Simplifying Layered Queueing Network Models

by

Farhana Islam

Thesis
Submitted to the Faculty of Graduate 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

© 2018, Farhana Islam
Abstract

The amount of detail to include in a performance model is usually regarded as a judgment to be made by an expert modeler and the question “how much detail is necessary?” is seldom asked and is difficult to answer. However, if a simpler model gives essentially the same performance predictions, it may be more useful than a detailed model. It may solve more quickly, for instance, and may be easier to understand. Or a model for a complex subsystem such as a database server may be usefully simplified so it can be included in larger system models. This research proposes an aggregation process for layered queuing models that reduces the number of queues (called tasks and hosts/processors, in layered models) while preserving the total execution demand and the bottleneck characteristics (the highest saturated resource(s)) of the detailed model. This thesis demonstrates how the simplification process can greatly reduce the number of tasks and processors with a very small relative error. The application of proposed simplification process is applied on a number of case studies. A Java application called “LQN model simplifier” was built that takes an LQN model as input and generates a series of simplified models after applying simplification operations to the original model.
Acknowledgements

Behind this journey, there are a lot of dedication, effort, patience and hard work. I have immense amount of gratefulness towards a lot of people which I will try to summarize in few words below.

At first, I would like to express my sincere gratitude to my co-supervisors Prof. Murray Woodside and Prof. Dorina Petriu for their continuous support of my Ph.D. study and related research, for their patience, motivation, and immense knowledge. Their guidance and expert advice helped me in all the time of research and writing of this thesis. I could not have imagined having better advisors and mentors for my Ph.D. study. I would like to thank Prof. Greg Franks also for his invaluable help with the LQNS tool.

I never imagined that I would pursue my Ph.D. ever. It was my idol - my father Prof. Md. Nazmul Islam who dreamt of this and encouraged me to go for it. His strong compassion, encouragement, inspiration and most importantly the trust on me made me who I am today. I really wish if he was alive today to see his daughter fulfilled his dream. I could not be able to achieve this without the continuous prayer of my mother Begum Akhter Jahan. Although she is very sick and cannot remember everything, she always remembers to pray for me. I am so grateful and proud to have such great parents.

A large amount of credits for this work goes to my beloved husband Dr. Adnan Faisal for his patience, understanding, love and support in all my good times and bad times. He never gave up on me though I was about to give up a lot of times during this long journey, especially when my father passed away. His continuous counseling to raise my confidence and to believe in myself helped me immensely. He took care of our son and our little family
when I was busy. Because of his great knowledge on this field, his tips and ideas helped me a lot in my research and experiments.

My son Farhaz Faisal has a very special role behind this work. He is the one who always used to cheer me up after a long day’s work. Playing with him and listening to his never ending stories always refreshed me. This work is achieved because of his sacrifice and understanding of not being with me all the time.

I am so grateful to my sister Dr. Rumana Islam for giving me the immense love and support during my research and always. I am thankful to my mother-in-law Halima Khatun for all her prayers and encouragement. Special thanks go to my very dear friend Taghreed Altamimi for her encouraging words and motivations. I am and will be forever grateful to all my family and friends for their enormous support on this journey.

Lastly, I cannot thank enough God Almighty for giving me the precious opportunity of pursuing my Ph.D. research.
This work is dedicated to

my beloved father Prof. Md. Nazmul Islam

my dear husband Dr. Adnan Faisal

and

my lovely son Farhaz Faisal
# Table of Contents

Abstract .................................................................................................................................................. ii
Acknowledgements .......................................................................................................................... iii
Table of Contents ............................................................................................................................. vi
List of Tables ........................................................................................................................................ ix
List of Figures ........................................................................................................................................ x
Listings .................................................................................................................................................. xiv
Notations ............................................................................................................................................... xv

## 1 Chapter: Introduction .................................................................................................................... 1

1.1 Motivations .................................................................................................................................... 1
1.2 Objectives ..................................................................................................................................... 3
1.3 Overview of the solution ............................................................................................................ 5
1.4 Contributions ............................................................................................................................ 6
1.5 Contents of the thesis ................................................................................................................ 7

## 2 Chapter: Background and State of the Art ................................................................................... 8

2.1 Performance modeling ............................................................................................................. 8

### 2.1.1 Petri Nets ............................................................................................................................ 8

### 2.1.2 Stochastic process algebra .............................................................................................. 10

### 2.1.3 Queuing Network (QN) .................................................................................................. 11

### 2.1.4 Layered Queueing Network (LQN) ................................................................................ 12

2.2 State of the art in model simplification .................................................................................... 20

### 2.2.1 Aggregation of activities in LQN models .......................................................................... 20

### 2.2.2 Norton’s theorem .............................................................................................................. 21

### 2.2.3 Flow equivalent server method ....................................................................................... 21

### 2.2.4 Hierarchical decomposition ............................................................................................. 22
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Pages</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.2.5</td>
<td>Surrogate delay methods</td>
<td>22</td>
</tr>
<tr>
<td>2.2.6</td>
<td>Simplification methods in Petri Nets</td>
<td>23</td>
</tr>
<tr>
<td>2.2.7</td>
<td>Regression methods</td>
<td>24</td>
</tr>
<tr>
<td>2.2.8</td>
<td>Shadow servers</td>
<td>25</td>
</tr>
<tr>
<td>2.2.9</td>
<td>Summary</td>
<td>25</td>
</tr>
<tr>
<td>3</td>
<td>Chapter: LQN model simplification process</td>
<td>26</td>
</tr>
<tr>
<td>3.1</td>
<td>Model simplification principles</td>
<td>27</td>
</tr>
<tr>
<td>3.2</td>
<td>Single-Group Aggregation</td>
<td>29</td>
</tr>
<tr>
<td>3.2.1</td>
<td>Single-Group Aggregation: Example 1</td>
<td>30</td>
</tr>
<tr>
<td>3.2.2</td>
<td>Single-Group Aggregation and Task/Processor Aggregation: Example 2</td>
<td>38</td>
</tr>
<tr>
<td>3.3</td>
<td>Bottleneck preservation</td>
<td>43</td>
</tr>
<tr>
<td>3.4</td>
<td>Single-Group Aggregation algorithm</td>
<td>44</td>
</tr>
<tr>
<td>3.5</td>
<td>Limitations of Single-Group Aggregation</td>
<td>46</td>
</tr>
<tr>
<td>4</td>
<td>Chapter: Dependency-Group Aggregation</td>
<td>47</td>
</tr>
<tr>
<td>4.1</td>
<td>Groups of tasks for aggregation</td>
<td>47</td>
</tr>
<tr>
<td>4.2</td>
<td>Tasks grouped by their dependencies</td>
<td>49</td>
</tr>
<tr>
<td>4.3</td>
<td>Desirable properties of the aggregation groups</td>
<td>51</td>
</tr>
<tr>
<td>4.4</td>
<td>Aggregation of a group of tasks</td>
<td>53</td>
</tr>
<tr>
<td>4.5</td>
<td>Dependency-Group Aggregation vs Single-Group Aggregation</td>
<td>54</td>
</tr>
<tr>
<td>4.6</td>
<td>An incremental strategy for error control</td>
<td>62</td>
</tr>
<tr>
<td>4.7</td>
<td>Dependency-Group Aggregation algorithm</td>
<td>63</td>
</tr>
<tr>
<td>5</td>
<td>Chapter: Dependency-Group Aggregation with Shared Deployment</td>
<td>66</td>
</tr>
<tr>
<td>5.1</td>
<td>Calling cycles and processor splitting</td>
<td>66</td>
</tr>
<tr>
<td>5.2</td>
<td>Split processors</td>
<td>69</td>
</tr>
<tr>
<td>5.3</td>
<td>Strategies for aggregating tasks with shared deployment</td>
<td>72</td>
</tr>
</tbody>
</table>
5.4 Application of split processor .......................................................... 73
5.5 Algorithm for Dependency-Group Aggregation with Shared Deployment .... 77

6 Chapter: Experiments and Case Studies ................................................. 81
6.1 Empirical evaluation of simplification accuracy .................................. 81
6.2 Scalability ......................................................................................... 90
6.3 Case studies .................................................................................... 93
  6.3.1 Case Study 1: Business Reporting System ..................................... 93
  6.3.2 Case study 2: Airline Reservation System ....................................... 96
  6.3.3 Case Study 3: Linux Network File System ..................................... 102
  6.3.4 Case Study 4: Large Telephony Switch ......................................... 106
  6.3.5 Case Study 5: E-Commerce .......................................................... 110
  6.3.6 Summary of case studies .............................................................. 112

7 Chapter: Automation and Tooling ....................................................... 114
7.1 Software requirements ....................................................................... 114
7.2 Software architecture ....................................................................... 114
7.3 Use of the Model Simplifier ............................................................... 119

8 Chapter: Conclusions ........................................................................ 120
8.1 The approach .................................................................................. 120
8.2 Major findings ................................................................................ 121
8.3 Contributions .................................................................................. 122
8.4 Limitations ..................................................................................... 123
8.5 Future work .................................................................................... 124

References ............................................................................................ 125
List of Tables

Table 3.1 Pre conditions and post conditions of simplification process ................................ 30
Table 3.2 Resource utilizations after each simplification operation of Example1 .......... 37
Table 3.3 System throughput and response time after each simplification operation of 
Example1 ........................................................................................................................................ 37
Table 3.4 Effects of the simplification operations on system response time and 
throughput for Example2 ................................................................................................................. 41
Table 3.5 Effects of simplification operations on utilizations of resources of Example2 41
Table 3.6 Effects of aggregating only non-bottleneck processors in Example2......... 42
Table 3.7 Utilization of resources after aggregating only non-bottleneck processors in 
Example2 .......................................................................................................................................... 42
Table 4.1 Errors of different aggregations of Example 5 .................................................. 63
Table 5.1 Service time of all the tasks originally deployed on processors p5, p6 and p8 
before and after splitting the processors and their % error ...................................................... 77
Table 6.1 Errors of different aggregations of “case-50” ....................................................... 92
Table 6.2 Model simplification results for different aggregations on Case study 2 ...... 102
Table 6.3 Model simplification results for different aggregations on Case study 3 ...... 106
Table 6.4 Model simplification results for different aggregations on Case study 4 ...... 110
List of Figures

Figure 1.1 Layered Queueing Network of the Business Reporting System generated from a software specification in PCM [34] ................................................................. 2

Figure 2.1 A Petri Net with resource contention [41] ........................................... 9

Figure 2.2 A single-line single-server system ...................................................... 11

Figure 2.3 LQN model of a three tier application ............................................ 15

Figure 2.4 An example of LQN model containing activity diagram in entries [45] .... 17

Figure 2.5 An LQN model with cycle between two entries of two tasks .............. 18

Figure 3.1 Example 1 and the result of aggregating activities ............................ 32

Figure 3.2 Aggregating entries and tasks in Example 1 ...................................... 35

Figure 3.3 An LQN model of eShop .................................................................. 39

Figure 3.4 Aggregation of processors and all their tasks on eShop ...................... 40

Figure 3.5 Aggregating bottleneck with a non-bottleneck from Example 1 ......... 43

Figure 4.1 LQN model showing tasks with PT-dependency set \{P1, P2\} ............. 50

Figure 4.2 “case-A” showing groups defined by four distinct PT-Dependency sets $P = \{U\}, \{TP_1\}, \{TP_2\}, \{U, TP_2\}$ ................................................................. 52

Figure 4.3 Aggregated model for “case-A” using the dependency groups shown in Fig. 4.2 ........................................................................................................... 52

Figure 4.4 LQN model of Example 3 ................................................................. 55

Figure 4.5 First aggregated model for Example 3 shown in Fig. 4.4, following the Single-Group Aggregation ................................................................. 56

Figure 4.6 Second aggregated model for Example 3 (shown in Fig. 4.4) by generating two groups preserving one bottleneck task .................................................... 57
Figure 4.7 Third aggregated model for Example 3 by generating groups preserving two heavily saturated tasks ................................................................. 58

Figure 4.8 LQN model Example 4 with bottleneck processor $p09$ and its deployed task $t09$ in bold outline and second bottleneck task $t05$ in dashed outline.......................... 59

Figure 4.9 Aggregated LQN model for Example 4 (shown in Fig. 4.8) following the Single-Group Aggregation................................................................. 60

Figure 4.10 Aggregated model from Example 4 of Fig. 4.8 by generating groups preserving bottleneck processor and a second bottleneck task ......................... 60

Figure 4.11 Original LQN model of Example 5 with 10 tasks and 10 processors .... 61

Figure 4.12 First aggregated model of Example 5 from Fig. 4.11 preserving task $t2$ ...... 62

Figure 5.1 An LQN example to show shared deployment of tasks $t3$ and $t6$ on the processor $p3$ ........................................................................................................... 67

Figure 5.2 Tasks $t3$ and $t6$ are aggregated as $mergedTask1$ on processor $p3$ creating a calling cycle ................................................................. 68

Figure 5.3 Model of Fig. 5.1 after splitting processor $p3$ .............................................. 69

Figure 5.4 LQN model of Example 6 ................................................................................. 75

Figure 5.5 LQN model after splitting all the processors in Fig. 5.4 ......................... 76

Figure 6.1 Trajectory of accuracy and size reduction as preserved tasks are added ...... 82

Figure 6.2 Preserved tasks vs. reduction in model size for 300 cases ....................... 83

Figure 6.3 Throughput error vs. preserved tasks for 300 cases ................................ 84

Figure 6.4 Throughput error vs. reduction in model size for 60 cases containing 11 tasks ................................................................................................. 85
Figure 6.5 Throughput error vs. reduction in model size for 60 cases containing 16 tasks ......................................................... 86
Figure 6.6 Throughput error vs. reduction in model size for 60 cases containing 21 tasks .......................................................... 87
Figure 6.7 Throughput error vs. reduction in model size for 60 cases containing 26 tasks .................................................................................................................. 87
Figure 6.8 Throughput error vs. reduction in model size for 60 cases containing 31 tasks .................................................................................................................. 88
Figure 6.9 Average reduction for a certain model size and error in 300 cases ............... 89
Figure 6.10 Average number of cases simplified in a certain model size and error in 300 cases .................................................................................................................. 89
Figure 6.11 LQN model of “case-50” with 50 tasks and 50 processors ......................... 91
Figure 6.12 First aggregated model of “case-50” with 7 tasks and 7 processors .......... 92
Figure 6.13 Second aggregated model of “case-50” with 9 tasks and 9 processors ....... 93
Figure 6.14 Layered Queueing Network of the Business Reporting System generated from PCM .................................................................................................................. 94
Figure 6.15 System Throughput (jobs/ms) of case study 1 after three simplification operations .................................................................................................................. 95
Figure 6.16 Relative error in system throughput after every simplification operation..... 95
Figure 6.17 System response time after three simplification operations ...................... 96
Figure 6.18 Relative error in system response time after each simplification operations 96
Figure 6.19 ARS_v4 where REST and ActiveMQ are used together [11]................. 98
Figure 6.20 First aggregated model of ARS_v4 where bottleneck resource ServiceH is preserved, deployed tasks on it are merged ................................................................. 99

Figure 6.21 Second aggregated model of ARS_v4 where ServiceH and ControllerH are preserved, deployed tasks on each processor are merged ........................................ 100

Figure 6.22 Third aggregated model of ARS_v4 where ServiceH, ControllerH and CatalogH are preserved, deployed tasks on each processor are merged ...................... 101

Figure 6.23 Layered Queueing Network of principle NFS operations [14] ..................... 103

Figure 6.24 First aggregated model of case study 3, preserving task rpcnfsd ................. 104

Figure 6.25 Second aggregated model of case study 3, preserving task rpcnfsd and processor server ................................................................................................................. 105

Figure 6.26 An LQN model of Telephony Switch Architecture [15] ............................... 107

Figure 6.27 A simplified model of Case Study 4 with 6 tasks and 6 processors with 5.21% throughput error, 4.95% response time error and 70.70% reduction .................. 108

Figure 6.28 Final simplified model of Case Study 4 ..................................................... 109

Figure 6.29 LQN model of an E-Commerce example .................................................. 111

Figure 6.30 Simplified model of Case Study 5 ............................................................ 112

Figure 6.31 Throughput error and reduction in five case studies ............................... 113

Figure 7.1 Class diagram of “LQN model simplifier” .............................................. 115
Listings

Listing 3.1 Summarized algorithm for Single-Group Aggregation ........................................ 44
Listing 4.1 Summarized algorithm for Dependency-Group Aggregation .......................... 64
Listing 5.1 Summarized algorithm for shared deployment .................................................. 77
## Notations

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$Y_E$</td>
<td>The number of calls to entry $E$, per user request</td>
</tr>
<tr>
<td>$y_{E1,E2}$</td>
<td>Mean calls to $E2$ per invocation of $E1$</td>
</tr>
<tr>
<td>$X_E$</td>
<td>The service time of entry $E$</td>
</tr>
<tr>
<td>$d_H$</td>
<td>Demand of each core in host $H$, per user response</td>
</tr>
<tr>
<td>$d_E$</td>
<td>Mean host demand in time units per invocation of entry $E$</td>
</tr>
<tr>
<td>$d_T$</td>
<td>Demand of each thread of task $T$, per user response</td>
</tr>
<tr>
<td>$m_H$</td>
<td>Multiplicity or number of threads of host $H$</td>
</tr>
<tr>
<td>$m_T$</td>
<td>Multiplicity or number of threads of task $T$</td>
</tr>
<tr>
<td>$S_H$</td>
<td>Speed factor of host $H$</td>
</tr>
<tr>
<td>$c_H$</td>
<td>Capacity of host $H$</td>
</tr>
<tr>
<td>$Z_{User}$</td>
<td>Think time of the reference task $User$</td>
</tr>
<tr>
<td>$\lambda_E$</td>
<td>Throughput of entry $E$ invocations/s</td>
</tr>
<tr>
<td>$\lambda_T$</td>
<td>Throughput of task $T$ invocations/s</td>
</tr>
<tr>
<td>$U_E$</td>
<td>Utilization of entry $E$</td>
</tr>
<tr>
<td>$U_T$</td>
<td>Utilization of task $T$</td>
</tr>
<tr>
<td>$U_H$</td>
<td>Utilization of host $H$</td>
</tr>
<tr>
<td>$SL_H$</td>
<td>Saturation level of host $H$</td>
</tr>
<tr>
<td>$TP$</td>
<td>Set of tasks to be preserved</td>
</tr>
<tr>
<td>$TP$</td>
<td>Preserved task</td>
</tr>
<tr>
<td>$PT-Dependent$</td>
<td>Preserved-Task Dependent</td>
</tr>
<tr>
<td>$T_1 \prec T_2$</td>
<td>Task $T_1$ is Preserved-Task Dependent ($PT-Dependent$) on task $T_2$</td>
</tr>
<tr>
<td>Symbol</td>
<td>Description</td>
</tr>
<tr>
<td>------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>$T_1 \parallel T_2$</td>
<td>Task $T_1$ has no dependency relation with $T_2$</td>
</tr>
<tr>
<td>$Pi$</td>
<td>$PT$-dependency set</td>
</tr>
<tr>
<td>$G(P)$ or $G$</td>
<td>Group of tasks that have the same $PT$-dependency set</td>
</tr>
<tr>
<td>$TA(G)$</td>
<td>Aggregated task created by aggregating the tasks in $G(P)$</td>
</tr>
<tr>
<td>$DA(G)$</td>
<td>Demand of the aggregated task created from $G(P)$, $TA(G(P))$</td>
</tr>
<tr>
<td>$YA(G)$</td>
<td>The invocations of $G$ per user response (or “total calls” to $G$)</td>
</tr>
<tr>
<td>$HA(TA(G))$</td>
<td>Aggregated host created by aggregating the hosts in $G$</td>
</tr>
<tr>
<td>$m_{HA(TA(G))}$</td>
<td>Multiplicity of aggregated host $HA(TA(G))$</td>
</tr>
<tr>
<td>$S_{HA(TA(G))}$</td>
<td>Speed factor of aggregated host $HA(TA(G))$</td>
</tr>
<tr>
<td>$R_i$</td>
<td>Response time of entry $E_i$ of task $T_i$</td>
</tr>
<tr>
<td>$f$</td>
<td>System throughput</td>
</tr>
<tr>
<td>$SP(T_i)$</td>
<td>Split-processor of Task $T_i$</td>
</tr>
<tr>
<td>$SAF_{SP(T_i)}$</td>
<td>Speed adjusting factor of $SP(T_i)$</td>
</tr>
<tr>
<td>$U$</td>
<td>Total utilization of all the tasks on the host $H$</td>
</tr>
<tr>
<td>$SSP$</td>
<td>Speed factor of split processor</td>
</tr>
<tr>
<td>$DAF_{SP(T_i)}$</td>
<td>Demand adjusting factor of $SP(T_i)$</td>
</tr>
</tbody>
</table>
1 Chapter: Introduction

Performance models are used to predict the performance and scalability of a system before it is completed and deployed, and methods have been devised to create such models from software specifications or architectural designs [43] [4] [40].

