- [57] Vittorio Cortellessa and Raffaella Mirandola. Deriving a queueing network based performance model from UML diagrams. In *Proceedings of the 2nd international workshop on Software and performance*, pages 58–70. ACM, 2000.
- [58] David Daly, Daniel D Deavours, Jay M Doyle, Patrick G Webster, and William H Sanders. Möbius: An extensible tool for performance and dependability modeling. In *Computer Performance Evaluation. Modelling Techniques and Tools*, pages 332–336. Springer, 2000.
- [59] Andrea D’Ambrogio. A model transformation framework for the automated building of performance models from uml models. In *Proceedings of the 5th international workshop on Software and performance*, pages 75–86. ACM, 2005.

- [60] Garth A. Gibson David Petrou, John W. Milford. Implementing lottery scheduling: Matching the specializations in traditional schedulers”. In *Proceedings of the 1999 USENIX Annual Technical Conference*, pages 1–14, Monterey, California, June 6–11 1999.
- [61] Daniel D. Deavours, Graham Clark, Tod Courtney, David Daly, Salem Derisavi, Jay M. Doyle, William H. Sanders, and Patrick G. Webster. The Möbius framework and its implementation. *IEEE Transactions on Software Engineering*, 28(10):956–969, October 2002.
- [62] Alan Demers, Srinivasan Keshav, and Scott Shenker. Analysis and simulation of a fair queueing algorithm. In *Symposium Proceedings on Communications Architectures & Protocols*, SIGCOMM ’89, pages 1–12, New York, NY, USA, 1989. ACM.
- [63] John Dille, Rich Friedrich, Tai Jin, and Jerome Rolia. Web server performance measurement and modeling techniques. *Performance Evaluation*, 33(1):5–26, 1998. Tools for Performance Evaluation.
- [64] Nicholas J Dingle, William J Knottenbelt, and Tamas Suto. PIPE2: a tool for the performance evaluation of generalised stochastic petri nets. *ACM SIGMETRICS Performance Evaluation Review*, 36(4):34–39, 2009.
- [65] Salvatore Distefano, Marco Scarpa, and Antonio Puliafito. From UML to Petri nets: The PCM-based methodology. *Software Engineering, IEEE Transactions on*, 37(1):65–79, 2011.
- [66] Kenneth J. Duda and David R. Cheriton. Borrowed-virtual-time (BVT) scheduling: Supporting latency-sensitive threads in a general-purpose scheduler. *SIGOPS Oper. Syst. Rev.*, 33(5):261–276, December 1999.

- [67] E de Souza e Silva and Richard R Muntz. A note on the computational cost of the linearizer algorithm for queueing networks. *IEEE Transactions on Computers*, 39(6):840–842, 1990.
- [68] Renato J Figueiredo, Peter Dinda, Jose Fortes, et al. A case for grid computing on virtual machines. In *Distributed Computing Systems, 2003. Proceedings. 23rd International Conference on*, pages 550–559, May 2003.
- [69] Paul Fortier and Howard Michel. *Computer systems performance evaluation and prediction*. Digital Press, 2003.
- [70] Jean-Michel Fourneau, Erol Gelenbe, and Rina Suros. G-networks with multiple classes of negative and positive customers. *Theoretical Computer Science*, 155(1):141–156, 1996.
- [71] Robert France and Bernhard Rumpe. Model-driven development of complex software: A research roadmap. In *2007 Future of Software Engineering, FOSE '07*, pages 37–54, Washington, DC, USA, 2007. IEEE Computer Society.
- [72] Greg Franks. Traffic dependencies in client-server systems and their effect on performance prediction. In *IEEE International Computer Performance and Dependability Symposium*, pages 24–33, Erlangen, Germany, April 1995.
- [73] Greg Franks. Simulating layered queueing networks with passive resources. In *Proceedings of the 2011 Spring Simulation Multiconference*, volume 4 (TMS/DEVS), pages 8–15, Boston, MA, June 4–7 2011.
- [74] Greg Franks, Tariq Al-Omari, Murray Woodside, Olivia Das, and Salem Derisavi. Enhanced modeling and solution of layered queueing networks. *IEEE Transactions on Software Engineering*, 35(2):148–161, March–April 2009.

- [75] Greg Franks, Danny Lau, and Curtis Hrischuk. Performance measurements and modeling of a java-based session initiation protocol (sip) application server. In *Proceedings of the Joint ACM SIGSOFT Conference – QoSA and ACM SIGSOFT Symposium – ISARCS on Quality of Software Architectures – QoSA and Architecting Critical Systems – ISARCS*, QoSA-ISARCS '11, pages 63–72, New York, NY, USA, 2011. ACM.
- [76] Greg Franks and Lianhua Li. Efficiency improvements for solving layered queueing networks. In *Proc. 3rd WOSP/SIPEW International Conference on Performance Engineering (ICPE '12)*, pages 279–282, Boston, MA, USA, April 22–25 2012.
- [77] Greg Franks, Peter Maly, Murray Woodside, Dorina C. Petriu, and Alex Hubbard. *Layered Queueing Network Solver and Simulator User Manual*. Real-time and Distributed Systems Lab, Carleton University, Ottawa, 2014.
- [78] Greg Franks and Murray Woodside. Performance of multi-level client-server systems with parallel service operations. In *Proceedings of the First International Workshop on Software and Performance (WOSP '98)*, pages 120–130, Santa Fe, NM, October 12–16 1998. Association for Computing Machinery.
- [79] Greg Franks and Murray Woodside. Effectiveness of early replies in client-server systems. *Performance Evaluation*, 36(1):165–184, August 1999.
- [80] Greg Franks and Murray Woodside. Multiclass multiservers with deferred operations in layered queueing networks, with software system applications. In Doug DeGroot and Pete Harrison, editors, *Proceedings of the Twelfth IEEE/ACM International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunications Systems (MASCOTS 2004)*, pages 239–248, Volendam, The Netherlands, October 2004.