Quite often performance models can be very large and complex, because they include all the design details, while only some details are significant for performance. The models may have unnecessary queueing servers that have very small impact on performance. Sometimes performance models are generated from software models in which some performance model elements could be uninteresting to the system designer. Thus, there is a need for compact models that represent only the elements that are important for the system performance.

1.1 Motivations

A performance model may include a very large amount of detail about resources and operations, which makes it difficult to create because there could be complex relationships between operations and resources. Such a performance model is difficult to maintain too, because for a small change in the model, we may need to re-estimate a large number of model parameters. A large and complex performance model is also hard to understand by its stakeholders (i.e., software architects, system designers, software developers, etc.). For example, if a performance modeler builds a performance model such as in Figure 1.1 (taken from [34]), it is very difficult for other stakeholders to understand this model, due to its inherent complexity caused by the large number of performance model elements. This is often true when a software model is automatically transformed into a performance model [1], because such performance models may contain many model elements that are needed
Figure 1.1 Layered Queueing Network of the Business Reporting System generated from a software specification in PCM [34]

for a software model, but have very little impact on performance. During the transformation process of a software model into a performance model, for every operation - even if very
small, a separate performance model element is created in the performance model. Ideally, performance models should be kept small and high-level, due to the fact that more iterations and execution time are needed for solving large performance models.

1.2 Objectives

The objective of this research is to develop a process for automatically simplifying a performance model to an essential core level of detail governed by accuracy requirements. The accuracy requirements include retaining the performance bottleneck(s) from the original model and keeping the relative error in throughput and response time as small as possible.

1.1 Scope and Problem Statement

The problem addressed in this thesis is to simplify Layered Queueing Network models of complex distributed software systems and subsystems. The LQN model [13] [38] [42] was developed as an extension of the well-known queueing network model. The main difference is that LQN can represent nested services (i.e., a server may also be a client to other servers). An LQN model is a graph whose nodes are either software tasks (shown as parallelograms) or hardware devices (shown as circles) and the arcs denote service requests, as illustrated in Figures 1.1 and 2.3. There are three types of service requests: synchronous (filled arrow), asynchronous (stick arrow) and forwarding (dotted arrow). The nodes with outgoing but no incoming arcs play the role of user tasks; the intermediate nodes with both incoming and outgoing arcs are usually software servers; and the leaf nodes are hardware servers. A software or hardware server node can be either a single-server (with multiplicity = 1) or a multi-server (with multiplicity > 1). Software tasks have one or more entries (shown as smaller parallelograms inside the tasks) corresponding to
different operations or services they offer. A server entry may contain one or more sequential phases of service; each phase has an execution time demand for CPU and makes service requests to other task entries. Phase 1 is the portion of the operation when the client is blocked waiting for a reply from the server (assuming that the client has made a synchronous request). At the end of phase 1, the server replies to the client, which will unblock and continue its execution. The remaining phases, if any, will be executed in parallel with the client. An extension to LQN [14] allows for an entry to be decomposed into several activities if more details are required to describe its execution, as shown in Figure 1.1. The activities are connected together into a directed graph, which may branch into parallel threads of control or may choose randomly between different branches. Just like phases, activities have execution time demands, and can make service requests to other entries. Although not explicitly shown in the LQN notation, every server (software or hardware) has an implicit message queue where incoming requests for any offered service are waiting their turn. More details on LQN are discussed in Section 2.1.4.

The LQN models considered in this thesis represent software and hardware resources, restricted in four ways: a) there is a single class of user tasks (of any size); b) there is no parallel execution within a single task (although there may be unlimited concurrency in processing responses to separate requests); c) there is no calling cycle in the requests passed between tasks; and d) each entry has a single phase of execution (that is, the operation by a server ends when a reply is sent to the requesting entity).

The simplification process should satisfy accuracy requirements, while retaining enough detail to solve useful performance problems. The process should be automated.

The hypothesis of the thesis is that simplification of the model by aggregation of some
model entities can provide sufficient accuracy. Aggregation provides useful traceability such that each original entity is associated with an identified aggregated entity, and entities which are not aggregated are kept unchanged in the simplified model.

The following challenges of this research have been identified:

1. Identify the model details that can safely be suppressed, and propose an approach for representing their contribution to performance.
2. The low-load performance should be correct, which requires representing the total execution delay for a response.
3. The saturation performance should be correct, which requires identifying and representing the bottleneck elements.
4. The most useful degree of simplification for a given model will depend on the model user’s requirements for accuracy and simplicity which must be incorporated in a stopping rule.
5. The process should be scalable to large models, since those are the ones that need it most.

1.3 Overview of the solution

In this research, an automated step-by-step simplification process of Layered Queuing Network models is proposed. The process starts with an initial performance model and after every step some model elements are aggregated resulting in a simplified model. This simplification process is applicable to Layered Queuing Network models of systems with a single class of users, and with distributed and layered operations and resources. The simplification process is based on merging/aggregating LQN model elements. In this
thesis, the terms ‘merge’ and ‘aggregation’ are used interchangeably.

The thesis proposes two approaches for simplification. The first approach called Single-Group Aggregation aggregates only two tasks at a time and preserves only one bottleneck task/processor. The second approach called Dependency-Group Aggregation is more robust. A number of tasks and processors including bottlenecks are preserved and the other tasks are grouped and aggregated.

1.4 Contributions

The contributions of this thesis are summarized as follows:

1. Single-Group Aggregation of LQN models (Chapter 3): An LQN model simplification process is developed that aggregates sub-operations (activities in LQN terms), operations (entries in LQN terms), software processes (tasks in LQN terms), and physical resources (processors or hosts in LQN terms) in models with a single class of users, preserving one bottleneck task/processor.

2. Dependency-Group Aggregation of LQN models (Chapter 4 & 5): This is an improved approach of model simplification process where a number of tasks and processors are preserved based on their level of saturation. Tasks are grouped under each preserved task and merged in one step.
   
   An extension of Dependency-Group Aggregation (presented in Chapter 5) is built that handles the tasks that are deployed on a common processor. The process is known as Dependency-Group Aggregation with Shared Deployment.

3. Experiments and Case Studies (Chapter 6): Evaluating of the simplification approach is done through experiments on a large number of models that cover the space of architectures and parameters.
4. Tool Design and Implementation (Chapter 7): A tool is developed that can automatically (a) identify bottlenecks and preserved tasks and (b) generate the simplified model according to different strategies.

1.5 Contents of the thesis

Chapter 1 (this chapter) presents the motivation, objective, scope, challenges and contributions of the thesis. Chapter 2 discusses background and state of the art of the thesis. Chapter 3 introduces model simplification approaches and presents Single-Group Aggregation, which is described with the help of some running examples. Chapter 4 describes Dependency-Group Aggregation with some examples. It also presents a comparison between single group and dependency group aggregation. Chapter 5 presents the approach called Dependency-Group Aggregation with Shared Deployment. With some examples, it shows the need for this approach and how the aggregation is done. Chapter 6 presents all the experiments and results throughout this research. A large number of empirical experiments are done to validate the process of simplification. This chapter also presents a number of case studies that are done. In Chapter 7, the automation and implementation of the proposed methodology are described. Finally conclusions and future work are presented in Chapter 8.
2 Chapter: Background and State of the Art

This Chapter describes the context of the research under two sections. At first, the chapter presents the background of the research, which refers to a number of methodologies for performance modeling. Then, it presents the state of the art in model simplification approaches which are related to the proposed methodology.

2.1 Performance modeling

The role of performance models is to make predictions for the performance of systems that do not yet exist, either using a model derived from a specification [43], or by studying modifications of an existing system and model [4]. There exist a number of techniques for modeling and performance analysis. In this section, some relevant performance modeling techniques are discussed.

2.1.1 Petri Nets

Petri nets provide a state-based modeling formalism which is convenient and intuitive to create and manage.

A Petri net is defined [18] [41] as a combined state-event model with a state space being represented by tokens in “places” and events by “transitions” which move the tokens from their input to output places. Formally, a Petri net is defined by a 5 tuple \( N = (P, T, I, O, M_0) \) where \( P \) is a set of places, \( T \) is a set of transitions, \( I \) is a set of input arcs from places to transitions, \( O \) is a set of output arcs from transitions to places and \( M_0 \) is a vector giving the initial marking of tokens in places. A marking is an assignment of tokens to the places of a Petri net. Petri nets are used in Computer Science and many other disciplines to model state-based systems and resource contention. They combine a well-defined mathematical theory with a graphical representation of the dynamic behavior of the system [41]. The
mathematical aspect of Petri nets allows for precise modeling and analysis of system behavior whereas the graphical representation of Petri nets allows for the visualization of the modeled system state changes.

The execution of a Petri net is controlled by the distribution and number of tokens in the Petri net. A Petri net executes by firing transitions that govern the flow of tokens. A Petri net can be represented as shown in [41] by a bipartite graph, where a circle represents a place, a bar or box represents a transition and a directed arc connects places with transitions. The directed arcs are there for presenting which places are pre and/or post conditions for which transitions [41].

Figure 2.1 shows how resource contention can be modeled in Petri nets. In this graph, the resource modeled by the place p₁ which is shared by the resources p₂ and p₃ can be held only in a mutually exclusive manner. For example, in this example, p₁ can be assumed to be a robot shared by two machines (p₂ and p₃) for loading and unloading. In such a system, these operations cannot take place at the same time, i.e., they are mutually exclusive.
Standard Petri nets generate very large and unstructured specifications for the systems being modeled because of indistinguishable tokens [41]. As a result, high level Petri nets (such as colored Petri nets and timed Petri nets) were developed [41] to allow compact system representation. According to [18], in Colored Petri nets, tokens have values or extra information representing the identity of the tokens. There is a functional dependency between the transitions and tokens in colored Petri nets. In timed Petri nets [41], different transitions take different time durations to operate. The two most widely used timed Petri nets are deterministic timed Petri nets and Stochastic Petri nets. In deterministic timed Petri nets, transitions have deterministic time delays, and in stochastic timed Petri nets stochastic firing times are associated with transitions [41]. A widely used version of timed nets is called Generalized Stochastic Petri Nets (GSPNs) [32], which allows for immediate and timed transitions.

2.1.2 Stochastic process algebra

Process algebras (i.e. Milner's CCS [36] and Hoare's CSP [20]) are abstract languages used for the specification and design of concurrent systems [8] [19]. They provide three important features which are: \textit{compositionality} (the ability to model a system as the interaction of its subsystems), \textit{formality} (all terms have precise meaning) and \textit{abstraction} (the ability to build complex models from detailed components hiding the internal behavior when it is appropriate to do so). Stochastic extensions to process algebra add time delays for actions and probabilistic branching for choices to the process algebra, so that they can be used for performance modeling [8]. In stochastic process algebra, a random variable is associated with the time duration of every action. The random variables are assumed to be exponentially distributed. Stochastic process algebras are used to study performance
properties of computer and communication systems such as throughput, utilization, response time and also qualitative properties such as freedom from deadlock.

2.1.3 Queuing Network (QN)

A QN model [35] is a collection of a number of interconnected queues. Each queue provides two elements – a waiting line and a resource that provides service to the customers. Customers are grouped into one or more classes. They move from one queue to another until they complete their execution. Figure 2.2 represents a single-line single-server queue [35].

![Figure 2.2 A single-line single-server system](image)

In the figure above, customers arrive at an average arrival rate $\lambda$ per sec, join the queue (waiting line) if the server is busy, wait for their turn to use the server according to a queueing discipline (e.g. FCFS) and depart after receiving the service of duration $S$. The average time spent in waiting in the queue is denoted by $W$, and the response time (i.e., the sum of the average waiting time plus the average service time) is denoted by $T$.

The input parameters of a QN model are divided into two groups [35] – workload intensity and service demands. Workload intensity is an indication of the load of the system which is measured in terms of arrival rates for open classes or number of customers for closed
classes. An open QN model has infinite stream of arriving customers and a closed model has a finite population. Service demand is the total average service time provided by a specific resource. Queuing network models are based on queuing theories (e.g., Mean Value Analysis) to provide mathematical analysis of performance problems.

Queuing network models are often used to analyze system performance for capacity planning. They are used to predict performance metrics so that systems can be reconfigured (optimal selection of various parameters like CPU speed, device capacity etc.). Performance impact on various hardware resource contention can be analyzed using QN models. This feature helps to make business decisions about the resources to achieve satisfactory performances.

2.1.4 Layered Queueing Network (LQN)

Layered Queuing Networks (LQNs) are an elegant way to express simultaneous resource possession in queueing systems (a token in service at one server can request another server). They are particularly intended to model layered software systems, in which a software server depends not just on its processor, but on other software servers as well [35]. Queueing models are used because they account for the effect of contention for resources, which is important in systems under load, and layered queueing network models (LQNs) are adapted to layered software systems. Figure 2.3 shows an example LQN model of a three tiered (three layered) application, discussed further below.

An LQN model describes the interaction of system-elements (which may be any kind of software or hardware entity) via requests for service from one entity to another. The entities in the model are called tasks (analogous to objects) which accept service requests as calls
made to *entries* (analogous to operations). Calls may be realized as procedure calls, RPCs, or synchronous or asynchronous messages over a network. Each task is executed by its host (processor) which may be single server or multiserver (e.g., multi-core or multiprocessor). In this thesis, processor and host are used interchangeably. Tasks and processors are servers with queues [22]. Tasks may be multi-threaded, with threads sharing a queue, and the threads are scheduled on the host by a host queueing discipline. Tasks may be used to model software processes and also other system features which generate contention, such as mutexes, buffer pools, or locks.

LQN models of real systems can be very large, if they describe systems with many servers, replicated servers, and storage devices. Models with a dozen layers and dozens of tasks are common, and hundreds of tasks may arise in complex cases or with large scale-out by replication. These large models are cumbersome and most of the detail does not impact the performance.

Performance, in the sense of response times and throughputs, is determined by how the system’s operations use its resources. System response time is the sum of the average waiting time plus the average service time throughout all the queues in the network and system throughput is the number of jobs exiting the system per unit time. The utilization of a single-threaded resource (task/host) is the fraction of time the task is busy (executing or blocked), meaning not idle [45]. The utilization of a multi-threaded or infinite-threaded resource is the mean number of busy threads of the resource.

Entry *E* is part of a task *T(E)* and has a host demand parameter *d_E* representing the mean host demand in time units per invocation of *E*, as determined for a “nominal” host type. A call (*E_1*, *E_2*) has a call parameter *y_{E_1,E_2}* which represents mean calls to *E_2* per invocation
of \( E1 \). In this work, calls which block the caller (i.e., the calling thread waits for the reply after the entry execution is finished) are considered, but LQN can also model asynchronous calls and calls which are forwarded along a chain of tasks for service.

Task \( T \) has a multiplicity \( m_T \) and a host \( H(T) \). Host \( H \) has a multiplicity \( m_H \) and a speed factor \( S_H \) (speed relative to the “nominal” host type for which the demand values are found). The actual demand of entry \( E \) on host \( H \) is \( d_E / S_H \) sec. The capacity of the host is the product \( c_H = m_H S_H \), with units of seconds of “nominal” execution per second; thus a single-core nominal host has a capacity of \( 1 \). In general, entries accept calls and also make calls to other tasks, usually at lower layers in a layered system.

A special \( User \) task represents the system’s user population, with a multiplicity \( m_{User} \) equal to the number of users. The \( User \) task has a single entry, which may include a thinking time (a pure delay \( Z_{User} \)) and one or more calls into the system. One User entry execution corresponds to one user response from the system.

Each task is represented as a parallelogram (labeled by the task’s name and thread multiplicity), containing parallelograms for its entries. Each entry is labeled by entry name and host demand or service demand \( d_E \) for one invocation of the entry \( E \). The service demand of an entry is a time duration that it demands from its host to carry out its operation completely. Every task is deployed on a host drawn as a circle. Icons are stacked to represent tasks or processors with multiplicity. A call from one entry to another is represented as an arrow labeled with the mean number of calls or requests \( y_{D,E} \) from entry \( D \) to entry \( E \). A task is a multiserver (the threads are the servers) with a single queue, usually FIFO, to hold all the calls to its entries; thus the calls are targeted to the entries but actually go first to the task queue.
In Figure 2.3, the LQN model has three tasks - *Client*, *WebServer* and *DBServer*, each of which is deployed on its own host - *ClientH*, *WebH* and *DbH*, respectively. Each of the 20 users take 1000 ms think time ($Z_{\text{User}} = 1000 \text{ ms}$) between requests. They are modeled as 20 identical tasks, each running on its own processor *ClientH*. *ClientH* has multiplicity 20 ($m_{\text{ClientH}} = 20$) shown in curly braces. Both *WebServer* and *DBServer* are single threaded tasks, each one having two entries with host service demands indicated in braces (i.e. $d_{\text{webEntry1}} = 2 \text{ ms}$). A single client operation includes one request to *webEntry1* and two to *webEntry2*. Storage devices are not shown but they can be modeled by a task representing the storage logic (read, write operations for example) running on a host representing the device.
**Activities in LQN models**

An entry may include an activity subgraph as shown in Figure 2.4 [45]. The model in this figure contains five tasks: *User, Server, BigLoop, Disk1* and *Disk2*. *Server* and *BigLoop* are deployed on processor *ServerP*. Each disk and *User* task have their own processor which are not shown in this figure. *User* interacts with *Server* for different services. Task *BigLoop* is a pseudo-task for a complex loop pattern. Both *Server* and *BigLoop* interacts with disks for reading and writing operations. Task *Server* contains entry *server* which is defined by activities, with the first activity being *serverStart*. Also the task *BigLoop* contains entry *bigLoop* defined by activities with the first one being *first*.

In the entry *server*, the activity diagram contains *OR-Fork* with probabilities at each path shown as small + in a circle. In the model, there is an *OR-Fork* from *serverStart* to *parInit* and *seqInit* with probabilities of 0.6 and 0.4, respectively. This fork means that either path will be taken with the above mentioned probability.

For parallel operations, there are *AND-Fork* and *AND-Join* shown as small & in a circle. Entries *server* and *bigLoop*, both contain *AND-Fork* and *AND-Join* in their activity diagram. For example, there is an *AND-Fork* from activity *first* to *second* and *third* in entry *bigLoop*. This fork indicates that there is a concurrent parallel flow from activity *first* to *second* and from *first* to *third*. Then the flows are joined by *Fork-Join* and go to the next activity *fourth*. 
In an activity diagram, there can also be Loop operations shown as small * in a circle. In the model of Figure 2.4, there is a Loop from activity seqInit with probability 3.5 to loopOperation which interacts with loop2 and loopEnd. For Loop, flow will go to activities loopOperation and loop2 being repeated 3.5 times on average, and then will go to loopEnd.

In this research, AND-Fork and AND-Join are not considered.

**Calling patterns with cycles in the call graph**

When there is a direct call or a calling path from one entry (say entry1) of a task (task1) to another entry (say entry2) of another task (task2), and a call or calling path from entry2 to entry1, then the calling pattern between the two entries of the two tasks is called a cycle.
Figure 2.5 represents an LQN model with cycle between two entries e03_0 (of task t03) and e05_0 (of task t05).

When a model with cycle is solved by the LQN solver [26], the model does not converge. In fact sometimes the solver stops without any diagnosis [45]. Therefore, in this research, LQN models with cycles are not considered.

Figure 2.5 An LQN model with cycle between two entries of two tasks

**Phases in the LQN model**

By default every entry has only one phase with an execution demand of $d$ and optionally a number of service requests to other task entries. In the case of a synchronous request, phase 1 of the receiving task represents the portion of service when the client is blocked, waiting for the server’s reply. At the end of phase 1, the server replies to the client, which will
unblock and continue its execution. If, however, the entry has more than one phase, the remaining phases will be executed by the server in parallel with the client. Usually background operations such as cache-clearing or logging take place in the second phase. After executing all the phases of the called entry, the server task is ready to accept a new call. In this research only the first phase of execution is considered.

**Performance Measures**

Here we present all the performance measures predicted by LQN models that are used in model simplification.

- $Y_E =$ the number of calls to entry $E$, per user request. $Y_E = \sum_D Y_{DE}$, where the sum is over all the entries, with $Y_{request} = 1$;

- $X_E =$ the service time of one request to entry $E$, including its own execution $d_E$, waiting for its host $H(T(E))$, and waiting for replies to calls it makes to other entries. Each call delay in turn includes waiting for the called task to accept the request, and the entry execution time. Thus $X_E$ is not known before the queueing delays are found; it is this property that makes layered system performance difficult to predict, and drives the use of LQN models;

- $d_H =$ demand of each core in host $H$, per user response = $(1/ m_H) \sum_{H(T(E))} Y_E d_E$, where the sum is over entries of tasks deployed on $H$, $m_H$ is the multiplicity or number of threads of host $H$;

- $d_T =$ demand of each thread of task $T$ per user response = $(1/ m_T) \sum_{T(E)} Y_E X_E$, where the sum is over the entries of task $T$, $m_T$ is the multiplicity or number of threads of task $T$. Some other performance measures predicted by LQN models, that we will use, are
$\lambda_E, \lambda_T =$ throughput of entry $E$ and task $T$ in [invocations/s];

$U_E, U_T, U_H =$ utilization of entry $E$, task $T$, host $H$,

where utilization is defined as:

$U_E = \lambda_E X_E,$

$U_T = \Sigma E U_E,$

$U_H = \Sigma T_{deployed on H} \Sigma E in T \lambda_E d_E / S_H.$

With this definition the utilization of a task or host ranges from zero to its multiplicity value. We also use the saturation level of a task or host, which ranges from zero to one. The saturation level of resource $H$ is defined as:

$SL_H = U_H / m_H.$

Under heavy load the saturation of a bottleneck task or processor approaches one. Under lighter load the most saturated resource can be identified as the bottleneck.

A saturated task or host has all its threads busy almost all the time. The system bottleneck is the model element with the largest utilization, provided it is not a client of an element that also has a large utilization [15] [37].

A method to solve LQN models for single class multiservers were shown in [38]. This process was later improved to support multiclass and phased operations in [16] and [13].

2.2 State of the art in model simplification

There are a number of simplification methods in different performance models which are discussed in the following sub sections.

2.2.1 Aggregation of activities in LQN models

An approach for activity aggregation in LQN models similar to that presented in this thesis
was proposed in [14]. In this approach, if the activities of an entry are separated into two groups, one before the reply and one after, they can be aggregated exactly into two phases by adding up their total CPU demand and calls. The evaluation of the accuracy of activity aggregation was not a focus of the work in [14].

2.2.2 Norton’s theorem

There is a powerful and much-used simplification result in the Norton Theorem for Queues [6] in product-form queuing networks. In this theorem, any subnetwork of queues can be replaced by a single server with a state-dependent service rate. The replacement is exact in the sense that the throughput and delay at the subnetwork interface is the same for the single server [22]. The original result was obtained for a single class of customers, and it was extended to multiple classes in [25].

In principle, Norton’s theorem could be applied to represent a group of tasks and processors in an LQN, and this was used in Rolia’s MOL solver [38]. Its disadvantage is the enormous labor of solving the original model many times for all possible user populations, especially for large systems with large user populations.

2.2.3 Flow equivalent server method

The construction technique used in Norton’s theorem creates a flow-equivalent server (FES), and this technique has been applied as an approximation to systems which are not product-form queueing models [27]. When a submodel is replaced by a FES, the entire model is smaller and easier to solve, and parameter changes outside the submodel can be studied efficiently. The FES construction method isolates the subnetwork and drives it with a fixed number of customers, cycling endlessly; the mean delay of a customer in the
subnetwork is taken as the service time of the FES for that number of customers. This is repeated for every user population that it may experience, which does not scale well to large systems with thousands of customers [22].

2.2.4 Hierarchical decomposition

In hierarchical decomposition, a large model is partitioned into a number of submodels. Each submodel is then evaluated and individual solutions are combined to get the solution of the original model [27]. In this technique, the system is modeled using multiple levels of models. The highest level (level 0) of the models consists of a number of FESs, each of which represents some portion of the system being modelled. The following level contains a number of models that are a more detailed representation of a subsystem represented in the first level as an FES. In general, every level in the hierarchy contains more detailed representation of the submodels from previous level until the final level (level L) where all models are fully detailed and does not have any FESs. The models in hierarchical decomposition should be evaluated from level L to level 0, so that the performance projections for the system being modeled are obtained from its solution.

2.2.5 Surrogate delay methods

Surrogate delay methods (e.g. [27]) replace a subsystem by a delay which is found by solving an auxiliary model. A surrogate delay is somewhat like a FES with a fixed delay rather than a state-dependent rate, but the construction method is different and requires an iterative solution which includes the auxiliary model [27]. Surrogate delays are most useful to address problems of simultaneous resource possession, but they can also be used for model simplification.
2.2.6 Simplification methods in Petri Nets

In [9], the authors propose a methodology of Petri Net model simplification that reduces the size and complexity of the model. They implement a tool called General Purpose Petri Net simulator (GPenSIM) for modeling and simulation of discrete event dynamic systems (DEDS). In this methodology, there are three ways of simplifying Petri nets - level of depth, level of breath and focus of the model. In case of level of depth, where the model has a number of levels each communicating with another above or below, each level can be separated into a submodel. Thus, each submodel becomes a simple model making a hierarchy of models. Level of breath can be applied when a model has many different functionalities. Each functionality can be modeled into a separate module. The focus of the model is a way where the modeler can either keep all the elements in the model (called hard wiring) or remove some elements (e.g., processes that are running in the background) from model and move them into another model or code implementation.

In stochastic Petri nets (SPN), time variables are associated with the transitions [41], i.e. transitions fire after a probabilistic delay determined by a random variable [33]. There exist a number of simplification techniques for SPN. In [30] and [31], the authors propose a Petri Net reduction technique that reduces the size of a marked graph stochastic Petri net in a way that is most suitable for performance calculations (in a marked graph SPN every place has one input and one output so that there cannot be any conflict but there can be concurrency [33]). After the reduction, a number of auxiliary models are generated whose union covers the original SPN model. Finally, auxiliary SPN models are iteratively tuned on some parameters based on a mean delay equivalence relationship till the models are statistically approximately equivalent. In this methodology, the size of the auxiliary models
can become very large for larger Petri Nets models. Also, the method does not guarantee to produce a simplified model since sometimes it may fail to converge.

In [29], the authors presented a complete decomposition technique applied to a class of generalized stochastic Petri nets (GSPNs) called generalized service networks with generalized service centers. This methodology can reduce the state space of auxiliary models to their smallest achievable sizes.

The authors of [2] proposed a time scale decomposition method that can hierarchically reduce GSPNs. A GSPN, can include immediate transitions with zero delay, and priority [3]. The time scale decomposition methodology has good accuracy for a GSPN containing activities whose durations differ by an order of magnitude.

In [7], Ciardo and Trivedi presented a decomposition approach for stochastic reward nets (SRNs), a kind of SPN. The decomposed subnets from the original model can interact with each other through an import graph. The decomposition is based on the concept of near independence.

The authors of [17] propose an aggregation methodology for GSPN that can merge a sequence of transitions into a single timed transition while preserving the stationary token distributions in the embedding environment. Novel formulae for the (locally) marking-dependent rates of the merged transitions are developed that depends on parameters of the aggregated subnet. These formulae exhibit a strong relation to flow equivalence.

2.2.7 Regression methods

When performance models are fitted by regression methods as in [44], a choice must be made for the level of detail in the model. A given structure is fitted, and the goodness of fit is evaluated. Regression thus automatically raises the question of detail, and can answer it
through tests of goodness of fit as discussed in [44] (the reference describes fitting ordinary queuing models but it applies equally to layered models). However this approach cannot be applied to models constructed from a design before a system is built, because it requires operational data for the regression.

2.2.8  Shadow servers
In the Shadow Server method, one service node that violates conditions required for efficient, exact analytic solution in QN model is replaced by two or more servers that enable efficient analytic solution, such that the performance represents the original server [39]. As an example, a CPU server with a priority queue-scheduling discipline can be replaced with a shadow CPU server for each priority class, with jobs of different priorities being routed to different servers. However, this technique does not generate a simpler and smaller model than the original one.

2.2.9  Summary
Overall, there does not appear to be any prior work directly on the problem addressed in this thesis of deriving a simplified layered queuing model directly from a detailed one. The methods for state-based models such as Petri nets do not suggest approaches for LQNs. Surrogate delays to represent a subsystem were considered but need to be congestion-dependent, which would imply using a FES instead. FES was rejected because of the scalability problem in calibrating the server rates for large populations.
3 Chapter: LQN model simplification process

LQN model simplification process begins with an LQN model $M$ (as described in Section 2.1.4) and concludes with another LQN model $M'$ with the same Users, but fewer other tasks and/or processors. The resulting model should give a close approximation to the computed mean response time (and, by Little’s law, also the throughput) seen by the Users (The performance measures are defined in Section 2.1.4).

This research focuses its attention on LQNs with:

- a single User task (there can in principle be more, representing different classes of users),
- no internal parallelism on execution paths (see Section 2.1.4 for more details on activities with parallel execution path),
- calling patterns with no cycles in the call graph (see Section 2.1.4 for more details on cycles in LQN models),
- entries that complete execution before replying (there is no “phase-2 execution” in LQN terms). More on phase-2 execution can be found in Section 2.1.4.

The simplification process starts by merging the smallest LQN model elements and proceed to the larger elements. So, it starts with merging activities, then entries and finally tasks and processors. Two approaches were examined:

- Single-Group Aggregation [22] in which a few tasks are preserved unchanged, and the rest of the model is aggregated into a single task and processor. This preliminary approach is described in this chapter. It is suitable for a limited range of models.
- Dependency-Group Aggregation [21], in which some tasks are preserved and groups of other tasks are identified according to their relationships to the preserved
tasks. The concept is introduced in this chapter, but detailed consideration is deferred to Chapters 4 and 5. This approach is the main contribution of the thesis.

3.1 Model simplification principles

The model aggregation was developed to have certain desirable properties which are termed “simplification principles”. *Single-Group Aggregation* [22] uses three principles:

a) *Capacity limit principle*: to preserve the capacity limit of the model by preserving the bottleneck element(s). Therefore, whenever tasks and processors are merged, the merging operation is applied to all tasks and processors except the bottleneck task and processor.

b) *Total workload principle*: to preserve the total workload, so that the total throughput and host demand of a merged entry or task, per user request, is the same as for the entities that were merged. This ensures the correct performance at the light-load asymptote.

c) *Concurrency principle*: to preserve concurrency available in software and hardware, by which the total multiplicity of a merged task or processor is equal to the sum of multiplicities of the entities that were merged.

If the host processors are homogeneous, the third principle also preserves the total computing capacity of the system.

*Single-Group Aggregation* is a 4-step procedure to preserve the *User* task, and any tasks or processors identified as bottleneck, and aggregates everything else into one task/processor pair with a single entry. Only one bottleneck task is preserved.

*Dependency-Group Aggregation* supports multiple preserved tasks, and pays attention to
the factors that determine the performance of the preserved tasks, expressed in this additional principle:

a) **Dependency principle**: the dependency of a preserved task on each original task $T$ will be preserved in $M'$ as a dependency on the aggregated task that includes $T$.

Activities and entries are aggregated according to the process *Single-Group Aggregation* as presented in this chapter. After that a set of preserved tasks is identified (using criteria described in Chapter 4), and based on the dependency principle, the remaining tasks are partitioned into Dependency Groups. An aggregated task is created for each dependency group as presented in Chapter 4.

**Expectations regarding aggregation**

The goal of LQN model simplification is to reduce the number of tasks and processors in the model while retaining the externally visible performance measures, in this case the mean throughput and response time seen by the users.

The LQN model simplification aggregates two kinds of things: queueing resources (the tasks and processors) and customer classes (the entries of the tasks that are aggregated).

For queueing resources there is a well-established expectation that overall queueing delays are reduced when resources are aggregated together, which is well-known, for example, if a set of identical servers with separate queues are merged into a multiserver [21]. However, there are no comparable exact results available for layered servers in a LQN.

For class aggregation the expected tendency is in the opposite direction. An LQN model includes many customer classes, for example, each different operation (entry) of a subsystem (task) is a different class of service. In order to simplify the model, it is necessary
to aggregate some of these classes, which introduces errors. For certain multi-class queueing network models, Dowdy et al. showed in [10] that a single-class aggregated model always has lower overall throughput (and thus longer delay). Although their result does not apply directly to the present models, this result suggests that class aggregation at a server may give worse performance. Thus class and resource aggregation may be expected to produce opposite effects in the performance measures of the simplified LQN.

3.2 Single-Group Aggregation

*Single-Group Aggregation* is a 4-step procedure to preserve the *User*, and any tasks or processors identified as bottleneck, and aggregates everything else into one task/processor pair with a single entry. Only one bottleneck task is preserved. It first merges activities for an entry (if they are defined in detail) into a single entry operation, and the entries of each task (if it has more than one) into a single entry, as described in Section 3.1. This produces a model with single-entry tasks with their original deployment to processors. One task is identified as a bottleneck, as described later.

Where multiple tasks are deployed on a common processor, they are merged into one task (except that a task which is identified as a bottleneck is not merged). The call rates from the one merged entry of the merged task are calculated based on the relative throughputs of the merged entries, as weights. The merged task’s multiplicity is the summation of multiplicities of all the tasks that are being merged. The formulas for aggregating tasks in Single-Group Aggregation are presented in Section 3.2.1. This produces a model with one task per processor, except possibly for the bottleneck task and its processor.

At this point all the tasks except the bottleneck task are collected into a single group, and
the tasks in this group (with their processors) are replaced by a single processor with a single task whose multiplicity is the sum of the multiplicities of the processors. It is done by aggregating two tasks at a time repeatedly until all the tasks are merged into one task. Application of this aggregation on example models can be found in the following sections. The aggregation operations described above have some pre-conditions and post-conditions as described in Table 3.1.