- [81] Roy Gregory Franks. *Performance Analysis of Distributed Server Systems*. PhD thesis, Department of Systems and Computer Engineering, Carleton University, Ottawa, Ontario, Canada, December 1999.
- [82] Mathias Fritzsche and Jendrik Johannes. Putting performance engineering into model-driven engineering: Model-driven performance engineering. In Holger Giese, editor, *Models in Software Engineering*, volume 5002 of *Lecture Notes in Computer Science*, pages 164–175. Springer Berlin Heidelberg, 2008.
- [83] Erol Gelenbe. Product-form queueing networks with negative and positive customers. *Journal of Applied Probability*, pages 656–663, 1991.
- [84] Erol Gelenbe and Jean-Michel Fourneau. G-networks with resets. *Performance Evaluation*, 49(1):179–191, 2002.
- [85] Erol Gelenbe and Isi Mitrani. *Analysis and Synthesis of Computer Systems: Texts*. Imperial College Press, London, UK, UK, 2nd edition, 2010.
- [86] Abdelouahed Gherbi and Ferhat Khendek. From UML/SPT models to schedulability analysis: a metamodel-based transformation. In *Object and Component-Oriented Real-Time Distributed Computing, 2006. ISORC 2006. Ninth IEEE International Symposium on*, pages 8–pp. IEEE, 2006.
- [87] Ali Ghodsi, Vyas Sekar, Matei Zaharia, and Ion Stoica. Multi-resource fair queueing for packet processing. *ACM SIGCOMM Computer Communication Review*, 42(4):1–12, 2012.
- [88] Christopher D Gill, David L Levine, and Douglas C Schmidt. The design and performance of a real-time corba scheduling service. In *Challenges in Design and Implementation of Middlewares for Real-Time Systems*, pages 3–40. Springer, 2001.

- [89] Stephen Gilmore and Jane Hillston. The PEPA workbench: A tool to support a process algebra-based approach to performance modelling. In *Computer Performance Evaluation Modelling Techniques and Tools*, pages 353–368. Springer, 1994.
- [90] Claude Girault and Rüdiger Valk. *Petri nets for systems engineering: a guide to modeling, verification, and applications*. Springer Science & Business Media, 2013.
- [91] John B. Goodenough. Exception handling: Issues and a proposed notation. *Communications of the ACM*, 18(12):683–696, December 1975.
- [92] William J. Gordon and Gordon F. Newell. Closed queuing systems with exponential servers. *Operations Research*, 15(2):254–265, 1967.
- [93] Gordon P Gu and Dorina C Petriu. Xslt transformation from uml models to lqn performance models. In *Proceedings of the 3rd international workshop on Software and performance*, pages 227–234. ACM, 2002.
- [94] Philip Heidelberger and Stephen S Lavenberg. Computer performance evaluation methodology. *Computers, IEEE Transactions on*, 33(12):1195–1220, December 1984.
- [95] William Henderson, Charles E. M. Pearce, Peter G. Taylor, and Nico M Van Dijk. Closed queueing networks with batch services. *Queueing systems*, 6(1):59–70, 1990.
- [96] William Henderson and Peter G. Taylor. Product form in networks of queues with batch arrivals and batch services. *Queueing systems*, 6(1):71–87, 1990.
- [97] Gary J Henry. The UNIX system: the fair share scheduler. *AT&T Bell Laboratories Technical Journal*, 63(8):1845–1857, 1984.

- [98] Holger Hermanns, Ulrich Herzog, Ulrich Klehmet, Vassilis Mertsiotakis, and Markus Siegle. Compositional performance modelling with the TIPPTool. *Performance Evaluation*, 39(1–4):5 – 35, 2000.
- [99] Jane Hillston. A tool to enhance model exploitation. *Performance Evaluation*, 22(3):59–74, February 1995.
- [100] Jane Hillston. *A Compositional Approach to Performance Modelling*. Cambridge University Press, New York, NY, USA, 1996.
- [101] Jane Hillston. *A compositional approach to performance modelling*, volume 12. Cambridge University Press, 2005.
- [102] Jane Hillston and Marina Ribaudo. Stochastic process algebras: a new approach to performance modeling. *Modeling and Simulation of Advanced Computer Systems*, pages 235–256, 1998.
- [103] Christophe Hirel, Bruno Tuffin, and Kishor S Trivedi. SPNP: Stochastic petri nets. version 6.0. In *Computer Performance Evaluation. Modelling Techniques and Tools*, pages 354–357. Springer, 2000.
- [104] John Hutchinson, Mark Rouncefield, and Jon Whittle. Model-driven engineering practices in industry. In *Software Engineering (ICSE), 2011 33rd International Conference on*, pages 633–642. IEEE, 2011.
- [105] Tauseef Israr, Murray Woodside, and Greg Franks. Interaction tree algorithms to extract effective architecture and layered performance models from traces. *J. Syst. and Soft.*, 80(4):474–492, April 2007.
- [106] Teerawat Issariyakul and Ekram Hossain. *Introduction to Network Simulator NS2*. Springer, 2011.