Table 3.1 Pre conditions and post conditions of simplification process

<table>
<thead>
<tr>
<th>Simplification operations</th>
<th>Pre-conditions</th>
<th>Post-conditions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Operation 1: Aggregation of activities</td>
<td>One or more entries have activities</td>
<td>Activities of each entry are replaced with a new service demand of entry</td>
</tr>
<tr>
<td>Operation 2: Aggregation of entries</td>
<td>One or more tasks have more than one entry</td>
<td>Entries at each task are merged into single entry</td>
</tr>
<tr>
<td>Operation 3: Aggregation of tasks deployed on a common processor</td>
<td>Every task has a single entry</td>
<td>All tasks that are deployed on a common processor are merged into a single task.</td>
</tr>
<tr>
<td>Operation 4: Aggregation of Task/Processor Pairs in the Single Group</td>
<td>Every processor has a single task assigned</td>
<td>All the processors and all the tasks are merged into a single processor and task respectively.</td>
</tr>
</tbody>
</table>

3.2.1 Single-Group Aggregation: Example 1

The details will be illustrated by a running example (Example 1) defined by the LQN model from [45], shown in Figure 3.1 (a). Each of a number of users ($N = 20$) makes one visit
to the Server task, which has one entry server with a number of activities. Some requests from different activities are delegated to the pseudo task BigLoop and some requests are made to the task Disk for diskread and diskwrite operations. Server and BigLoop (which represents a part of the logic of Server) are deployed on the same processor ServerP which has a processor-shared queuing discipline. Processor-sharing models the usual time-slicing behavior of computer operating system schedulers. Task Disk is deployed on DiskP with FIFO queuing discipline, which models the fact that disk operations must be served to completion. From the initial experiments, Disk and its processor DiskP are found to be the bottleneck in this model. Thus, Disk and DiskP are to be preserved in the simplification process.

The simplification operations are described as follows. Some calculations can take advantage of finding a solution of the initial model being simplified (as shown in Figure 3.1), and this is assumed to be available.

**Operation 1: Aggregation of activities**

In each task $T$, for each entry $E$ that has activities in its definition, the activities are aggregated. For activity $I$, let:

\[
\begin{align*}
    d_E, d_I &= \text{execution demand of entry } E \text{ (to be found), and activity } I, \text{ (given),} \\
    \lambda_E, \lambda_I &= \text{throughput of entry } E \text{ and activity } I, \text{ in any solution of the model,} \\
    w_I &= \text{executions of activity } I \text{ per request to entry } E \text{ (this may be calculated by examining} \\
    &\text{the activity graph, or from a model solution as } w_I = \frac{\lambda_I}{\lambda_E}, \\
    y_{I,B} &= \text{mean calls from activity } I \text{ to another entry } B \text{ of another task,} \\
    y_{E,B} &= \text{aggregated mean calls from entry } E \text{ to entry } B \text{ (to be found).}
\end{align*}
\]
Figure 3.1 Example 1 and the result of aggregating activities

(a) Original model from [45]
(b) After aggregating the activities
Then the aggregated execution demand of entry $E$ is

$$d_E = \sum I w_I d_I$$  \hspace{1cm} (3.2.1.1)

and the aggregated number of calls from entry $E$ to another entry $B$ is

$$y_{E,B} = \sum I w_I y_{I,B}$$  \hspace{1cm} (3.2.1.2)

where the sum in both cases is over the activities of entry $E$.

In Example 1, the entry server executes activities for which the values of \((activity\ name, weight, execution\ demand)\) are as follows:

\begin{align*}
(serverStart, 1, 1.e^{-6}), & \quad (parinit, 0.6, 0.1), & \quad (parA, 0.6, 0.05), \\
(parReply, 0.6, 0.01), & \quad (seqinit, 0.4, 0.3), & \quad (loopOperation, 1.4, 0.1), \\
(loop2, 1.4, 1.e^{-6}), & \quad (loopEnd, 0.4, 1.e^{-6}), & \quad (bigLoopDriver, 0.48, 1.e^{-6}), \\
(seqReply, 0.4, 0.005).
\end{align*}

Applying Eq (3.2.1.1) for the CPU demand we obtain for the merged activities $d_{server} = 0.358$. Applying Eq (3.2.1.2) for the calls, first for the call from the activity $bigLoopDriver$ to the entry $bigLoop$, the aggregated call has $y_{server,bigLoop} = 0.48$. Second, for the calls to entry $diskread$, the aggregated call has $y_{server,diskread} = 3.8$. Figure 3.1 (b) represents the model after aggregating all the activities from Figure 3.1 (a).

**Operation 2: Aggregation of entries**

The second operation merges the entries of each task having more than one entry. For a task $T$ let:

$d_M, d_K = $ execution demand of the merged entry $M$ (to be found), and of an original entry $K$ of task $T$.

$y_{K,B}$, $y_{M,B}$ = mean number of calls from entry $K$ of task $T$ to any entry $B$ of another task, and from the merged entry $M$ of task $T$ to entry $B$, respectively.
\( w_K = \) weight of original entry \( K = \) fraction of all calls made to task \( T \), that go to entry \( K \).

From any solution, \( w_K \) can be found as \( \lambda_K / \sum K \lambda_K \), where the sum is over the entries to be merged. Then the service demand of the merged entry \( M \) is:

\[
d_M = \sum_K w_K d_K
\]

and the calls from entry \( M \) to entry \( B \) of some other task are:

\[
y_{M,B} = \sum_K w_K y_{K,B}
\]

where the sums are over the entries to be merged in both equations.

In Example 1, as shown in Figure 3.1 (b), only task \( \text{Disk} \) has more than one entry. The values of \((\text{entry name}, \text{weight}, \text{execution demand})\) for the entries are \((\text{diskread}, 0.797, 1)\) and \((\text{diskwrite}, 0.203, 0.04)\). Applying Eq (3.2.1.3), \( d_{\text{eDisk}} = 1 * 0.797 + 0.04 * 0.203 = 0.805 \). There are no outgoing calls from \( \text{Disk} \). The incoming calls are simply transferred to the merged entry (if this results in more than one call from a specific entry, the calls are merged and the numbers summed). Figure 3.2 (a) represents the model after merging entries.

**Operation 3: Aggregation of tasks deployed on a common processor**

In this operation, we consider merging two tasks that share a host. Each task has a single entry (entries have been previously merged if necessary). There are two cases: vertical merging and horizontal merging.

**Vertical merging:** If one task calls the other, we call it vertical merging. Let,

\( d_A, d_B, d_M = \) the service demands of the entries \( A \) and \( B \) of the two tasks to be merged, and the entry \( M \) of the merged task, respectively.

\( y_{A,B} = \) the number of calls from entry \( A \) to entry \( B \).
\( y_{A,C}, y_{B,C}, y_{M,C} \) = the number of calls from entry \( A \) to entry \( C \) and from entry \( B \) to entry \( C \), respectively, where entry \( C \) is an entry of another task which is called by one or both of the

![Diagram](image)

(a) After merging entries  
(b) After merging tasks Server and BigLoop

Figure 3.2 Aggregating entries and tasks in Example 1

tasks to be merged, and from the merged entry \( M \) to entry \( C \).

Then the service demand and the number of calls to entry \( C \), for the entry of the merged task, are:
\[ d_M = d_A + y_{A,B} \, d_B \]  \hfill (3.2.1.5)

\[ y_{M,C} = y_{A,C} + y_{A,B} \, y_{B,C} \]  \hfill (3.2.1.6)

The call parameters on incoming calls in vertically merged tasks are calculated as for merged entries.

In Example 1 as shown in Figure 3.2 (a), the tasks Server and BigLoop both are deployed on the same processor ServerP and Server calls BigLoop, so they require vertical merging. They are merged in Figure 3.2 (b) as MergedTask1 with an entry mergedEntry1 with service demand of \(0.3580003 + 0.479998 \times 0.140001 = 0.425204\) (following Eq (3.2.1.5)). The number of outgoing calls from MergedTask1 to eDisk is \(3.7599926 + 0.479998 \times 2\) = 4.71999 (following Eq (3.2.1.6)).

**Horizontal merging** occurs when there is no calling relationship between the tasks. Just as for merging two entries of the same task, the service demand and the calls of the merged task are computed by Eq (3.2.1.3) and (3.2.1.4), where the entry \(K\) designates the single entry of any one of the tasks to be merged, and the sums are over the entries of these tasks.

As in merging entries, the calls into the separate entries are transferred to the merged entry \(M\) and if this results in multiple calls between a pair of entries, the calls are merged and the call parameters are summed. Example 1 does not show a case of horizontal merging, so this calculation is not applied.

**Operation 4: Aggregation of Task/Processor Pairs in the Single Group**

Task aggregation continues from this point to aggregate all task/processor pairs but the User and the bottleneck. Since Example 1 does not have enough tasks to illustrate this operation, it will be discussed in connection with the second example in Section 3.2.2.
The performance of the simplified models for Example 1 after each operation is shown in Table 3.2 and Table 3.3. In Example 1, tasks Server and BigLoop are infinite servers (i.e., they have no thread limit), whereas Disk and the processors ServerP and DiskP are single servers (the number of threads are shown in curly braces in Table 3.2). On the first row of Table 3.2, the utilization of each resource (tasks and processors) of the original model is shown. In the subsequent rows, the utilization of each resource is reported after each of the activity, entry and task simplification steps. After the simplification procedure, the utilizations of tasks and processors remain almost same.

Table 3.2 Resource utilizations after each simplification operation of Example 1

<table>
<thead>
<tr>
<th>Operations</th>
<th>Example1</th>
<th>$U_{\text{Server}}$</th>
<th>$U_{\text{BigLoop}}$</th>
<th>$U_{\text{Disk}}$</th>
<th>$U_{\text{ServerP}}$</th>
<th>$U_{\text{DiskP}}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Original model</td>
<td></td>
<td>6.67</td>
<td>1.30</td>
<td>0.993</td>
<td>0.111</td>
<td>0.993</td>
</tr>
<tr>
<td>Operation1</td>
<td>Activity simplification</td>
<td>6.60</td>
<td>1.28</td>
<td>0.991</td>
<td>0.111</td>
<td>0.991</td>
</tr>
<tr>
<td>Operation2</td>
<td>Entry simplification</td>
<td>6.49</td>
<td>1.41</td>
<td>1.000</td>
<td>0.112</td>
<td>1.000</td>
</tr>
<tr>
<td>Operation3</td>
<td>Task Simplification</td>
<td>6.81</td>
<td>0.976</td>
<td>0.109</td>
<td>0.976</td>
<td></td>
</tr>
</tbody>
</table>

Table 3.3 System throughput and response time after each simplification operation of Example 1

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Original model</td>
<td></td>
<td>0.261</td>
<td>26.5</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>Operation1</td>
<td>Activity simplification</td>
<td>0.261</td>
<td>26.6</td>
<td>0.144</td>
<td>0.415</td>
</tr>
<tr>
<td>Operation2</td>
<td>Entry simplification</td>
<td>0.263</td>
<td>26.0</td>
<td>0.723</td>
<td>2.073</td>
</tr>
<tr>
<td>Operation3</td>
<td>Task Simplification</td>
<td>0.257</td>
<td>27.8</td>
<td>1.657</td>
<td>4.864</td>
</tr>
</tbody>
</table>

Table 3.3 represents the system throughput and system response time of the original model in the first row and then after activity, entry and task simplifications respectively in the
consecutive rows. From the two rightmost columns of Table 3.3, it is observed that the amount of relative errors incurred by each simplification is relatively low comparing to the gain in the size of the models. Here, relative error (RE) in throughput/response time is computed with the following equation:

$$RE \, (\%) = \frac{\text{Metric of original model} - \text{Metric of simplified model}}{\text{Metric of original model}} \times 100$$  \hspace{1cm} (3.2.1.7)

where, Metric can be either System throughput or response time.

In this example, throughput error due to activity and entry simplifications are less than 1%, and due to task simplification less than 2%. The errors incurred by activity, entry and task simplifications on system response time is less than 1%, about 2% and almost 5% respectively.

3.2.2 Single-Group Aggregation and Task/Processor Aggregation: Example 2

Figure 3.3 (a) represents another example of an LQN model called eShop, where a number of users’ requests go through StoreApp, CustomerDB, InventoryDB and FileServer for read and write operations. This model has no activity detail and just one entry per task so it is ready for task level simplification. Preliminary experiments show that the bottleneck is the task StoreShopping.

**Operation 3: Merge Tasks on a Common Processor**

In this model, tasks CustomerDB and InventoryDB are merged since they are deployed on the same processor. The values of \((\text{entry name, weight, execution demand})\) are \((cRW, 0.439, 3)\) and \((iRW, 0.561, 2)\). Thus, applying Eq (3.2.1.3) the service demand of mergedEntry1 is obtained as \(d_{\text{mergedEntry1}} = 3 \times 0.439 + 2 \times 0.561 = 2.439\) (where the throughputs of \(cRW\) and \(iRW\) are 0.03001 and 0.03833, respectively). The number of incoming calls to the merged entry is 9 since the incoming calls from storeShopping should be directly summed.
up. For the number of outgoing calls, the values of \((\text{entry name, weight, number of calls from merged entry of task to fRW})\) are \((cRW, 0.439, 3)\) and \((iRW, 0.561, 3)\). Thus, applying Eq (3.2.1.4), the number of calls from the merged entry to \(fRW\) is \(3 \times 0.439 + 3 \times 0.561 = 3\). Figure 3.3 (b) represents the model after merging \(CustomerDB\) and \(InventoryDB\) tasks.

![Original model of eShop](image1.png)

![After merging](image2.png)

(a) Original model of eShop
(b) After merging \(CustomerDB\) and \(InventoryDB\)

**Figure 3.3 An LQN model of eShop**

**Operation 4: Aggregation of Task/Processor Pairs in the Single Group**

The next step of aggregation for this example will be merging the single group that contains tasks that are deployed on separate processors. In preserving the bottleneck task
StoreShopping, the single group contains FileServer and MergedTask1.

In case of horizontal as well as vertical merging of such tasks, the service demands, incoming and outgoing calls and multiplicities of tasks are calculated as for horizontal and vertical merging of tasks on the same processor, as discussed in Operation 3 in Section 3.2.1. The merged processor’s multiplicity is the summation of multiplicities of merging processors. In Figure 3.4, FileServer and MergedTask1 (originally deployed on different processors) are merged.

![Diagram](image.png)

**Figure 3.4 Aggregation of processors and all their tasks on eShop**

Table 3.4 and Table 3.5 represent the performance results of the simplification process for Example 2. From Table 3.4, it is observed that merging tasks CustomerDB and
InventoryDB incur only about 1% error in system throughput and system response time. Then, merging processors mergedProcessor1 and pFileServers along with their all their tasks MergedTask1 and FileServer incur less than 6% and 4% errors in system throughput and system response time respectively.

Table 3.4 Effects of the simplification operations on system response time and throughput for Example2

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Original model</td>
<td></td>
<td>0.00244</td>
<td>12236.2</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>Operation3</td>
<td></td>
<td>0.00246</td>
<td>12360.6</td>
<td>0.820</td>
<td>1.017</td>
</tr>
<tr>
<td>Operation4</td>
<td></td>
<td>0.00258</td>
<td>11816.4</td>
<td>5.738</td>
<td>3.431</td>
</tr>
</tbody>
</table>

The task and processor resources are all single servers in the original model but the merged resources become multiservers, with the multiplicities shown in curly brackets after the utilization in Table 3.5. The saturation level of the merged resource is then (utilization)/(multiplicity). Because of averaging, the saturation level for a merged resource is lower than for the highest utilized resource before merging.

Table 3.5 Effects of simplification operations on utilizations of resources of Example2

<table>
<thead>
<tr>
<th>Operations</th>
<th>Example 2</th>
<th>U_{StoreShopping}</th>
<th>U_{CustomerDB}</th>
<th>U_{InventoryDB}</th>
<th>U_{FileServer}</th>
<th>U_{pWebServers}</th>
<th>U_{pDBServers}</th>
<th>U_{pFileServers}</th>
</tr>
</thead>
<tbody>
<tr>
<td>Original model</td>
<td></td>
<td>0.999</td>
<td>0.457</td>
<td>0.520</td>
<td>0.812</td>
<td>0.022</td>
<td>0.167</td>
<td>0.812</td>
</tr>
<tr>
<td>Operation3</td>
<td></td>
<td>1.000</td>
<td>0.977{2}</td>
<td>0.811</td>
<td>0.023</td>
<td>0.164</td>
<td>0.811</td>
<td></td>
</tr>
<tr>
<td>Operation4</td>
<td></td>
<td>0.999</td>
<td>0.977{3}</td>
<td>0.023</td>
<td>0.984{2}</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
It is worth noting in Table 3.5, which shows the utilizations of tasks and processors after the simplification operations, that the system bottleneck (i.e., the StoreShopping task) remains the same throughout the simplification process.

**Merging processors only**

If we investigate further on Example 2 and merge only the processors pWebServers and mergeProcessor2, we get results shown in Table 3.6 and Table 3.7.

### Table 3.6 Effects of aggregating only non-bottleneck processors in Example2

<table>
<thead>
<tr>
<th>Example2</th>
<th>Sys. Thruput (jobs/ms)</th>
<th>Sys. Response time (ms)</th>
<th>Relative error in Sys.Thruput(%)</th>
<th>Relative error in Sys. Response time(%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Original model</td>
<td>0.00244</td>
<td>12236.2</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>Aggregating only processors</td>
<td>0.003</td>
<td>10139.9</td>
<td>17.213</td>
<td>17.132</td>
</tr>
</tbody>
</table>

### Table 3.7 Utilization of resources after aggregating only non-bottleneck processors in Example2

<table>
<thead>
<tr>
<th>Example2</th>
<th>U_{StoreShopping}</th>
<th>U_{CustomerDB}</th>
<th>U_{Inventories}</th>
<th>U_{FileServer}</th>
<th>U_{pWebServers}</th>
<th>U_{pDBServers}</th>
<th>U_{pFileServers}</th>
</tr>
</thead>
<tbody>
<tr>
<td>Original model</td>
<td>0.999</td>
<td>0.457</td>
<td>0.520</td>
<td>0.812</td>
<td>0.022</td>
<td>0.167</td>
<td>0.812</td>
</tr>
<tr>
<td>Aggregate processors only</td>
<td>0.999</td>
<td>0.977{3}</td>
<td>0.992{3}</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

From Table 3.5, we know that utilization of pWebServers is 2.3% and mergeProcessor2 is 49.2% utilized. Merging processor in this case incurred much higher errors (about 17% each, see Table 3.6). Since database processor utilization is only 0.16, compared to 0.81 for the file server processor (Table 3.5), when the total capacity is shared, the contention is significantly lower for the fileserver accesses, and this effect is even stronger after the very lightly loaded webserver processor is merged (note that the merged processor utilization of 0.992 is relative to a capacity of 3, so it is only 33% saturated, see Table 3.7).
This effect would be much less pronounced if the original database processor utilization were lower. At 81% saturation, it is almost a bottleneck itself. So, it appears that merging near bottleneck resources (tasks and processors) degrades accuracy.

### 3.3 Bottleneck preservation

One of the heuristic simplification principles is to preserve the bottleneck task and processor. A question may arise about the necessity of this principle. In this section, we examine this question.

In Section 3.2.1, three simplification operations are done on Example 1 and the final simplified model is shown in Figure 3.2 (b) where task *Disk* and processor *DiskP* are highly saturated (utilization 97.6%, see Table 3.2). We further investigate the performance results after merging the non-bottleneck task *MergedTask1* with task *Disk* and merging the non-bottleneck processor *mergedProcessor1* with processor *DiskP*. Figure 3.5 presents the resultant simplified model.

![Figure 3.5 Aggregating bottleneck with a non-bottleneck from Example 1](image-url)
In this model (Figure 3.5), the utilization of MergedTask2 is 2.436 and mergedProcessor2 is 1.44, which means the bottleneck is considerably relaxed (less saturated) after merging a bottleneck model element with a non-bottleneck. This is why the relative errors in System throughput and System response time are quite large at 30.452% and 67.561% respectively. There is a tendency that the queuing delay will decrease when two resources are merged [24]. Since adding the number of threads of merging model elements gives additional capacity for both task and processor, the bottleneck gets relaxed when we merge a bottleneck model element with a non-bottleneck one.

### 3.4 Single-Group Aggregation algorithm

A high-level algorithm of Single-Group Aggregation is given below.

#### Listing 3.1 Summarized algorithm for Single-Group Aggregation

```plaintext
SingleGroupAggregation(Model originalModel): Model

/* Make a copy of original model to keep it unchanged. All modification of this algorithm are made on the SimplifiedModel */
Model SimplifiedModel := OriginalModel;
Solve OriginalModel;

/* Find bottleneck of the model based on the level of saturation */
Entity Bottleneck = FindBottleneck(OriginalModel);

/* Merge activities and entries */
For each task T in SimplifiedModel
   If (T has activities)
      Merge activities according to Eq (3.2.1.1) and Eq (3.2.1.2);
      Transfer incoming calls to the merged entity;
      If (this results in more than one call from a specific entry)
         Merge the calls and sum the mean numbers of calls;
   End-If
End-If

If (T has multiple entries)
   Merge entries according to Eq (3.2.1.3) and Eq (3.2.1.4);
   Transfer incoming calls to the merged entry;
   If (this results in more than one call from a specific entry)
```

44
Merge the calls and sum the mean numbers of calls;
End-If
End-If
End-For
Update SimplifiedModel with these aggregated activities and entries;

/* Merge tasks deployed on a processor */
For each processor P which is not a bottleneck
If (there are multiple tasks deployed to P)
While (true)
If(all tasks deployed to P are merged into one except User task and Bottleneck)
Break;
End-if
For every two tasks (T1, T2) deployed to P
If ((T1, T2) are vertical and none is Bottleneck)
   Do vertical merging according to Eq (3.2.1.5) and Eq (3.2.1.6);
   Do task aggregation by summing up all the threads;
   Calculate incoming calls as for merged entries;
Else if ((T1, T2) are horizontal and none is Bottleneck)
   Do horizontal merging according to Eq (3.2.1.3) and Eq (3.2.1.4);
   Do task aggregation by summing up all the threads;
   Calculate incoming calls as for merged entries;
End-If
End-For
End-While
End-If
End-For
Update SimplifiedModel with these aggregated tasks;

/* Merge task/processor pair in single group */
While (true)
If (all tasks are merged except User task and Bottleneck)
Break;
End-If
For every two tasks (T1, T2) in SimplifiedModel
If ((T1, T2) are vertical and none is Bottleneck)
   Do vertical merging according to Eq (3.2.1.5) and Eq (3.2.1.6);
   Do host aggregation by summing up all the threads of the hosts;
   Do task aggregation by summing up all the threads of the tasks;
Calculate incoming calls as for merged entries;
Else if ((T1,T2) are horizontal and none is Bottleneck)
    Do horizontal merging according to Eq (3.2.1.3) and Eq (3.2.1.4);
    Do host aggregation by summing up all the threads of the hosts;
    Do task aggregation by summing up all the threads of the tasks;
    Calculate incoming calls as for merged entries;
End-If
End-For
End-While
Update SimplifiedModel with these aggregated processors and tasks;
Return SimplifiedModel;

3.5 Limitations of Single-Group Aggregation

In some cases Single-Group Aggregation gives a reduced model with just two or three elements and little loss of accuracy [22]. But, it was also observed that there are many other cases where this aggregation does not work as effectively as required (more on this is discussed in Chapter 4). Some limitations of Single-Group Aggregation are:

- Other than the bottleneck, there could be other resources which are quite highly saturated, that are aggregated with other resources in the simplified model.
- Single-Group Aggregation process is not fully-automated but rather semi-automated, where simplification is done incrementally by adding one task at a time.
- In every step of Single-Group Aggregation, the aggregated model is solved by the solver and some parameters (throughput) from the generated solution file is used for the next step of aggregation. This makes the process to read data from the intermediate solution files which is time-consuming.

Chapter 4 presents Dependency-Group Aggregation that supersedes the Single-Group Aggregation and eliminates the limitations presented above.
4 Chapter: Dependency-Group Aggregation

This chapter describes a more flexible and robust process for simplifying LQN models compared to Single-Group Aggregation presented in the last chapter. The flexibility comes from the preservation of the second or more bottleneck resources, based on their saturation. But this may partition the model in such a way that a single aggregated task will introduce calling cycles and makes a solution impossible. Therefore, there is a need to aggregate only the tasks (along with their processors) on which a preserved task is dependent. So for each preserved task, a group of tasks is identified and are aggregated in the simplified model.

In Dependency-Group aggregation the first two steps of the Single-Group procedure will be retained and will be called Stage 1, producing a Stage 1 aggregated model, and tasks which are not to be aggregated will be called preserved tasks. Stage 1 aggregates all activities and entries of a task $T_i$ into a single entry as described in Chapter 3. Since each task now has only one entry, we can without confusion label the entry parameters $d_E$ and calls $(E_1, E_2)$ by the task names, as demand $d_T$ and calls $(T_1, T_2)$. The aggregated host-demand and call values at Stage 1 are exact; the queueing times in this Stage 1 model have some degree of approximation error due to merging of classes, which was however found in [22] to often be small.

4.1 Groups of tasks for aggregation

The tasks that will not be aggregated (the preserved tasks) may include:

1) Bottleneck tasks: It is shown in Chapter 3 that aggregating a bottleneck task together with others sometimes gives poor accuracy, and this seems to be generally true. Also, the bottleneck is important in defining the saturation properties of a system, so
preserving it should preserve those properties.

2) Other highly saturated tasks: If we intend to improve the system by mitigating the bottleneck then another task may emerge as a candidate “second bottleneck” and it would then (for the same reasons) be desirable not to have merged that task [22]. In general, the candidates for bottlenecks are the highest tasks in an ordering based on task saturation level.

3) Tasks subject to change: We may also choose to keep a task out of aggregation if we want to study the effect of major changes in that task.

4) Tasks with key measures: We would like to observe the performance measures gathered for that task. In particular, the measures for the User task define the user-related performance measures of the system, and it will always be a preserved task.

5) Tasks deployed on bottleneck processors or highly saturated processor: Tasks deployed on bottleneck processors also need to be preserved since merging the task (deployed on bottleneck processor) with other non-bottleneck tasks (deployed on other processor) requires merging the bottleneck processor with other non-bottleneck processors. For the same reason as mentioned in item (1) above for tasks, bottleneck processors are preserved in this aggregation. In [22], it has been shown that merging a highly utilized/saturated processor can degrade the accuracy. So, we may preserve a second bottleneck processor as well as the task(s) deployed on it. If there is more than one task deployed on a bottleneck processor and none of them is a bottleneck task, those tasks can be merged into one task and preserved.

Before defining groups for aggregation, we define a set \( TP = \{TP_1, TP_2 \ldots\} \) of tasks to be preserved. We wish the performance measures of these tasks to be well approximated in
the aggregated model. Therefore, we define the groups to preserve the dependency of the performance of the preserved tasks, on the tasks that are grouped, according to the Dependency Principle in Chapter 3.

4.2 Tasks grouped by their dependencies

A task \( T \) may affect the performance of a preserved task \( TP \) through delay dependency or by processor contention dependency. Delay dependence arises if \( TP \) makes a blocking call to \( T \) directly, or calls intermediary tasks with delay dependency on \( T \) (that is, if there is a path of blocking calls from \( TP \) to \( T \)). Processor contention dependency arises if \( T \) shares a processor with \( TP \). In this chapter we focus on the effect of delay dependency by restricting the original system to provide a separate host, (possibly a virtual machine) for each task, to eliminate processor contention dependency for now. Chapter 5 presents an extension of this aggregation called Dependency-Group Aggregation with Shared Deployment that handles processor contention dependency, used for the simplification of LQN models that have more than one task deployed on a single processor.

A blocking dependency of task \( T_1 \) on task \( T_2 \) in an LQN model is created by existence of a call path from \( T_1 \) to \( T_2 \) (that is, there is a direct call from \( T_1 \) to \( T_2 \), or indirect calls via one or more other tasks). We will define this dependency relative to a set of preserved tasks, as follows.

**Definition:** Task \( T_1 \) is Preserved-Task Dependent (PT-Dependent) on task \( T_2 \) if there is a call path from \( T_1 \) to \( T_2 \) that does not pass through a preserved task.

We will express this PT-dependency as \( T_1 \prec T_2 \). We assume that the system (and the layered model) does not have cyclical dependencies. If task \( T_1 \) has no dependency relation
with $T_2$ we can write $T_i || T_2$.

For each task $T_i$ its PT-dependency set $P_i$ is then defined as

$$P_i = \{P | P < T_i\}$$

$P_i$ is non-empty for every non-preserved task $T_i$ since every task except the User may be called in executing the application, therefore it always contains at least the User task.

![LQN model showing tasks with PT-dependency set \{P1, P2\}]()

Figure 4.1 shows a model with three preserved tasks in the set $TP = \{U, TP_1, TP_2\}$. The three shaded tasks all have the same PT-dependency set $P = \{TP_1, TP_2\}$. The PT-dependency sets $P$, one for each non-preserved task, partition the non-preserved tasks into subsets $G(P)$ which will be a basis for the groups for aggregation. All the tasks in $G(P)$ have the same PT-dependency set, and all other tasks have different PT-dependency sets. The tasks in a particular $G(P)$ have an impact (through blocking calls) on just those preserved tasks in subset $P$, and no others, and in $M'$ all those blocking delay effects are captured approximately by blocking calls to a single task $TA(G(P))$ created by aggregating
the tasks in $G(P)$.

The groups and the aggregated tasks are illustrated in Figure 4.2 by an example “case-A” with three preserved tasks $U$, $TP_1$ and $TP_2$ (shown as parallelograms with bold borders). The tasks have four different dependency sets: \{U\}, shown as tasks with no shading, \{TP_1\} as tasks with diagonal stripes, \{TP_2\} as tasks with grey shading, and \{U, TP_2\} as the one task with diamond shading. Tasks with the same shading form a group.

To simplify the model, each group is aggregated into a single task running on its own processor, following the method in Section 4.4. This gives the simplified model in Figure 4.3.

4.3 Desirable properties of the aggregation groups

Dependency-Group Aggregation is motivated by a desire to preserve the essential components of the system, and to make the results traceable back to these components, while merging the elements that contribute less to the performance result. Single-Group Aggregation as in Chapter 3 preserves for each group

- its total workload
- the total processing capacity available to it
- the maximum concurrency of execution threads and cores

What is new here in Dependency-Group Aggregation is:

- there can be an arbitrary number of preserved tasks, giving the modeller flexibility
- the dependencies of the preserved tasks are also preserved, in the following sense. For every preserved task $P$ and unpreserved task $T$:
  - If $P$ depends on $T$, then $P$ depends in $M'$ on a merged task derived from
a group containing $T$

- If $T$ depends on $P$, then a merged task derived from a group containing $T$ depends on $P$.

Figure 4.2 “case-A” showing groups defined by four distinct PT-Dependency sets $P = \{U\}, \{TP_1\}, \{TP_2\}, \{U, TP_2\}$

Figure 4.3 Aggregated model for “case-A” using the dependency groups shown in Fig. 4.2

Thus, if we can think of each task as contributing a shadow of itself to its merged task, then there is a shadow of every non-persistent task and the dependencies between the persistent
4.4 Aggregation of a group of tasks

We consider a group or subset $G$ of tasks $T$, each of which has one entry (as produced by Stage 1 aggregation) and its own processor (Chapter 5 discusses simplification of tasks sharing a common processor). $T$ has demand $d_T$ and makes an average of $y_{T,T_i}$ calls to each other task $T_i$, each time it is invoked.

An aggregated task $TA(G)$ is substituted for $G$, with CPU demand $DA(G)$, and $y_{T_i,TA(G)}$ calls coming to $TA(G)$ from each task $T_i$ not in $G$. The calculation of $DA$ and $y$ begins by finding $Y_i$ for each task $T_i$ in $M$:

$Y_i = \text{mean invocations of } T_i \text{ per user response, which will be called the “total calls” to } T_i.$

Total calls $Y_i$ for each task $T_i$ is found by setting $Y_{User} = 1$ (for one user response) and solving these equations for all tasks $T_i$ in $M$:

$Y_i = \sum_{j} (Y_j \ast y_{T_j,T_i}) \text{ for all tasks } T_j \text{ calls } T_i$  \hspace{1cm} (4.4.1)

From this the invocations of $G$ per user response (or “total calls” to $G$) is $YA(G)$:

$YA(G) = \sum_{T_j \in G} \sum_{T_i \in G} Y_i \ast y_{T_i,T_j}$ \hspace{1cm} (4.4.2)

Then the demand $DA(G)$ is the total demand of $G$ per user response, divided by the number of calls to $G$:

$DA(G) = \sum_{T_j \in G} (Y_j \ast d_{T_j}) / YA(G)$ \hspace{1cm} (4.4.3)

The number of calls $y_{TA(G),T_k}$ from $TA(G)$ to target task $T_k \notin G$, per call into group $G$, is defined by a weighted average of calls from tasks $T_j$ in $G$, weighted by $Y_j$:

$y_{TA(G),T_k} = \text{(total calls from } G \text{ to } T_k) / \text{total calls to } G$

$= \sum_{T_j \in G} (Y_j \ast y_{T_j,T_k}) / YA(G)$ \hspace{1cm} (4.4.4)
If the target tasks are also members of other groups, the number of calls between groups is calculated directly as follows. The number of calls from $TA(G1)$ to a task representing another group, say $TA(G2)$, is $y_{TA(G1),TA(G2)}$, which is the sum of the calls from $TA(G1)$ over tasks $T_k$ in $G2$, thus:

$$y_{TA(G1),TA(G2)} = \sum_{T_k \in G2} \sum_{T_j \in G1} [(Y_j * y_{T_j,T_k}) / YA(G1)]$$  \hspace{1cm} (4.4.5)$$

An aggregated host $HA(TA(G))$ is created for the aggregated task $TA(G)$, with processing capacity $c$ and multiplicity $m$, equal to the total capacity and multiplicity. In $M'$ the host $HA(TA(G))$ has the properties:

$$c_{HA(TA(G))} = \sum_{T_j \in G} c(H(T_j))$$ \hspace{1cm} (4.4.6)$$

$$m_{HA(TA(G))} = \sum_{T \in G} m_{H(T)}$$ \hspace{1cm} (4.4.7)$$

$$S_{HA(TA(G))} = c_{HA(TA(G))/m_{HA(TA(G)}}$$ \hspace{1cm} (4.4.8)$$

Figure 4.3 represents an aggregated model of the original model “case-A” in Figure 4.2. The preserved tasks from the original model are shown with bold borders. The aggregated model has 7 tasks and 7 processors whereas the original model in Figure 4.2 has 21 tasks and 21 processors.

4.5 Dependency-Group Aggregation vs Single-Group Aggregation

In Single-Group Aggregation, a single group of tasks is formed from all the non-bottleneck tasks (as described in Chapter 3 and [22]). This section shows examples which demonstrate the improved accuracy obtained when using the dependency groups presented in this chapter.

Example 3: Cycle introduced by aggregation

This randomly generated example (which we will call Example 3, shown in Figure 4.4) shows the value of the dependency groups. It has 11 tasks and 11 processors. The
bottleneck task is $t_3$ with 89.81% saturation level, shown with a bold outline.

Applying the Single-Group Aggregation algorithm, we get the model in Figure 4.5. In the aggregated model, the bottleneck task $t_3$ and its processor $p_3$ are preserved and all other tasks except the user task $c_0$ are merged into one task (with one processor). This produces a cycle in the call graph as seen in Figure 4.5.

![Figure 4.4 LQN model of Example 3](image)

The model is structurally different, and the cycle creates a call explosion and an explosive increase in delay and drop in throughput. The LQN solver did not converge, but stopped with very large values for response time (giving 100% System throughput error and $3.78E+10\%$ System response time error).
The Dependency-Group Aggregation algorithm gives the second aggregated model shown in Figure 4.6. The bottleneck task \( t3 \) is preserved. There are two groups of tasks in the model in Figure 4.4 that are identified by Dependency-Group Aggregation algorithm. One group is “below” the bottleneck (and produces the \textit{MergedTask0} in Figure 4.6) and the other group is “above” the bottleneck (and produces \textit{MergedTask1} in Figure 4.6).

To summarize the results:

\textit{Accuracy: Relative absolute error for Example 3}

- Single-Group: 100% in throughput and 3.78E+10% in response time
- Dependency groups based on one bottleneck task: 23.55% in throughput and 19.06% in response time

Task aggregation based on groups provides a much better aggregated model, although the error is still large.
Further examination reveals a second highly saturated task $t_2$ (97.26% saturation level) in Example 3 in Figure 4.4 shown in dashed outline. Task $t_2$ is a direct caller of the bottleneck task, saturated due to pushback (waiting for service that is delayed by congestion). Preserving both $t_2$ and $t_3$ gives the third aggregated model shown in Figure 4.7.