- [107] James R Jackson. Jobshop-like queueing systems. *Management science*, 10(1):131–142, 1963.
- [108] Patricia A. Jacobson and Edward D. Lazowska. Analyzing queueing networks with simultaneous resource possession. *Communications of the ACM*, 25(2):142–151, February 1982.
- [109] Raj Jain. *The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling*. John Wiley & Sons, 1991.
- [110] Kurt Jensen. Coloured petri nets. In W. Brauer, W. Reisig, and G. Rozenberg, editors, *Petri Nets: Central Models and Their Properties*, volume 254 of *Lecture Notes in Computer Science*, pages 248–299. Springer Berlin Heidelberg, 1987.
- [111] Kurt Jensen. *Coloured Petri nets: basic concepts, analysis methods and practical use*, volume 1. Springer Science & Business Media, 2013.
- [112] Kurt Jensen and Lars M Kristensen. *Coloured Petri nets: modelling and validation of concurrent systems*. Springer Science & Business Media, 2009.
- [113] P. Jogalekar and M. Woodside. Evaluating the scalability of distributed systems. *Parallel and Distributed Systems, IEEE Transactions on*, 11(6):589–603, Jun 2000.
- [114] Pekka Kähkipuro. UML based performance modeling framework for object oriented systems. In *UML '99, The Unified Modeling Language, Beyond the Standard*, number 1723 in *Lecture Notes in Computer Science*, pages 356–371, Berlin, 1999. Springer-Verlag.
- [115] Lucia Kapova and Ralf Reussner. Application of advanced model-driven tech-

- niques in performance engineering. In *Computer Performance Engineering*, pages 17–36. Springer, 2010.
- [116] Judy Kay and Piers Lauder. A fair share scheduler. *Communications of the ACM*, 31(1):44–55, January 1988.
- [117] Stuart Kent. Model driven engineering. In *Integrated formal methods*, pages 286–298. Springer, 2002.
- [118] Stephen R Kimbleton. A heuristic approach to computer systems performance improvement, I: a fast performance prediction tool. In *Proc. AFIPS 1975 National Computer Conference*, volume 44, pages 839–846, Montvale, NJ, May 1975. AFIPS Press.
- [119] Anneke G Kleppe, Jos Warmer, and Wim Bast. *MDA Explained, The model driven architecture: practice and promise*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, 2003.
- [120] Samuel Kounev and Alejandro Buchmann. SimQPN—A tool and methodology for analyzing queueing Petri net models by means of simulation. *Performance Evaluation*, 63(4):364–394, 2006.
- [121] Samuel Kounev and Christofer Dutz. QPME: a performance modeling tool based on queueing petri nets. *ACM SIGMETRICS Performance Evaluation Review*, 36(4):46–51, 2009.
- [122] Samuel Kounev, Simon Spinner, and Philipp Meier. Introduction to queueing petri nets: modeling formalism, tool support and case studies. In *Proceedings of the 3rd ACM/SPEC International Conference on Performance Engineering*, pages 9–18. ACM, 2012.

- [123] Heiko Koziolk. Performance evaluation of component-based software systems: A survey. *Performance Evaluation*, 67(8):634 – 658, 2010. Special Issue on Software and Performance.
- [124] Heiko Koziolk and Ralf Reussner. A model transformation from the Palladio component model to Layered Queueing Networks. In *Performance Evaluation: Metrics, Models and Benchmarks*, pages 58–78. Springer, 2008.
- [125] Peter Kubat. Estimation of reliability for communication/computer networks-simulation/analytic approach. *IEEE Transactions on Communications*, 37(9):927–933, September 1989.
- [126] Simon Künzli, Francesco Poletti, Luca Benini, and Lothar Thiele. Combining simulation and formal methods for system-level performance analysis. In *Proc. 2006 Design, Automation and Test in Europe (DATE '06)*, pages 1–6, Munich, Germany, March 6–10 2006. IEEE.
- [127] Simon S. Lam and Y. Luke Lien. A tree convolution algorithm for the solution of queueing networks. *Comm. ACM*, 26(3):203–215, March 1983.
- [128] Daniel J. Lasser. Productivity of multiprogrammed computers progress in developing an analytic prediction method. *Comm. ACM*, 12(12):678–684, December 1969.
- [129] Edward D. Lazowska, John Zhorjan, Scott G. Graham, and Kenneth C. Sevcik. *Quantitative System Performance; Computer System Analysis Using Queueing Network Models*. Prentice Hall, Englewood Cliffs, NJ, 1984.
- [130] Young Hae Lee and Sook Han Kim. Optimal production-distribution planning in supply chain management using a hybrid simulation-analytic approach. In *Proceedings of the 32nd conference on Winter simulation (WSC '00)*, pages

- 1252–1259, Orlando, FL USA, December 10–13 2000. Society for Computer Simulation International.
- [131] Jim (Zhanwen) Li, John Chinneck, Murray Woodside, and Marin Litoiu. Fast scalable optimization to configure service systems having cost and quality of service constraints. In *Proceedings of the 6th International Conference on Autonomic Computing, ICAC '09*, pages 159–168, New York, NY, USA, 2009. ACM.
- [132] Lianhua Li and Greg Franks. Performance modeling of systems using fair share scheduling with layered queueing networks. In *Proceedings of the Seventeenth IEEE/ACM International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunications Systems (MASCOTS 2009)*, pages 1–10, London, September 21–23 2009.
- [133] Lianhua Li and Greg Franks. Hybrid performance modeling using layered queueing networks. In *cmg-2012*, volume 1, pages 85–93, Las Vegas, Nevada, USA, December 3–7 2012. Computer Measurement Group, Curran Associates, Inc.
- [134] Lianhua Li and Greg Franks. Improved chain calculation for sub-chain dependencies in layered queueing networks. In *UIC-ATC-ScalCom-CBDCom-IoP 2015*, pages 726–733, Beijing, China, May 2015.
- [135] Lianhua Li and Greg Franks. Improved interlock correction when solving layered queueing networks using decomposition. In *Electrical and Computer Engineering (CCECE), 2015 IEEE 28th Canadian Conference on*, pages 541–546, May 2015.
- [136] Lianhua Li and Greg Franks. Analytic solution of fair share scheduling in layered queueing networks (accepted). In *13th European Workshop on Performance Engineering (EPEW 2016)*, Chios, Greece, October 2016. Springer.