**Accuracy: Relative absolute error for Example 3 (continued)**

- Using dependency groups based on two preserved (highly saturated) tasks: 6.91% in throughput and 6.47% in response time.

The saturation level of both of the preserved tasks remain similar to the original model. Task $t_2$’s and $t_3$’s level of saturation are 97.26% (same as original model) and 87.43% (changed by 2%) respectively.
We can draw two lessons from this example: first that it may be important to preserve more than one highly saturated task, and second that the grouping should avoid aggregations that introduce cyclical calls between aggregated entries. Cyclical calling changes the structure of the system and totally distorts the predictions.

Figure 4.7 Third aggregated model for Example 3 by generating groups preserving two heavily saturated tasks

**Example 4: Bottleneck processor**

Example 4 was randomly generated and is shown in Figure 4.8. It has 25 tasks and 25 processors along with a reference task and its processor, with a bottleneck at the processor $p09$ with 78.35% saturation (shown in bold outline). The task deployed on $p09$ is $t09$ (also shown in bold outline). Single-Group Aggregation gives the model presented in Figure 4.9, in which the bottleneck processor $p09$ along with its task $t09$ are preserved and all other tasks except the user task $c00$ are merged into one task (and their corresponding processors
are merged into one processor)

Figure 4.8 LQN model Example 4 with bottleneck processor \( p09 \) and its deployed task \( t09 \) in bold outline and second bottleneck task \( t05 \) in dashed outline

In the model of Figure 4.9 the System throughput error is 27.57% and System response time error is 21.61%, which are substantial. Analysis of Example 4 shows that there is a second bottleneck task \( t05 \) (shown in dashed outline) having 98% saturation which is a direct caller of the preserved task \( t09 \). Applying the simplification algorithm using task dependency groups which also preserves the second bottleneck, we get the aggregated model as presented in Figure 4.10, with 5 tasks and 5 processors along with the reference task and its processor. The results for the aggregation based on dependency groups are much better than those for Single-Group Aggregation:
Figure 4.9 Aggregated LQN model for Example 4 (shown in Fig. 4.8) following the Single-Group Aggregation

Figure 4.10 Aggregated model from Example 4 of Fig. 4.8 by generating groups preserving bottleneck processor and a second bottleneck task
**Accuracy: Relative absolute error for Example 4**

- Single-Group: 27.57% in throughput and 21.61% in response time. The saturation level of p09 is changed by 27%.
- Dependency-Group based on two preserved tasks and one bottleneck processor: 1.78% in throughput and 1.77% in response time. The saturation of t05 and p09 are 98.33% (changed by 0.34%) and 79.76% (changed by 1.8%) respectively.

**Example 5: Preserving multiple tasks**

This randomly generated Example 5 in Figure 4.11 shows the necessity of preserving multiple tasks to get better accuracy.

![Figure 4.11 Original LQN model of Example 5 with 10 tasks and 10 processors](image)

**Figure 4.11 Original LQN model of Example 5 with 10 tasks and 10 processors**

When preserving the one bottleneck task, this model had a substantial error when aggregated following the grouping strategy: over 49% in response time and over 97% in throughput. The simplified model following Dependency-Group Aggregation is shown in Figure 4.12.
Inspection of the details suggests that the error arises partly through aggregation of very different classes, both moderately heavy loaded, in one of the groups. This is an example of the known threat of errors due to class aggregation. To deal with this threat the method must preserve additional tasks. To show that preserving additional tasks is effective, Example 5 was aggregated repeatedly with one task added at each step to the list of preserved tasks, with the results shown in Table 4.1. At each step, the most saturated non preserved task or processor was chosen to be preserved. We see that we can obtain as small an error as we wish.

### 4.6 An incremental strategy for error control

From Table 4.1, it is observed that as more tasks are preserved depending on the level of saturation, the error becomes smaller. This suggests a simple strategy to improve an aggregation which gives too large an error:
Table 4.1 Errors of different aggregations of Example 5

<table>
<thead>
<tr>
<th>Preserved tasks</th>
<th>Throughput error (%)</th>
<th>Response time error (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>{t2}</td>
<td>97.14</td>
<td>49.25</td>
</tr>
<tr>
<td>{t2,t0}</td>
<td>14.29</td>
<td>13.12</td>
</tr>
<tr>
<td>{t2,t0,t5}</td>
<td>8.57</td>
<td>6.8</td>
</tr>
<tr>
<td>{t2,t0,t5,t6}</td>
<td>0</td>
<td>0.12</td>
</tr>
</tbody>
</table>

1. Choose the initial preserved tasks
2. Simplify based on the preserved tasks
3. Solve the simplified model and evaluate the error
4. If the error is too large, add another preserved task and repeat from 2, else terminate.

This will always terminate, since once all tasks are preserved, the error is zero. It requires:

- a value for a satisfactory level of error to give the stopping condition, and
- a strategy for choosing the “next” task to preserve.

In this work various error thresholds from 2% to 10% were examined and tasks were preserved in order of saturation in the original model (highest to lowest). Additional details and the evaluation of this error-control strategy is deferred to Chapter 6, where it is also used to examine the trade-off between accuracy and degree of simplification.

### 4.7 Dependency-Group Aggregation algorithm

Dependency-Group Aggregation is summarized by the high-level algorithm given in Listing 4.1.
Listing 4.1 Summarized algorithm for Dependency-Group Aggregation

```cpp
DependencyGroupAggregation(Model OriginalModel, Real TargetAccuracy,
List<Task> PreservedTasks, List<Processor> PreservedProcessors): Model
    /* PreservedTasks stores the preserved tasks (user defined and based on level
     of saturation) and the reference task */
    /* PreservedProcessors stores the preserved processors (user defined and
     based on level of saturation) and the processor containing reference task*/
    /* Make a copy of original model to keep original model unchanged. All
     modification of this algorithm are made on the SimplifiedModel. */

    Model SimplifiedModel := OriginalModel;
    Do activity and entry simplification on simplifiedModel based on Eq (3.2.1),
    (3.2.2), (3.2.3) and (3.2.4) from Chapter 3;

    Solve OriginalModel;
    /* ThroughputError below refers to Relative error in System throughput
     between original model and simplified model */
    Real ThroughputError := MAX_INT;

    While(ThroughputError > TargetAccuracy)
        Find next most highly saturated entity SatEnt;
        If (SatEnt is of type Task)
            Add SatEnt to PreservedTasks;
            Add SatEnt’s host to PreservedProcessors;
        Else if (SatEnt is of type Processor)
            Add SatEnt to PreservedProcessors;
            Add SatEnt’s task to PreservedTasks;
        End-If

        /* Create group for each preserved task */
        List<Task> Tasks := all tasks of the model except the PreservedTasks;
        Map<Task, Set<Task>> TPTmap; /* Task to Preserved Task Map */
        For each task T in Tasks
            Set<Task> CallerPT := All preservedTasks that call T;
            Add (T, CallerPT) to TPTMap;
        End-For

        For each set of tasks in the values of TPTmap
            List<Task> CalledT := all tasks from the keys of TPTMap that have
            same CallerPT; /* avoid taking same task twice by keeping a flag */
            Create a map called Groups having the CallerPT and its
            associated CalledT as (key,value);
```

64
It is worth mentioning that the tasks aggregation based on Dependency-Group doesn’t use the solution file as was needed for single group aggregation. Also the aggregation process can identify the groups and merge all the tasks within each group in a single step instead of merging two tasks at a time, as done in Single-Group Aggregation.

This chapter considered that each host in the LQN model has only one task deployed. In Chapter 5, an extension of this aggregation is presented that allows hosts to have multiple tasks deployed on it.
5 Chapter: Dependency-Group Aggregation with Shared Deployment

When multiple tasks are deployed on a common processor and the tasks belong to the same group, the first idea that comes to mind is that the deployed tasks can be easily aggregated as in Single-Group Aggregation. But this aggregation can result in calling cycles between tasks in the model, demonstrated in an example below. Also, if the tasks belong to different groups, then the process has to keep all those tasks un-aggregated since tasks from different groups cannot be aggregated according to Dependency-Group Aggregation. But keeping a number of tasks un-aggregated does not contribute to model simplification which is the main goal of this research.

This chapter presents a solution to these problems based on splitting of processors. This split-processor mechanism is considered as an extension of Dependency-Group Aggregation where rest of all the strategies and principles will remain same as in Dependency-Group Aggregation.

5.1 Calling cycles and processor splitting

Figure 5.1 presents a randomly generated LQN model with 8 tasks and 7 processors. Each processor has only one task deployed except the processor \( p3 \) that has tasks \( t3 \) and \( t6 \) deployed (the processor and the two tasks are shown in bold outline). Figure 5.2 presents the model after aggregating \( t3 \) and \( t6 \). The aggregated task is called \( \text{mergedTask1} \) (shown in bold outline in Figure 5.2). This aggregation creates a calling cycle between the merged task and \( t5 \) as shown in Figure 5.2.

To avoid creating cycle in the aggregated model, this chapter proposes to split the
Figure 5.1 An LQN example to show shared deployment of tasks $t3$ and $t6$ on the processor $p3$ processor in such a way that every task (that is deployed on the processor) is given its own processor (i.e., split processor). Figure 5.3 presents the model of Figure 5.1 after splitting the processor $p3$ (shown in bold). The split processors are called $p3\_splitt0$ and $p3\_splitt1$. $T3$ is deployed on $p3\_splitt0$ and $T6$ is deployed on $p3\_splitt1$. 
Figure 5.2 Tasks t3 and t6 are aggregated as mergedTask1 on processor p3 creating a calling cycle
5.2 Split processors

This section presents the process of splitting processors. The process is similar to shadow server described in Section 2.2.8, but in this case a processor on which several tasks are deployed is split into a set of processors (one for each task). The splitting process assumes that the processor scheduling has been modeled by the processor sharing scheduling discipline, and that there are no priorities among the tasks deployed on it. If these
assumptions are violated the processor should not be split, but it and its deployed tasks
should be preserved. Also splitting is not applied to a preserved processor.

First consider a single processor with the processor sharing scheduling discipline. The
response time equation for open arrivals and multiclass processor sharing is [35]:

\[ R_i = \frac{d_{T_i}(E_i)}{1 - d_{T_i}(E_i) \cdot f} \]

where, \( R_i \) is the response time of entry \( E_i \) of task \( T_i \),

\( d_{T_i}(E_i) \) is the service demand per system response, for the same entry, and

\( f \) is the System throughput.

Splitting a processor gives one task per processor, with a separate “split-processor (SP)”
for each task. To keep the response times equal for the deployed task before and after split,
a speed adjusting factor is used to scale down the speed of a “split-processor”.

The response time for the one class on the original processor is set equal to the response
time for that same class with its own processor (split-processor) scaled by a speed adjusting
factor \( SAF_{SP(T_i)} \), which gives the following equation:

\[ \frac{d_{T_i}(E_i)}{1 - \sum d_{T_i} \cdot f} = \frac{d_{T_i}(E_i) / SAF_{SP(T_i)}}{1 - \frac{d_{T_i}(E_i) \cdot f}{SAF_{SP(T_i)}}} \]

Here, \( d_{T_i}(E_i) \) represents the service demand of all the entries of all tasks \( T \) per system
response. Using the utilization law [35], the above equation is simplified as:

\[ => d_{T_i}(E_i) = \frac{d_{T_i}(E_i) / SAF_{SP(T_i)}}{1 - \frac{d_{T_i}(E_i) \cdot f}{SAF_{SP(T_i)}}} \]

where, \( U \) is the total utilization of all the tasks on the host \( H \).

\[ => d_{T_i}(E_i) \left( 1 - \frac{d_{T_i}(E_i) \cdot f}{SAF_{SP(T_i)}} \right) = \frac{d_{T_i}(E_i)}{SAF_{SP(T_i)}} (1 - U) \]
\[
=> SAF_{SP(Ti)} - d_{T_i(E_i)} * f = (1 - U)
\]
\[
=> SAF_{SP(Ti)} = 1 - U + d_{T_i(E_i)} * f
\]
\[
= 1 - \sum_{j \neq i} d_{T_j(E_j)} * f
\]
\[
= 1 - \sum_{j \neq i} u_j
\]

If there are multiple tasks that belong to the same group \( G(P) \) where \( P \) refers to the preserved tasks (see Section 4.2), they can be kept together on a shared split-processor. In such case, Eq (5.2.1) is written as,

\[
SAF_{SP(Ti \in G(P))} = 1 - \sum_{T_j \in G(P)} u_{T_j}
\] (5.2.1.1)

As a heuristic, this speed adjusting factor is applied to layered servers which do not have open arrivals, and to multiprocessors. The split-processor has the same multiplicity as the original processor. For a multi-core processor, \( u_j \) needs to be divided by the multiplicity \( m_H \) of the original host \( H \), as:

\[
SAF_{SP(Ti)} = 1 - \sum_{j \neq i} u_j / m_H
\] (5.2.2)

The speed factor \( S_{SP} \) of each split-processor \( SP \) is obtained by multiplying \( SAF_{SP(Ti)} \) with the speed factor \( S_H \) of the original host \( H \):

\[
S_{SP} = S_H * SAF_{SP(Ti)}
\] (5.2.3)

Since the next step is to do a group aggregation where tasks and processors are aggregated, the speed factors of the processors and the service demands of the associated tasks need to be normalized. This normalization is done by setting the speed factor of the processors to 1 and adjusting the deployed tasks’ service demands to give the required service time. This normalization can always be performed, to remove differences in speed factors across a set of processors. The demand adjusting factor, \( DAF \) of split-processor \( SP \) for task \( Ti \) is:

\[
DAF_{SP(Ti)} = 1 / S_{SP}
\] (5.2.4)
and, the adjusted service demand of task $T_i$ is:

$$ d_{Ti}' = d_{Ti} \times DAF_{SP(Ti)} $$  \hspace{1cm} (5.2.5) 

Section 5.4 shows the results obtained by applying this splitting process to layered servers in an example model.

5.3 Strategies for aggregating tasks with shared deployment

Using processor splitting, there are three strategies for aggregation of tasks that are deployed on a common processor, depending on the tasks’ dependency groups.

**Tasks belonging to a single dependency group:** If all the tasks that are deployed on a common processor belong to the same dependency group, they are aggregated into one task provided aggregating them does not create a cycle; if it does create a cycle the processor is split so each task has its own processor (see Section 5.1 for an example).

**Tasks belonging to multiple dependency groups:** When the tasks belong to multiple dependency groups, a sub-group of tasks is formed for each dependency group. The processor is split to give a separate processor for each sub-group, and the tasks in the sub-group are aggregated if no cycle is created (as above).

**Tasks on a preserved processor, multiple groups:** If the deployed tasks belong to different dependency groups, the preserved processor will not be split. Tasks that belong to the same dependency group are aggregated together (provided it does not create a cycle) but the aggregated tasks are all deployed on the same processor. As a result, the preserved processor may have fewer tasks than before aggregation. All the tasks on that processor (aggregated and non-aggregated) are considered as preserved tasks for further aggregation.

The result of applying these three strategies is a model in which tasks that share a processor are either preserved tasks or tasks which all belong to the same dependency group. Then
the Dependency-Group Aggregation process of Chapter 4 can be extended trivially to apply to this model. The extension is to allow a set of tasks in a dependency group to share a processor (which has no other tasks) and uses the same operations and equations on the tasks and processors related to the group.

Thus, the final updated process of LQN model simplification includes the following steps in ordered basis: a) entry aggregation, b) activity aggregation, c) preservation of a small number of tasks and processors based on their level of saturation (the preservation includes user defined tasks and processors as well), d) splitting processors and adjusting the service demands of the associated tasks, e) task aggregation based on the strategies of shared deployment for all processors of the original model having multiple tasks deployed, and f) task aggregation based on Dependency-Group Aggregation.

Again, for error control, if the simplified model is not accurate enough (e.g., throughput error $\leq 2$) then more tasks/processors are preserved and the process is repeated, as before. A high-level algorithm for the whole process is presented in Listing 5.1 in Section 5.5.

5.4 Application of split processor

The heuristic that applies the processor splitting process to layered servers and multiservers was tested on a number of examples. Figure 5.4 presents one of them (called Example 6) that has three processors containing multiple tasks. The processors are $p0$, $p3$ and $p5$, each of them has a speed factor of 1. Processor $p0$ holds two tasks: $t0$ and $t9$. Since they are from two different dependency groups, $p0$ is split and each task gets its own processor. The split processors for $p0$ are $p0_{split0}$ and $p0_{split1}$. Similarly, processor $p3$ holds two tasks: $t3$ and $t7$, each of them belong to different groups. So, $p3$ is also split in $p3_{split0}$ and
p3_splitt1. There are three tasks \( t_5, t_6 \) and \( t_8 \) deployed on the processor \( p_5 \). Among the three tasks, \( t_5 \) and \( t_8 \) belong to a same dependency group of tasks. So, \( t_5 \) and \( t_8 \) are aggregated as \textit{MergedTask0} following Dependency-Group Aggregation. Processor \( p_5 \) is split in \( p_5_{\text{split0}} \) (running \textit{MergedTask0}) and \( p_5_{\text{split1}} \) (running \( t_6 \)). The model after splitting all the processors is presented in Figure 5.5.
Figure 5.4 LQN model of Example 6
The following table presents the performance results in the form of service time of the deployed tasks before and after splitting and their percentage error for each processor. In the case of \( t_5 \) and \( t_8 \) the error in the combined service time (the weighted average of the two) is given, as this is what affects the solution.

![Figure 5.5 LQN model after splitting all the processors in Fig. 5.4](image-url)
Table 5.1 Service time of all the tasks originally deployed on processors p5, p6 and p8 before and after splitting the processors and their % error

<table>
<thead>
<tr>
<th>Task</th>
<th>Before split</th>
<th>After split</th>
<th>Error (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>t0</td>
<td>23.1882</td>
<td>23.3006</td>
<td>0.48473</td>
</tr>
<tr>
<td>t9</td>
<td>0.3698</td>
<td>0.3747</td>
<td>1.32504</td>
</tr>
<tr>
<td>t3</td>
<td>6.54135</td>
<td>6.6014</td>
<td>0.91801</td>
</tr>
<tr>
<td>t7</td>
<td>0.760626</td>
<td>0.7912</td>
<td>4.01958</td>
</tr>
<tr>
<td>t5</td>
<td>4.2887</td>
<td>6.3033</td>
<td>4.08272</td>
</tr>
<tr>
<td>t8</td>
<td>2.2829</td>
<td></td>
<td></td>
</tr>
<tr>
<td>t6</td>
<td>3.6194</td>
<td>3.62471</td>
<td>0.14671</td>
</tr>
</tbody>
</table>

The error is less than 1% for most tasks, only a few have error around 4%. The overall system throughput and system response time after splitting is around 2%.

5.5 Algorithm for Dependency-Group Aggregation with Shared Deployment

Listing 5.1 Summarized algorithm for shared deployment

```csharp
DependencyGroupAggregationSharedDeployment(Model OriginalModel, Real TargetAccuracy, List<Task> PreservedTasks, List<Processor> PreservedProcessors): Model

/* PreservedTasks stores the preserved tasks (user defined and based on level of saturation) and the reference task */
/* PreservedProcessors stores the preserved processors (user defined and based on level of saturation) and the processor containing reference task */
/* Make a copy of original model to keep original model unchanged. All modification of this algorithm are made on the simplifiedModel. */

Model SimplifiedModel := OriginalModel;
Do activity and entry simplification on simplifiedModel based on Eq (3.2.1.1), (3.2.1.2), (3.2.1.3) and (3.2.1.4) from Chapter 3;

Solve OriginalModel;

/* ThroughputError below refers to Relative error in System throughput between original model and simplified model */
Real ThroughputError := MAX_INT;

While(ThroughputError > TargetAccuracy)
    Find next most highly saturated entity SatEnt;
```
If (SatEnt is of type Task)
   Add SatEnt to PreservedTasks;
   Add SatEnt’s host to PreservedProcessors;
Else if (SatEnt is of type Processor)
   Add SatEnt to PreservedProcessors;
   Add SatEnt’s task to PreservedTasks;
End-If

List<Processor> Processors := all processors of the model that have multiple tasks;

/* For each shared deployment, split and merge according to the strategies presented in Section 5.3
For each Processor P from Processors
   List<Task> CallerPT = All PreservedTasks that call the deployed tasks on P;
   List<List<Task>> CalledT = For each task in CallerPT, list of called Tasks deployed on P;
   /* If a deployed task is a preserved task, its callerPT will be itself */
   /* A case where all the deployed tasks belong to the same dependency Group */
   If (CallerPT has 1 task)
      Check for cycle by following the calling path and look for any branch that leads to the same source;
      If(cycle is found)
         Split the processor by the number of tasks in CalledTasks;
         For each split of the processor HS
            Calculate the speed factor by using Eq (5.2.1), Eq (5.2.2) and Eq (5.2.3);
            Deploy the associated task on HS;
         End-For
      Else if (cycle is not found)
         /* Dependency-Group Aggregation */
         Aggregate the tasks using Eq (4.4.1), (4.4.2), (4.4.3), (4.4.4), (4.4.7);
      End-If
   /* A case where all the deployed tasks belong to multiple dependency groups
   Else if (CallerPT has multiple tasks && P is not in
Split the processor by the number of tasks in CallerPT; 
For each split of the processor HS 
  Calculate the speed factor by using Eq (5.2.1), Eq (5.2.2) and Eq (5.2.3); 
  Deploy the associated tasks List<Task> CalledTasks on HS; 
  List<Task> MergeableMembers = All tasks in CalledTasks that are not preserved tasks; 
  List<Task> NonMergeableMembers = All tasks that are preserved tasks; 
  If(MergeableMembers is not empty) 
    Check for cycle in MegeableMembers by following the calling path and look for any branch that leads to the same source; 
    If(cycle is found) 
      Break; /* the tasks will be kept and aggregated later with other tasks in Dependency-Group Aggregation */ 
    End-If 
  Else if (MergeableMembers has multiple tasks and cycle is not found) 
    /* Dependency-Group Aggregation */ 
    Aggregate the tasks using Eq (4.4.1), (4.4.2), (4.4.3), (4.4.4), (4.4.7); 
  End-If 
End-For 

/* A case where all the deployed tasks belong to multiple dependency groups and are deployed on a preserved processor */ 
Else if (CallerPT has multiple tasks && P is a preserved processor) 
  For each task CallerTask in CallerPT 
    List<Task> CalledTasks = All the deployed tasks on P called by CallerTask; 
    List<Task> MergeableMembers = All tasks in CalledTasks that are not preserved tasks; 
    List<T> NonMergeableMembers = All tasks that are preserved tasks; 
    If(MergeableMembers is not empty) 
      Check for cycle in MegeableMembers by following the calling path and look for any branch that leads to the same source; 
      If(cycle is found) 
        Break; /* the tasks will be kept and aggregated later */
with other tasks in Dependency-Group Aggregation */
End-If
Else if (MergeableMembers has multiple tasks and cycle is not found)
/* Dependency-Group Aggregation */
Aggregate the tasks using Eq (4.4.1), (4.4.2), (4.4.3), (4.4.4), (4.4.7);
End-If
End-For
End-If
End-For
For each processor in the model
If (the processor has speed factor not equals to 1)
Speed factor will be normalized and service demand of the deployed tasks will be adjusted using Eq (5.2.3) and (5.2.4);
End-If
All tasks of the model will be aggregated according to Dependencygroup Aggregation as shown in Listing 4.1;
Update SimplifiedModel with these aggregated processors, tasks and entries;
Calculate ThroughputError using Eq (3.2.1.7);
End-While
Return SimplifiedModel;
6 Chapter: Experiments and Case Studies

This chapter presents the evaluation of the simplification process by applying it to a large number of models of five different sizes from 11 tasks to 31 tasks, with randomly generated structure and parameters. The models were generated by the library program lqngen.exe. 150 random models were generated (30 of each size), each having a random number X of users for the workload, and 150 additional models were created with the number of users doubled to 2X. The evaluation addresses the effectiveness of simplification and the tradeoff between simplification and approximation accuracy.

Since our goal is to simplify large models, the scalability of the process was examined through cases with still larger models.

6.1 Empirical evaluation of simplification accuracy

The greater the simplification, the greater an approximation error we can expect. As described in Chapter 4 and 5, the process begins with a small number of preserved tasks (maximum simplification) and adds preserved tasks (reducing the simplification) to achieve a target accuracy. For the evaluation, this process was tracked and the intermediate accuracy values were recorded, giving a sequence of values of preserved tasks, percentage reduction in size, and error in predicted throughput. Figure 6.1 shows these values for one example of size 31. Note that the percentage reduction in model size is the reduction in the sum of tasks plus processors, versus the original model. That is, an 80% reduction means a simplified model of 1/5 the size.

In the graph of Figure 6.1, the x axis represents the number of preserved tasks that are being added step by step, and the two y-axes show the accuracy and the reduction in model size. It is to be noted that when preserving 3 tasks, a processor is preserved along with its
deployed tasks which makes 6 tasks, which greatly reduces the reduction factor. It is

![Figure 6.1 Trajectory of accuracy and size reduction as preserved tasks are added]

observed from the graph that as the preserved tasks are added, the throughput error improves (y axis on the left represents throughput error in %). For 6 preserved tasks, the throughput error is more than 2%. For 18 preserved tasks, throughput error is reduced to less than 2%. However, as the preserved tasks are added, the reduction in model size gets smaller (y axis on right side presents reduction of model in size in %). So, if 50% reduction is required for simplification, throughput error is 4.5% for this particular model.

Some cases could not achieve 2%. Even with zero simplification there may be some error, due to the aggregation of entries before aggregation of tasks begins.

Before examining the approximation errors in detail, it is useful to consider the nature of the relationship between preserving tasks, and the resulting model size reduction. The processors and tasks in the simplified model include the preserved tasks, their processors,
and tasks representing the aggregation groups; the number of aggregation groups depends on the task call relationships, so there is no simple relationship. Figure 6.2 shows a scatter plot of percent reduction versus percent preserved tasks, for all evaluations.

Figure 6.2 Preserved tasks vs. reduction in model size for 300 cases

While the inverse relationship is clear, there is a wide scatter in reduction values for a given fraction of preserved tasks. We will use percent reduction as the primary metric for the degree of simplification.

Figure 6.3 is a scatter plot of throughput error vs. preserved tasks (as a percentage of model size) for all the different evaluations for all 300 cases. This graph clearly shows the broad trend, that as the number of preserved tasks increases, throughput error decreases. At the left, for very few preserved tasks, throughput error can be from almost 0 to about 98%. But for more than 10% of preserved tasks, all the errors are less than 15%. Above 50% preserved tasks, the error is less than 8%, and above that the further decrease varies greatly between cases. Some cases resist further decrease, but in most cases the 2% target is reached.
Graphs from Figure 6.4 to Figure 6.8 examine this data in more detail, to see how the error behaves for different sizes of the original model. They also do the comparison on the basis of model size reduction rather than preserved tasks, since that is the metric that matters most. Figure 6.4 shows a graph that represents throughput error vs. the percent reduction in model size for 60 cases containing 11 tasks. To read the graph, bear in mind that for each case the process gives a series of points starting with high reductions on the right and moving towards lower reductions with more preserved tasks, terminating at either no reduction, or when the error falls below 2%. Thus all the points below 2% are the terminating point for that case. If these cases had been continued for even more preserved tasks and smaller reductions there would be many more points with low errors, on the left-
hand side of the graph.

We can see in the graph that even for no reduction, there can be throughput error as high as 7%. This is because although there is no task aggregation, the entry aggregation gives this error. For up to 43% reduction, most errors were less than 5%, many more were less than 10% and a few were in the range 10-20%. Above this reduction, about a quarter of the cases had errors above 10%, and there are some cases where throughput error is quite large. We need to keep it in mind that smaller the model size is, bigger the throughput error is for the same reduction. This trend is also represented in the next graphs.

![Graph](image)

**Figure 6.4 Throughput error vs. reduction in model size for 60 cases containing 11 tasks**

Figure 6.5 presents the graph for the same analysis (throughput error vs. model size reduction) for the 60 random models containing 16 tasks each. From the graph we can see that, for about 65% of reduction, the error is within 8 to 10%. For the higher reduction, the
error for most of the models within 10% but very few have higher error. Figure 6.6, Figure 6.7 and Figure 6.8 show the graphs for the same analysis for the models of 21 tasks, 26 tasks and 31 tasks respectively. In these graphs, we see that for more reduction in the simplified model (e.g., 70% of reduction) we get more throughput error.

Figure 6.5 Throughput error vs. reduction in model size for 60 cases containing 16 tasks
Figure 6.6 Throughput error vs. reduction in model size for 60 cases containing 21 tasks

Figure 6.7 Throughput error vs. reduction in model size for 60 cases containing 26 tasks
The most important observation from Figures 6.4 – 6.8 is that larger models give better reductions. This is important for our goal of simplifying large models; it is not so important to a user to obtain a large percent reduction in the size of a model which is already small.

Figure 6.9 relates the average reduction achieved, to the model size and throughput error. For 2% error, the average reduction achieved was 50% or less for all model sizes, and for lower accuracy the average reductions were better (a clear upward trend) but not very much better, only the largest models and lowest accuracy giving over 60% average reduction. From this graph we can deduce an expected reduction roughly in the range 40% – 60%, somewhat larger for larger models.
Figure 6.9 Average reduction for a certain model size and error in 300 cases

Figure 6.10 looks at the data from another viewpoint. For each model size and chosen accuracy, it reports what percentage of cases was able to reach that accuracy. These are very encouraging results. Every case achieved 90% accuracy, and well over 90% achieved 95%, including all model sizes. For the larger model sizes over 80% achieved over 98% accuracy, but for 11 and 16-task model the fraction was lower.

Figure 6.10 Average number of cases simplified in a certain model size and error in 300 cases
From all these experiments, we can say that the simplification process works better if the throughput error allowed is within 5% to 8%. Then the average reduction in model size can be 60% to 70%. Also, the larger the model, the better the simplification works.

6.2 Scalability

In this section, a much larger and more complex case is included here to see the scalability of the simplification process in terms of model size. Figure 6.11 shows a model called “case-50” with 50 tasks and 50 processors that was generated randomly [21]. In this example, we preserve one resource or more at a time to show how amount of preservation effects the model’s accuracy.

The tasks outlined in bold were preserved based on high saturation level. Figure 6.12 shows a first aggregated model of “case-50” with 7 tasks and 7 processors based on preserving only the bottleneck task and other highly saturated tasks. If we preserve the next most highly saturated resource which is processor \textit{p31} (47.92% saturated) and its deployed task \textit{t31} as shown in Figure. 6.13, the error is reduced as shown in Table 6.1.
Figure 6.11 LQN model of “case-50” with 50 tasks and 50 processors
The errors under the simplest task-preservation in the first case are barely satisfactory but the reduction is very good; one improvement step gives a satisfactory accuracy with still a very good amount of reduction.
Figure 6.13 Second aggregated model of “case-50” with 9 tasks and 9 processors

6.3 Case studies

The case studies in this section show the application of the simplification process to models related to real industrial software systems.

6.3.1 Case Study 1: Business Reporting System

The first Case study example is a Business Reporting System described in [34], which can create reports on business processes from a data base. Figure 6.14 (a) shows the original LQN model (generated from a design specified as a Palladio Component Model) from [34]. It has 43 tasks with a large number of entries and activities, running on 43 processors, so it has just one task per processor.
Figure 6.14 (b) shows the final simplified model of the original model from [34]. The two highly utilized tasks are preserved in the simplified model: one is the software bottleneck of the system and the other is a direct caller of the bottleneck tasks, saturated due to pushback (waiting for services that are delayed by congestion). These two tasks remain the most highly utilized in the simplified model. All the other tasks are merged into a single task.

For interest, the two highly utilized tasks were merged, the relative error in throughput was 23.74% and in response time is 69.93%. It is clearly correct to preserve them separately.

Figure 6.14 Layered Queueing Network of the Business Reporting System generated from PCM

The performance results after different steps of the simplification process (i.e., activity, entry and task simplification) are shown in Figures 6.15 and 6.16. The differences are small throughout.
Figure 6.15 System Throughput (jobs/ms) of case study 1 after three simplification operations

Figure 6.16 Relative error in system throughput after every simplification operation

Figure 6.15 shows that the system throughput is very little changed by the simplification. Figure 6.16 further shows that the error caused by task simplification is higher than that of entry simplification, which in turn is higher than that of activity simplification. Also, the simplification process incurs larger percentage errors for fewer customers.
Figure 6.17 System response time after three simplification operations

Figure 6.17 shows similar effects on the response time accuracy. With more customers (e.g., \( N>20 \)), the error in response time is less than 5% (see Figure 6.18). However, for small \( N \) (e.g., \( N=10 \)), the error is larger. What is important is that, throughout the simplification process, the bottlenecks of the system remain unchanged with similar utilizations. For example, with \( N = 10 \) the two heaviest task utilizations (73% and 70%) changed by less than 1%.

Figure 6.18 Relative error in system response time after each simplification operations

6.3.2 Case study 2: Airline Reservation System

This case study, shown in Figure 6.19, is taken from A. Faisal’s thesis [11]. Users make
two types of request in this model: browse and purchase. Requests go through a controller to be dispatched to a service layer. Browse requests then go to a catalog server, which replies to the service layer with product information, which is sent to controller and then to the user. Purchase requests go to a purchase server. The purchase request-responses are encrypted. The model uses REST and ActiveMQ as middleware. An ActiveMQ broker is placed between the user and the controller. Requests for browse operations go from user to the broker’s queue, from where it is picked up by the controller. Upon generating a response, the controller replies to the broker and the broker forwards it to the user.

The model for Case 2 has 14 tasks including the reference task and 6 processors. After the initial execution of the model, the processor ServiceH is found to be the bottleneck with utilization 90.8%. All the tasks deployed on this processor are merged. Figure 6.20 shows the first aggregated model after preserving ServiceH along with its deployed tasks.
Figure 6.19 ARS_v4 where REST and ActiveMQ are used together [11]
Figure 6.20 First aggregated model of ARS_v4 where bottleneck resource ServiceH is preserved, deployed tasks on it are merged.

The throughput error for this first aggregation is 3.85% and the reduction in model size is 61.9%. Since the error exceeds 2% the next most highly utilized resource ControllerH is also preserved in a second aggregation, which is presented in Figure 6.21.
Figure 6.21 Second aggregated model of ARS_v4 where ServiceH and ControllerH are preserved, deployed tasks on each processor are merged

For the second model, the throughput error is 2.37% and reduction is 52.38%. Since throughput error is still more than 2%, the next highly utilized resource is preserved which is CatalogH with utilization 27.35. Figure 6.22 shows the model preserving all three processors.
Figure 6.22 Third aggregated model of ARS_v4 where ServiceH, ControllerH and CatalogH are preserved, deployed tasks on each processor are merged

For this model (Figure 6.22), the throughput error is 1.49% and the reduction is 42.86%.

Table 6.2 shows all the aggregations and their effects.
<table>
<thead>
<tr>
<th>Figure</th>
<th>Preserved resources</th>
<th>Throughput error %</th>
<th>Response time error %</th>
<th>Reduction %</th>
</tr>
</thead>
<tbody>
<tr>
<td>Figure 6.20</td>
<td>1 processor</td>
<td>3.85</td>
<td>3.71</td>
<td>61.9</td>
</tr>
<tr>
<td>Figure 6.21</td>
<td>2 processors</td>
<td>2.37</td>
<td>2.32</td>
<td>52.38</td>
</tr>
<tr>
<td>Figure 6.22</td>
<td>3 processors</td>
<td>1.49</td>
<td>1.46</td>
<td>42.86</td>
</tr>
</tbody>
</table>

6.3.3 Case Study 3: Linux Network File System

This case study model shown in Figure 6.23 is taken from G. Frank’s thesis [14]. It was used to study the performance of the Network File System implemented in the Linux. The model has 7 tasks including the reference task and 4 processors. The first preserved resource is task `rpcnfsd` with utilization 68.198%. Since this task is deployed on the processor `server` along with another task `servercache`, the processor is split. As a result, `rpcnfsd` is deployed on its own processor `server_splitt0` which is preserved, while `servercache` is deployed on its own processor, which is then merged with others. The first aggregated model is shown in Figure 6.24.
Figure 6.23 Layered Queueing Network of principle NFS operations [14]
The model above gives throughput error of 58.82% which is very large, while the reduction is only 27.27%. So, the next highly utilized resource is preserved: processor server with utilization of 37.99% along with its deployed task. This time, processor server is not split and both the tasks rpcnfsd and servercache are preserved. Figure 6.25 provides the second aggregated model of this case study.
Figure 6.25 Second aggregated model of case study 3, preserving task \textit{rpcnfsd} and processor \textit{server}

This aggregation gives throughput error of 4.35% and a reduction of 18.18%. Although the error is more than 2%, still it is quite a small error. Also it is found that no more reduction is possible if more resources are preserved. The overall reduction is not large in this case.
study, possibly related to the small size of the original model (as discussed earlier). Table 6.3 shows the detailed results for both the aggregations.