- [137] Lianhua Li and Greg Franks. Modeling decisions in layered queueing networks. In *48th Summer Computer Simulation Conference 2016 (SCSC 2016)*, pages 175–182, Montreal, Quebec, Canada, July 2016. ACM.
- [138] Christoph Lindemann. DSPNexpress: A software package for the efficient solution of deterministic and stochastic petri nets. *Performance Evaluation*, 22:15–29, 1995.
- [139] John D. C. Little. A proof for the queuing formula: $L = \lambda W$. *Operations Research*, 9(3):383–387, 1961.
- [140] Te-Kai Liu, A. Behroozi, and S. Kumaran. A performance model for a business process integration middleware. In *E-Commerce, 2003. CEC 2003. IEEE International Conference on*, pages 191–198, June 2003.
- [141] Peter Maly and C. Murray Woodside. Layered modeling of hardware and software, with application to a lan extension router. In BoudewijnR. Haverkort, HenrikC. Bohnenkamp, and ConnieU. Smith, editors, *Computer Performance Evaluation. Modelling Techniques and Tools*, volume 1786 of *Lecture Notes in Computer Science*, pages 10–24. Springer Berlin Heidelberg, 2000.
- [142] Marco Ajmone Marsan, Gianfranco Balbo, Gianni Conte, Susanna Donatelli, and Giuliana Franceschinis. *Modelling with Generalized Stochastic Petri Nets*. John Wiley & Sons, Inc., New York, NY, USA, 1st edition, 1994.
- [143] Moreno Marzolla. The qnetworks toolbox: A software package for queueing networks analysis. In Khalid Al-Begain, Dieter Fiems, and WilliamJ. Knottenbelt, editors, *Analytical and Stochastic Modeling Techniques and Applications*, volume 6148 of *Lecture Notes in Computer Science*, pages 102–116. Springer Berlin Heidelberg, 2010.

- [144] Philipp Meier, Samuel Kounev, and Heiko Koziolk. Automated transformation of component-based software architecture models to Queueing Petri Nets. In *Modeling, Analysis & Simulation of Computer and Telecommunication Systems (MASCOTS), 2011 IEEE 19th International Symposium on*, pages 339–348. IEEE, 2011.
- [145] Daniel A. Menascé. Two-level iterative queuing modeling of software contention. In *Proc. 10th IEEE/ACM International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS 2002)*, pages 267–276, Fort Worth, TX, October 12–16 2002.
- [146] Andrew Miga. *Application of use case maps to system design with tool support*. PhD thesis, Carleton University, 1998.
- [147] Joaquin Miller and Jishnu Mukerji. Mda guide version 1.0, object management group. URL: <http://www.omg.org/mda/mdafiles/MDAGuideVersion1-0.pdf>, 2003. (*Accessed June 18, 2016*).
- [148] Ahmad Mizan and Greg Franks. Automated performance model construction through event log analysis. In *Software Testing, Verification and Validation (ICST), 2012 IEEE Fifth International Conference on*, pages 636–641, April 2012.
- [149] Michael K Molloy. Performance analysis using stochastic petri nets. *Computers, IEEE Transactions on*, C-31(9):913–917, Sept 1982.
- [150] Martin Mroz and Greg Franks. A performance experiment system supporting fast mapping of system issues. In *Fourth International Conference on Performance Evaluation Methodologies and Tools*, pages 246–255, Pisa, Italy, October 20–22 2009. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering).

- [151] Tadao Murata. Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, 77(4):541–580, Apr 1989.
- [152] John Nagle. On packet switches with infinite storage. *IEEE transactions on communications*, 35(4):435–438, April 1987.
- [153] John E. Neilson. *PARASOL Users Manual*. School of Computer Science, Carleton University, Ottawa, Canada, (version 3.1) edition.
- [154] John E. Neilson. PARASOL: A simulator for distributed and/or parallel systems. Technical Report SCS TR-192, School of Computer Science, Carleton University, Ottawa, Ontario, Canada, May 1991.
- [155] E. Neron and C. M. Woodside. A performance model for rendezvous based systems with processor-shared service. Technical report, Department of Systems and Computer Engineering, Carleton University, Ottawa, Ontario, Canada, September 1986.
- [156] Kyle J Nesbit, Nidhi Aggarwal, James Laudon, and James E Smith. Fair queueing memory systems. In *Microarchitecture, 2006. MICRO-39. 39th Annual IEEE/ACM International Symposium on*, pages 208–222, Dec 2006.
- [157] Object Management Group. UML profile for schedulability, performance, and time specification. Technical Report formal/2005-07-04., Object Management Group, July 2005.
- [158] Object Management Group. UML profile for MARTE: Modeling and analysis of real-time and embedded systems. Technical Report formal/2011-06-02, Object Management Group, June 2011.
- [159] Tariq Omari, Salem Derisavi, and Greg Franks. Deriving distribution of thread service time in layered queueing networks. In *Proceedings of the Sixth Inter-*

- national Workshop on Software and Performance (WOSP '07)*, pages 66–77, Buenos Aires, Argentina, February 5–7 2007.
- [160] Tariq Omari, Salem Derisavi, Greg Franks, and Murray Woodside. Performance modeling of a quorum pattern in layered service systems. In *Proceedings of the Fourth International Conference on the Quantative Evaluation of Systems (QEST)*, pages 201–210, Edinburgh, Scotland, September 16–19 2007.
- [161] Tariq Omari, Greg Franks, Murray Woodside, and Amy Pan. Solving layered queueing networks of large client-server systems with symmetric replication. In *Proceedings of the Fifth International Workshop on Software and Performance (WOSP '05)*, pages 159–166, Palma de Mallorca, Spain, July 11–14 2005.
- [162] Tariq Omari, Greg Franks, Murray Woodside, and Amy Pan. Efficient performance models for layered server systems with replicated servers and parallel behaviour. *Journal of Systems and Software*, 80(4):510–527, April 2007.
- [163] Randy Otte, Paul Patrick, and Mark Roy. *Understanding CORBA (Common Object Request Broker Architecture)*. Prentice-Hall, Inc., 1995.
- [164] Chandandeep Singh Pabla. Completely fair scheduler. *Linux Journal*, 2009(184):4, 2009.
- [165] James L. Peterson. Petri nets. *ACM Comput. Surv.*, 9(3):223–252, September 1977.
- [166] Carl Adam Petri. Kommunikation mit automaten. 1962.
- [167] Dorin Petriu, Daniel Amyot, and Murray Woodside. Scenario-based performance engineering with UCMNAV. In *SDL 2003: System Design*, pages 18–35. Springer, 2003.