<table>
<thead>
<tr>
<th>Figure</th>
<th>Preserved task</th>
<th>Preserved processor</th>
<th>Throughput error %</th>
<th>Response time error %</th>
<th>Reduction %</th>
</tr>
</thead>
<tbody>
<tr>
<td>Figure 6.24</td>
<td>1</td>
<td>0</td>
<td>58.82</td>
<td>37.03</td>
<td>27.27</td>
</tr>
<tr>
<td>Figure 6.25</td>
<td>2</td>
<td>1</td>
<td>4.35</td>
<td>4.27</td>
<td>18.18</td>
</tr>
</tbody>
</table>

### 6.3.4 Case Study 4: Large Telephony Switch

The case study [15] presented in this section is an example of voice telephony switch architecture. The study was used to identify the bottleneck of the system and improve the system by shifting and relaxing the bottleneck. The LQN model of the architecture is presented in Figure 6.26. The model has 22 tasks, 19 processors and 1000 users. Table 6.4 shows the throughput and response time error between the simplified and the original model, as well as the reduction of the simplified model after preserving different number of resources.

One of the simplified models shown in Figure 6.27 has 6 tasks and 6 processors. The throughput error for this model is 5.21% and the response time error is 4.95%. Although the errors are not less than 2%, still they are not very large. Also, this simplification provides 70.70% reduction from the original model.
Figure 6.26 An LQN model of Telephony Switch Architecture [15]
Figure 6.27 A simplified model of Case Study 4 with 6 tasks and 6 processors with 5.21% throughput error, 4.95% response time error and 70.70% reduction
The final simplified model is generated after preserving more tasks to get throughput error less than 2%. The model is presented in Figure 6.29. It has 15 tasks and 15 processors.
Table 6.4 Model simplification results for different aggregations on Case study 4

<table>
<thead>
<tr>
<th>Preserved task</th>
<th>Preserved processor</th>
<th>Throughput error (%)</th>
<th>Response time error (%)</th>
<th>Reduction (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>7.32</td>
<td>6.15</td>
<td>85.37</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>5.38</td>
<td>5.11</td>
<td>80.49</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>5.21</td>
<td>4.95</td>
<td>70.73</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>4.11</td>
<td>3.95</td>
<td>65.85</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>3.18</td>
<td>3.09</td>
<td>60.98</td>
</tr>
<tr>
<td>6</td>
<td>1</td>
<td>2.67</td>
<td>2.6</td>
<td>51.22</td>
</tr>
<tr>
<td>7</td>
<td>1</td>
<td>2.15</td>
<td>2.11</td>
<td>41.46</td>
</tr>
<tr>
<td>8</td>
<td>1</td>
<td>2.15</td>
<td>2.11</td>
<td>36.59</td>
</tr>
<tr>
<td>12</td>
<td>5</td>
<td>1.41</td>
<td>1.39</td>
<td>26.83</td>
</tr>
</tbody>
</table>

It is observed that, preserving 1 task gives 85.37% reduction from the original model while providing 7.32% throughput error and 6.15% response time error. As more resources are preserved, the error gets reduced.

6.3.5 Case Study 5: E-Commerce

The LQN model in this case study is an example of an E-Commerce system from [28]. The example deals with optimal deployment with replicated (scaled-out) servers in a cloud [28]. The authors used this model architecture as a template to create different applications (same structure, different parameters) to study co-locating many applications in a cloud. The model is shown in Figure 6.29. This is a small model with 8 tasks, 6 processors and has 100 users. The simplified model shown in Figure 6.30 has only 3 tasks and 3 processors with 0.95% throughput and response time error. Also the reduction is 57.14% from the original model. This is an excellent approximation.
Figure 6.29 LQN model of an E-Commerce example
6.3.6 Summary of case studies

Among the five case studies, the highest target accuracy of 2% was obtained in all but one case. Figure 6.31 shows the throughput error and the reduction for each case study. The minimum error was 0.06% and the maximum reduction was 90.7%. Both were obtained from Case Study 1. Note that Case Study 1 was the largest model among all having 43 tasks and 43 processors. On the contrary, the worst reduction was achieved in Case Study 3, which was the smallest model. This model has 7 tasks and 4 processors. The only case that could not achieve 2% accuracy was Case Study 3. The minimum error achieved in this model was 4.35% and the reduction was 18.18%. This supports our previous argument that usually the simplification of bigger models gives better reduction with better accuracy. Case Study 4 is an interesting model. It has 22 tasks and 19 processors and it was reduced by 26.83% (12 preserved tasks and 5 preserved processors) when an accuracy level of 2%
was targeted. But, if the targeted accuracy level is changed to 4.11% (as shown in Table 6.4), a reduction of 65.85% (4 preserved tasks and 1 preserved processor) with throughput error of 3.95% is achieved. Higher reduction can be achieved by preserving fewer resources and targeting a lower accuracy.

![Throughput error(%), Case Study 1: 0.06, Case Study 2: 1.49, Case Study 3: 4.35, Case Study 4: 1.41, Case Study 5: 0.95, Reduction(%), Case Study 1: 90.7, Case Study 2: 42.86, Case Study 3: 18.18, Case Study 4: 26.83, Case Study 5: 57.14](image)

**Figure 6.31 Throughput error and reduction in five case studies**
7 Chapter: Automation and Tooling

This chapter discusses the automation of the simplification process and its implementation. A Java program is implemented which executes all four simplification operations (aggregation of activities, entries, tasks and processors) on the input model and generates a set of simplified models as output.

7.1 Software requirements

The entire simplification process was automated in the tool “LQN Model Simplifier”. Java jdk 1.8 and Eclipse Neon IDE were used for the implementation. It is a console-based program which can run on both Windows and Linux.

LQN Model Simplifier takes a fully functional LQN model, a target accuracy for simplification and a list of tasks and processors to be preserved as inputs and generates a series of simplified LQN models that have activities, entries, tasks and processors merged. The simplification is done step by step so that the modeler can go back to any step of simplification to verify and compare the performance results with the original one.

The resultant models have been found to be free of errors, and to have no missing information (i.e., performance parameters). The simplified models are traceable since we can go back to any simplification step and analyze the model.

7.2 Software architecture

This section presents the software architecture of the implementation of Dependency-Group Aggregation with Shared Deployment which is based on MVC (model-view-controller) pattern [5]. Figure 7.1 shows the class diagram that presents an abstract view of the implementation. There is a large number of utility methods in each class which we
The classes are described as follows:

**ConsoleView:** This class acts as the view of this program. It has a `main()` method which takes an LQN model, a target accuracy and a list of preserved tasks and processors as inputs from the user. The user needs to give the name of the LQN model that needs to be
simplified. After that ConsoleView sends the input to the AggregationController for further processing.

**AggregationController:** This class acts as the controller of this program. The class gets the input model from ConsoleView and saves the model as the input model for running the simplification. It also saves it as a base model since the program compares each simplified model with the base model. It then executes command to run the model. For execution, it calls different methods from ExecuteCommand class. Then it saves the output model. The class uses different accessors and mutators for saving and returning models and variables. Through findAggregationType(Model), the class identifies which type of aggregation the input model needs (i.e. activity aggregation, task aggregation etc.). Depending on the type of aggregation input model requires, callAggregation(Model) calls different simplification operations on the abstract class Aggregation. Every time an aggregation is done (entry, activity or task aggregation), this class saves the model as a simplified model for that aggregation and continues the following step of aggregation by setting it as an input model for that aggregation.

**ExecuteCommand:** This class executes system commands to run the LQN model using the LQNS solver [26]. To run the LQN model using the LQNS solver, we use the command line “lqns –x infile.lqn” and it produces an output files with the default name infile.out and an xml file infile.lqxo. Command “lqn2xml” is used to convert an LQN file that follows the textual syntax from [12] to an XML file, since the LQN API used in the program requires the model in xml format. The LQSIM simulator is invoked as “lqsim [run controls] infile.lqn” and generates an output file infile.out, which includes confidence intervals on the estimated performance parameters. More information on LQNS and LQSIM can be
found at [12]. The methods
execWinCmd(String) and execLinuxCmd(String) execute the LQN model in Windows and Linux OS respectively.

**LQN_API:** The LQN library provides the model for the architecture. The API for LQN provides a number of classes, including Activity.java, Entry.java, Task.java, Processor.java, GenericCall.java, ActivityCall.java, Phase.java etc. These classes provide access to model parameters and output results. **LQN_API** is used by both the classes AggregationController and Aggregation.

**PerformanceResults:** AggregationController contains a list of PerformanceResults on which each object of PerformanceResults comprises information about each simplified model that the program generates after entry, activity and task simplification. The information includes: model name, throughput, number of preserved tasks and number of preserved processors that the generated simplified model has.

**Aggregation:** Aggregation is part of the model of this program. It is an abstract class that is extended by three sub classes: ActivityAggregation, EntryAggregation and TaskAggregation. The constructor of this class takes an object of AggregationController and a list of tasks as parameters. So, AggregationController calls different types of aggregation on this abstract class by sending the parameters: itself and the task list on which aggregation needs to be done. Aggregation has an abstract method doAggregation() which is overridden by its three subclasses to perform aggregation on the list of tasks. There is another important method called cloneModel(Model) which makes a copy of the input model and aggregation is done on this model. The model is saved as a simplified model and returned after the aggregation is done. It acts as the input model for the following step
of the aggregation. This class also stores the reference entry throughput and response time so that they can be later compared with results for other models.

**ActivityAggregation:** The method `doAggregation()` returns a model where all the activities are aggregated.

**EntryAggregation:** The overridden method `doAggregation()` returns a model with one entry per task. The method, `mergeIncomingCalls(entries: Vector<Entry>, task: Task)` merges all the incoming calls of the merging entries. Similarly, `mergeOutgoingCalls(entries: Vector<Entry>, task: Task)` merges all the outgoing calls from merging entries. `calculateSD()` calculates the service demand of the aggregated entry.

**TaskAggregation:** This is the class that handles the algorithms of *Dependency-Group Aggregation* and *Shared Deployment*. The method `doAggregation()` controls and coordinates all the operations from splitting processors to aggregating tasks and processors. The class has `compThr()` and `compResp()` methods to compare throughput and response time respectively between the base model and the simplified model. The method `setPrsrvedResources(Model)` sets the preserved tasks and processors in variables `prsrvedTasks` and `prsrvedProcessors`. The method `splitProc(Processor)` takes the parameter Processor and identifies the strategy of shared deployment that should be applied to the processor. Based on the identified strategy, it calls `singleGrp(Processor)`, `MultipleGrps(Processor)` or `prsrvedProcMultipleGrps(Processor)` for splitting processors and aggregating tasks on split processors. The class has `createGrps()` method that identifies and creates dependency groups of tasks. `setTPTMap()` creates a map called `TPTMap` which is task to preserved tasks map. This map is used to create dependency groups. `dependencyGrpAggregation()` is
called after the operations of shared deployment from the method `doAggregation()` to perform task aggregation within each dependency group. `adjustSD()` is used to adjust the service demand of the entry of a task whose processor gets normalized. The methods `mergeIncomingCalls(...)` and `merOutgoingCalls(...)` are used to merge incoming and outgoing calls respectively for merging tasks. Method `isCircular(Entry, Vector<Entry>)` is used to check if the aggregating entries can create any cycle. `calculateSD(Vector<Entry>)` calculates the service demand of the aggregated task.

### 7.3 Use of the Model Simplifier

The software takes as input:

- an LQN model
- a target accuracy (presently fixed at 2%)
- a list of tasks and processors to be preserved

It gives as output:

- a directory with a set of simplified models as files that include the performance results (in *.lqxo format as described in [12])
- a table summarizing the preserved elements in each simplified model, and the results, suitable to be loaded into a spreadsheet program for analysis and graphing.
8 Chapter: Conclusions

The thesis objective from Chapter 1 is to develop a process for automatically simplifying a performance model to an essential core level of detail governed by accuracy requirements, which include retaining the performance bottleneck(s) from the original model and keeping the relative error in throughput and response time as small as possible.

8.1 The approach

The approach taken in the thesis is to aggregate parts of the model and preserve others, as opposed to creating a simple model from scratch. This approach provides traceability of the preserved parts, so that they can be changed independently of the reduced parts for sensitivity studies. The aggregation is governed by four principles as described in Section 3.1:

1. Capacity limit principle,
2. Total workload principle,
3.Concurrency principle,
4. Dependency principle

The model simplification process requires the original model to be solved just once. The process successfully reduces the size of a given LQN model while still giving acceptable accuracy of the results. It allows to achieve any desired accuracy by incrementally adding to the preserved parts of the model. Acceptable accuracy in model simplification depends on the user requirement. Throughput errors ranging from 2% to 10% and reductions ranging upwards from 40% are considered here to be potentially useful.
8.2 Major findings

The major findings from the experiments are:

1. Usually, larger original models are reduced more and their simplification gives better accuracy. In general, the expected reduction in simplification is 40% to 60% as shown in Figure 6.9, and it is larger for larger models. For larger model sizes, more than 80% models achieve less than 2% error. For smaller models, the accuracy is lower, as shown in Figure 6.10.

2. There is a tradeoff between accuracy and reduction. Through the experiments it was found that the tradeoff is model size dependent. As shown in Figure 6.10, different size models achieve 10% error and over 90% cases achieve 5% error.

3. The number of preserved tasks and throughput errors are inversely proportional. As shown in Figure 6.3, the errors are less than 15% for more than 10% of preserved tasks. For 50% or more preserved tasks, the average error is less than 8% and in most cases it reaches the 2% target.

4. More users sometimes give better accuracy. As shown in Case Study 1 in Section 6.3.1, the error is smaller for a larger population. Also, from the experiments of 300 case models, it was observed that when doubling the number of users compared to the original models, the accuracy in the simplified models is either better than or similar to that of the original models.

5. An approximation is used for splitting processors in Chapter 5; the chapter includes experimental results that show that its error is small.
8.3 Contributions

The contributions identified in Section 1.4 are provided as follows.

- **Single-Group Aggregation of LQN models:** This is the first approach for LQN model simplification, where a number of activities, entries, tasks and processors are aggregated. The process compacts a given LQN model to its smallest possible size by reducing non-bottleneck task and processor resources to a single task and a processor. Only one bottleneck along with the reference task are preserved. The process is easy to apply and is very successful in dramatically reducing the model size in many cases while retaining the mean throughput and response time.

  The major limitations with Single-Group Aggregation are
  
  - It can create cycles in the simplified model in some cases;
  - It does not preserve a second or third bottleneck. As a result, the accuracy may suffer in the simplified models.

- **Dependency-Group Aggregation of LQN models:** This is an improved and more flexible approach of model simplification where a number of resources (tasks and processors) are preserved based on their utilizations. The other resources are aggregated into one. In LQN, bottlenecks are naturally layered due to “push back”. Since the process preserves multiple bottlenecks, it allows the effect of bottlenecks to be captured. It is a fully automated process of simplifying LQN models. This aggregation ensures that there are no calling cycles in the simplified model.

- **Dependency-Group Aggregation with Shared Deployment:** In Dependency-Group Aggregation, it was considered that there is only one task per processor in the original model. This assumption is eliminated in Dependency-Group Aggregation with Shared
Deployment. Here, each processor can have multiple tasks. There is a special analysis to prevent the introduction of cyclic calling, when tasks are deployed on a common processor. Processors are split, and tasks are aggregated on processors based on three strategies, as presented in Section 5.3.

- **Experiments and Case studies:** The simplification process is evaluated through 150 cases of LQN models of five different sizes from 11 tasks to 31 tasks (30 of each size), with randomly generated structures and parameters. Each model has a random number of users \( X \) for the workload. 150 additional models are generated with \( 2X \) users. The simplification starts by preserving a small number of tasks and adds more to get the required accuracy. The major findings of the experiments are presented in Section 8.2. Section 6.3 presents five case studies based on models of real industrial systems ranging from 7 tasks to 43 tasks, and Section 6.2 presents a random model of 50 tasks. All these cases confirm the conclusions from the random models presented in Section 6.1.

- **Tool Implementation:** A Java program called “LQN Model Simplifier” was built to perform model simplification. It is fully automated. It takes an LQN model as input and generates a set of simplified models that have activities, entries, tasks and processors merged. Since all the models are saved, it is possible to trace the tasks that are merged into an aggregated task. It is also possible to keep track of the identified dependency groups.

8.4 Limitations

The simplification process proposed in this thesis has the following limitations:

- It is applicable to only a single class of users.
- It is not applicable when there is a cycle in the call graphs.
- It cannot aggregate parallel activities in the same execution path.
- It cannot aggregate tasks with priorities.
- In its current state, it has not dealt with phases or with activities that create a second phase, or with forwarding and asynchronous calls.
- It is only applicable to processors whose scheduling can be modeled by processor sharing scheduling discipline.

Apart from the first two, these limitations could be overcome simply by preserving tasks with features such as priorities, scheduling that cannot be modeled by processor-sharing, parallel activities or second phases.

### 8.5 Future work

The simplification process could be extended in many ways.

It would be useful to support multiple classes of users, each with different workloads, different bottlenecks and different use of resources. If two classes of users are lightly coupled, it might be possible to simplify the model elements of each user class separately, keeping the shared part as it is. If two user classes are tightly coupled, the simplification might be done by fixing the ratio of throughput of the two classes.

The process can also be extended to handle cycles in the call graph. This might be achieved by keeping entries in cycles un-merged, each task having multiple entries. LQNS is able to solve such models since cycles of calls between entries are supported by the tool.

It would be interesting to preserve classes with very different service times to see how they affect the accuracy of the simplified models.
References


[29] Li, Y., Woodside, C. M., Complete Decomposition of Stochastic Petri Nets