- [168] Dorin Petriu and Murray Woodside. Analysing software requirements specifications for performance. In *Proceedings of the Third International Workshop on Software and Performance (WOSP '02)*, pages 1–9, Rome, Italy, July 24–26 2002.
- [169] Dorin Petriu and Murray Woodside. An intermediate metamodel with scenarios and resources for generating performance models from UML designs. *Software & Systems Modeling*, 6(2):163–184, 2007.
- [170] Dorina Petriu, Hoda Amer, Shikharesh Majumdar, and Istabrak Abdull-Fatah. Using analytic models predicting middleware performance. In *Proceedings of the 2nd International Workshop on Software and Performance, WOSP '00*, pages 189–194, New York, NY, USA, 2000. ACM.
- [171] Dorina C. Petriu. Approximate mean value analysis of client–server systems with multi-class requests. In *Proceedings of the 1994 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems.*, pages 77–86, Nashville, TN, USA, May 1994.
- [172] Dorina C Petriu. Software model-based performance analysis. *Model Driven Engineering for distributed Real-Time Systems: MARTE modelling, model transformations and their usages (JP Babau, M. Blay-Fornarino, J. Champeau, S. Robert, A. Sabetta, Eds.)*, ISTE Ltd and John Wiley & Sons Inc, 2010.
- [173] Dorina C Petriu and Hui Shen. Applying the UML performance profile: Graph grammar-based derivation of LQN models from UML specifications. In *Computer Performance Evaluation: Modelling Techniques and Tools*, pages 159–177. Springer, 2002.
- [174] Dorina C. Petriu and C. Murray Woodside. Approximate MVA for software client/server models by markov chain task-directed aggregation. In *The Third*

- IEEE Symposium on Parallel and Distributed Processing*, pages 322–329, Dallas, TX USA, December 2–5 1991.
- [175] Yan Qiang, Yi Li, and Junjie Chen. The workload adaptation in autonomic dbmss based on layered queuing network model. In *Knowledge Discovery and Data Mining, 2009. WKDD 2009. Second International Workshop on*, pages 781–785, Jan 2009.
- [176] Parameswaran Ramanathan and Prathima Agrawal. Adapting packet fair queueing algorithms to wireless networks. In *Proceedings of the 4th Annual ACM/IEEE International Conference on Mobile Computing and Networking, MobiCom '98*, pages 1–9, New York, NY, USA, 1998. ACM.
- [177] Sridhar Ramesh and Harry G Perros. A multi-layer client-server queueing network model with synchronous and asynchronous messages. In *Proc. 1st International Workshop on Software and Performance (WOSP '98)*, pages 107–119, Santa Fe, NM USA, October 12–16 1998.
- [178] Martin Reiser. A queueing network analysis of computer communication networks with window flow control. *IEEE Transactions on Communications*, 27(8):1199 – 1209, August 1979.
- [179] Martin Reiser and Hisashi Kobayashi. Horner’s rule for the evaluation of general closed queueing networks. *Comm. ACM*, 18(10):592–593, October 1975.
- [180] Martin Reiser and Hisashi Kobayashi. Queuing networks with multiple closed chains: theory and computational algorithms. *IBM Journal of Research and Development*, 19(3):283–294, 1975.
- [181] Martin Reiser and Stephen S Lavenberg. Mean value analysis of closed multi-chain queueing networks. *Journal of the ACM*, 27(2):313–322, April 1980.

- [182] Thomas G Robertazzi. *Computer networks and systems: queueing theory and performance evaluation*. Springer Science & Business Media, 2012.
- [183] Jerome A. Rolia and Kenneth C. Sevcik. The method of layers. *IEEE Transactions on Software Engineering*, 21(8):689–700, August 1995.
- [184] Jerome Alexander Rolia. *Predicting the Performance of Software Systems*. PhD thesis, Univerisity of Toronto, Toronto, Ontario, Canada, January 1992.
- [185] Robin A Sahner, Kishor Trivedi, and Antonio Puliafito. *Performance and reliability analysis of computer systems: an example-based approach using the SHARPE software package*. Springer Science & Business Media, 2012.
- [186] Paul J Schweitzer. Approximate analysis of multiclass closed networks of queues. In *Proceedings of the International Conference on Stochastic Control and Optimization*, pages 25–29, Amsterdam, 1979.
- [187] Paul J. Schweitzer, Giuseppe Serazzi, and Marco Broglia. A queue-shift approximation technique for product-form queueing networks. In Ramon Puigjaner, NunzioN. Savino, and Bartomeu Serra, editors, *Computer Performance Evaluation*, volume 1469 of *Lecture Notes in Computer Science*, pages 267–279. Springer Berlin Heidelberg, 1998.
- [188] H. D. Schwetman. Hybrid simulation models of computer systems. *Comm. ACM*, 21(9):718–723, September 1978.
- [189] Kenneth C Sevcik and Isi Mitrani. The distribution of queueing network states at input and output instants. *Journal of the ACM*, 28(2):358–371, April 1981.
- [190] J. George Shanthikumar and Robert G. Sargent. A unifying view of hybrid simulation/analytic models and modeling. *Operations Research*, 31(6):1030–1052, November–December 1983.

- [191] Fahim Sheikh and Murray Woodside. Layered analytic performance modelling of a distributed database system. In *Distributed Computing Systems, 1997., Proceedings of the 17th International Conference on*, pages 482–490, May 1997.
- [192] Yasir Shoaib and Olivia Das. Web application performance modeling using layered queueing networks. *Electronic Notes in Theoretical Computer Science*, 275:123 – 142, 2011. Fifth International Workshop on the Practical Application of Stochastic Modelling (PASM).
- [193] Dmitriy Shorin, Armin Zimmermann, and Paulo Maciel. Transforming UML state machines into stochastic petri nets for energy consumption estimation of embedded systems. In *Sustainable Internet and ICT for Sustainability (SustainIT), 2012*, pages 1–6. IEEE, 2012.
- [194] Madhavapeddi Shreedhar and George Varghese. Efficient fair queueing using deficit round robin. *SIGCOMM Comput. Commun. Rev.*, 25(4):231–242, October 1995.
- [195] Connie U Smith. Introduction to software performance engineering: Origins and outstanding problems. In M. Bernardo and J. Hillston, editors, *Formal Methods for Performance Evaluation (SFM 2007)*, volume 4486 of *Lecture Notes in Computer Science*, pages 395–428. Springer-Verlag, Berlin, 2007.
- [196] Connie U. Smith and Lloyd G. Williams. Software performance engineering: A case study including performance comparison with design alternatives. *Software Engineering, IEEE Transactions on*, 19(12):720–741, July 1993.
- [197] Connie U Smith and Lloyd G Williams. *Performance Solutions: A Practical Guide to Creating Responsive, Scalable Software*. Object Technology Series. Addison Wesley, 2002.

- [198] Simon Spinner, Samuel Kounev, and Philipp Meier. Stochastic modeling and analysis using QPME: Queueing petri net modeling environment v2. 0. In *Application and Theory of Petri Nets*, pages 388–397. Springer, 2012.
- [199] Thomas Stahl, Markus Voelter, and Krzysztof Czarnecki. *Model-driven software development: technology, engineering, management*. John Wiley & Sons, 2006.
- [200] Tony Spiteri Staines. Intuitive mapping of UML 2 activity diagrams into fundamental modeling concept petri net diagrams and colored petri nets. In *Engineering of Computer Based Systems, 2008. ECBS 2008. 15th Annual IEEE International Conference and Workshop on the*, pages 191–200. IEEE, 2008.
- [201] Dimitrios Stiliadis and Anujan Varma. Efficient fair queueing algorithms for packet-switched networks. *Networking, IEEE/ACM Transactions on*, 6(2):175–185, Apr 1998.
- [202] Rajan Suri, Sushanta Sahu, and Mary Vernon. Approximate mean value analysis for closed queueing networks with multiple-server stations. In *Proceedings of the 2007 Industrial Engineering Research Conference*, 2007.
- [203] Mirco Tribastone. Relating layered queueing networks and process algebra models. In *Proceedings of the first joint WOSP/SIPEW International Conference on Performance Engineering*, pages 183–194. ACM, 2010.
- [204] Mirco Tribastone. A fluid model for layered queueing networks. *IEEE Transactions on Software Engineering*, 39(6):744–756, June 2013.
- [205] Mirco Tribastone and Stephen Gilmore. Automatic extraction of PEPA performance models from UML activity diagrams annotated with the MARTE profile. In *Proceedings of the 7th international workshop on Software and performance*, pages 67–78. ACM, 2008.

- [206] Mirco Tribastone, Philip Mayer, and Martin Wirsing. Performance prediction of service-oriented systems with layered queueing networks. In Tiziana Margaria and Bernhard Steffen, editors, *Leveraging Applications of Formal Methods, Verification, and Validation*, volume 6416 of *Lecture Notes in Computer Science*, pages 51–65. Springer Berlin Heidelberg, 2010.
- [207] Kishor S. Trivedi, Gianfranco Ciardo, and Jogesh K. Muppala. SPNP version 6 user manual. *Dept. of Electrical Eng., Duke Univ., Durham, NC*, 1999.
- [208] Michel Véran, Bull Sems, France Eychirolles, and Dominique Potier. QNAP 2: A portable environment for queueing systems modelling. *Technical Report 314*, 1984.
- [209] Tom Verdickt, Bart Dhoedt, Frank Gielen, and Piet Demeester. Modelling the performance of CORBA using layered queueing networks. In *Euromicro Conference, 2003. Proceedings. 29th*, pages 117–123. IEEE, Sept 2003.
- [210] Tom Verdickt, Bart Dhoedt, Filip De Turck, and Piet Demeester. Hybrid performance modeling approach for network intensive distributed software. In *Proc. 6th Intl. Workshop on Software and Performance (WOSP '07)*, pages 189 – 200, Buenos Aires, Argentina, February 5–8 2007.
- [211] Carl A. Waldspurger. *Lottery and Stride Scheduling: Flexible Proportional-Share Resource Management*. PhD thesis, Massachusetts Institute of Technology, September 1995.
- [212] Carl A. Waldspurger and William E. Weihl. Lottery scheduling: Flexible proportional-share resource management. In *Proceedings of the 1st USENIX Conference on Operating Systems Design and Implementation, OSDI '94*, page 1, Berkeley, CA, USA, November 14–17 1994. USENIX Association.

- [213] Carl A. Waldspurger and William E. Weihl. Stride scheduling: Deterministic proportional-share resource management. Technical Report TM-528, Massachusetts Institute of Technology, Cambridge, MA, USA, June 1995.
- [214] Hai Wang and Kenneth C. Sevcik. Experiments with improved approximate mean value analysis algorithms. *Performance Evaluation*, 39(1–4):189 – 206, 2000.
- [215] Hai Wang, Kenneth C Sevcik, Giuseppe Serazzi, and Shouhong Wang. The general form linearizer algorithms: A new family of approximate mean value analysis algorithms. *Performance Evaluation*, 65(2):129–151, 2008.
- [216] Chuliang Weng, Zhigang Wang, Minglu Li, and Xinda Lu. The hybrid scheduling framework for virtual machine systems. In *Proceedings of the 2009 ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '09, pages 111–120, New York, NY, USA, 2009. ACM.
- [217] Timothy Wood, Prashant J Shenoy, Arun Venkataramani, and Mazin S Yousif. Black-box and gray-box strategies for virtual machine migration. In *NSDI*, volume 7, pages 17–17, 2007.
- [218] C. Murray Woodside. Controllability of computer performance tradeoffs obtained using controlled-share queue schedulers. *IEEE Transactions on Software Engineering*, 12(10):1041–1048, October 1986.
- [219] C. Murray Woodside. Throughput calculation for basic stochastic rendezvous networks. *Performance Evaluation*, 9:143–160, 1989.
- [220] C. Murray Woodside, John E. Neilson, Dorina C. Petriu, and Shikharesh Majumdar. The stochastic rendezvous network model for performance of synchronous client-server-like distributed software. *IEEE Transactions on Computers*, 44(8):20–34, August 1995.

- [221] C. Murray. Woodside, E. Neron, E. D.-S. Ho, and B. Mondoux. An “Active Server” model for the performance of parallel programs written using rendezvous. *Journal of Systems and Software*, pages 844–848, 1986.
- [222] Murray Woodside. From annotated software designs (UML SPT/MARTE) to model formalisms. In *Formal Methods for Performance Evaluation*, pages 429–467. Springer, 2007.
- [223] Murray Woodside, Greg Franks, and Dorina C. Petriu. The future of software performance engineering. In Lionel C. Briand and Alexander L. Wolf, editors, *Future of Software Engineering FOSE 07*, pages 171–187, Minneapolis, MN, USA, May 23–25 2007. IEEE Computer Society Press.
- [224] Pengfei Wu. *Extending Layered Queueing Network with Hybrid Sub models Representing Exceptions and Decision Making*. PhD thesis, Department of Systems and Computer Engineering, Carleton University, Ottawa, Ontario, Canada, 2013.
- [225] Pengfei Wu and Murray Woodside. An aggregation approach to constructing hybrid layered queueing models. In *Proc. 7th Int. Workshop on Performability Modeling of Computer and Communication Systems (PMCCS7)*, Torino, Italy, September 2005.
- [226] Jing Xu, Alexandre Oufimtsev, Murray Woodside, and Liam Murphy. Performance modeling and prediction of enterprise javabeans with layered queueing network templates. *SIGSOFT Softw. Eng. Notes*, 31(2), September 2005.
- [227] Xu Zhang and Chung-Horng Lung. Improving software performance and reliability with an architecture-based self-adaptive framework. In *Computer Software and Applications Conference (COMPSAC), 2010 IEEE 34th Annual*, pages 72–81. IEEE, 2010.

- [228] Xu Zhang, Chung-Horng Lung, and Greg Franks. Towards architecture-based autonomic software performance engineering. In *CAL*, pages 144–156, 2010.
- [229] Armin Zimmermann. Modeling and evaluation of stochastic Petri nets with TimeNET 4.1. In *Performance Evaluation Methodologies and Tools (VALUE-TOOLS), 2012 6th International Conference on*, pages 54–63, Oct 2012.
- [230] Armin Zimmermann, Jörn Freiheit, Reinhard German, and Günter Hommel. Petri Net modelling and performability evaluation with TimeNET 3.0. In *Computer Performance Evaluation. Modelling Techniques and Tools*, volume 1786 of *Lecture Notes in Computer Science*, pages 188–202. Springer Berlin Heidelberg, 2000.

Appendix A

Randomized Model

Listing A.1 shows complete XML input file of the random model used to verified the LQN-DPS approach in Section 7.3.3.2. This model is shown in Figure 7.5.

Listing A.1: XML input file for webserver model in Figure.

```
1<?xml version="1.0"?>
2<!-- lqn2lqx -experiments=100 -customers=3 -think-time=3 -
    service-time=1 -no-request-rate -lqx-output -->
3<lqn-model name="" description="lqn2lqx_6.0_solution_for_
    model_from:_" xmlns:xsi="http://www.w3.org/2001/
    XMLSchema-instance" xsi:noNamespaceSchemaLocation="/usr/
    local/share/lqns/lqn.xsd">
4    <solver-params comment="lqngen_--random_--layers=3_--
        clients=3_--tasks=8_--processors=2" conv_val="1e-05"
        it_limit="50" underrelax_coeff="0.9" print_int="10"/>
5    <processor name="c0" scheduling="inf">
6        <task name="c0" scheduling="ref" multiplicity="$c_c0"
            think-time="$Z_c0">
7            <entry name="c0" type="PH1PH2">
8                <entry-phase-activities>
```

```

9          <activity name="c0_1" phase="1" host-demand-
            mean="$s_c0_1">
10          <synch-call dest="e0" calls-mean="1"/>
11          </activity>
12        </entry-phase-activities>
13      </entry>
14    </task>
15  </processor>
16  <processor name="c1" scheduling="inf">
17    <task name="c1" scheduling="ref" multiplicity="$c_c1"
        think-time="$Z_c1">
18      <entry name="c1" type="PH1PH2">
19        <entry-phase-activities>
20          <activity name="c1_1" phase="1" host-demand-
            mean="$s_c1_1">
21          <synch-call dest="e1" calls-mean="1"/>
22          </activity>
23        </entry-phase-activities>
24      </entry>
25    </task>
26  </processor>
27  <processor name="c2" scheduling="inf">
28    <task name="c2" scheduling="ref" multiplicity="$c_c2"
        think-time="$Z_c2">
29      <entry name="c2" type="PH1PH2">
30        <entry-phase-activities>
31          <activity name="c2_1" phase="1" host-demand-
            mean="$s_c2_1">
32          <synch-call dest="e2" calls-mean="1"/>

```

```

33             </activity>
34         </entry-phase-activities>
35     </entry>
36 </task>
37 </processor>
38 <processor name="p1" scheduling="cfs" multiplicity="$
    m_p1" quantum="0.1">
39     <group name="group1" share="$sh1" cap="1">
40         <task name="t0" scheduling="fcfs" multiplicity="$
            c_t0">
41             <entry name="e0" type="PH1PH2">
42                 <entry-phase-activities>
43                     <activity name="e0_1" phase="1" host-
                        demand-mean="$s_e0_1">
44                         <synch-call dest="e3" calls-mean="1"/>
45                     </activity>
46                 </entry-phase-activities>
47             </entry>
48         </task>
49         <task name="t3" scheduling="fcfs" multiplicity="$
            c_t3">
50             <entry name="e3" type="PH1PH2">
51                 <entry-phase-activities>
52                     <activity name="e3_1" phase="1" host-
                        demand-mean="$s_e3_1">
53                         <synch-call dest="e5" calls-mean="1"/>
54                         <synch-call dest="e7" calls-mean="1"/>
55                     </activity>
56                 </entry-phase-activities>

```

```

57         </entry>
58     </task>
59     <task name="t5" scheduling="fcfs" multiplicity="$
        c_t5">
60         <entry name="e5" type="PH1PH2">
61             <entry-phase-activities>
62                 <activity name="e5_1" phase="1" host-
                    demand-mean="$s_e5_1"/>
63             </entry-phase-activities>
64         </entry>
65     </task>
66     <task name="t7" scheduling="fcfs" multiplicity="$
        c_t7">
67         <entry name="e7" type="PH1PH2">
68             <entry-phase-activities>
69                 <activity name="e7_1" phase="1" host-
                    demand-mean="$s_e7_1"/>
70             </entry-phase-activities>
71         </entry>
72     </task>
73 </group>
74 <group name="group2" share="$sh2" cap="1">
75     <task name="t1" scheduling="fcfs" multiplicity="$
        c_t1">
76         <entry name="e1" type="PH1PH2">
77             <entry-phase-activities>
78                 <activity name="e1_1" phase="1" host-
                    demand-mean="$s_e1_1">
79                 <synch-call dest="e4" calls-mean="1"/>

```



```

80             </activity>
81         </entry-phase-activities>
82     </entry>
83 </task>
84 <task name="t2" scheduling="fcfs" multiplicity="$
      c_t2">
85     <entry name="e2" type="PH1PH2">
86         <entry-phase-activities>
87             <activity name="e2_1" phase="1" host-
                  demand-mean="$s_e2_1"/>
88         </entry-phase-activities>
89     </entry>
90 </task>
91 </group>
92 </processor>
93 <processor name="p0" scheduling="ps" multiplicity="$m_p0
      " quantum="0.1">
94     <task name="t4" scheduling="fcfs" multiplicity="$c_t4
      ">
95         <entry name="e4" type="PH1PH2">
96             <entry-phase-activities>
97                 <activity name="e4_1" phase="1" host-demand-
                      mean="$s_e4_1">
98                     <synch-call dest="e6" calls-mean="1"/>
99                 </activity>
100             </entry-phase-activities>
101         </entry>
102     </task>

```

```

103      <task name="t6" scheduling="fcfs" multiplicity="$c_t6
      ">
104      <entry name="e6" type="PH1PH2">
105          <entry-phase-activities>
106              <activity name="e6_1" phase="1" host-demand-
                  mean="$s_e6_1"/>
107          </entry-phase-activities>
108      </entry>
109  </task>
110 </processor>
111 <lqx><![CDATA[
112 $c_t0 = 1;
113 $m_p1 = 1;
114 $c_t1 = 1;
115 $m_p1 = 1;
116 $c_t2 = 1;
117 $m_p1 = 1;
118 $c_t3 = 1;
119 $m_p1 = 1;
120 $c_t4 = 1;
121 $m_p0 = 1;
122 $c_t5 = 1;
123 $m_p1 = 1;
124 $c_t6 = 1;
125 $m_p0 = 1;
126 $c_t7 = 1;
127 $m_p1 = 1;
128 $sh1 = 0.6;
129 $sh2 = 0.4;

```

```
130     j = 0;
131
132     for ( i = 0; i < 10; i = i + 1 ) {
133         $c_c0 = poisson( 2 ) + 1;
134         $Z_c0 = gamma( 1.5, 2 );
135         $s_c0_1 = gamma( 0.5, 2 );
136         $c_c1 = poisson( 2 ) + 1;
137         $Z_c1 = gamma( 1.5, 2 );
138         $s_c1_1 = gamma( 0.5, 2 );
139         $c_c2 = poisson( 2 ) + 1;
140         $Z_c2 = gamma( 1.5, 2 );
141         $s_c2_1 = gamma( 0.5, 2 );
142         $s_e0_1 = gamma( 0.5, 2 );
143         $s_e1_1 = gamma( 0.5, 2 );
144         $s_e2_1 = gamma( 0.5, 2 );
145         $s_e3_1 = gamma( 0.5, 2 );
146         $s_e4_1 = gamma( 0.5, 2 );
147         $s_e5_1 = gamma( 0.5, 2 );
148         $s_e6_1 = gamma( 0.5, 2 );
149         $s_e7_1 = gamma( 0.5, 2 );
150         solve();
151         j += 1;
152         g1_util =task("t0").proc_utilization
153             + task("t3").proc_utilization
154             + task("t5").proc_utilization
155             + task("t7").proc_utilization;
156
157         g2_util =task("t1").proc_utilization
158             + task("t2").proc_utilization;
```

```
159      println_spaced( " , " , j ,
160                      $sh1 ,
161                      $sh2 ,
162                      task("c0").throughput ,
163                      task("c1").throughput ,
164                      task("c2").throughput ,
165                      processor("p1").utilization ,
166                      task("t0").utilization ,
167                      task("t3").utilization ,
168                      task("t5").utilization ,
169                      task("t7").utilization ,
170                      task("t1").utilization ,
171                      task("t2").utilization ,
172                      task("t4").utilization ,
173                      task("t6").utilization ,
174      g1_util , g2_util);
175  }
176  ]]>
177  </lqx>
178 </lqn-model>
```