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

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

Practical Analysis of Software Bottlenecks

by

D. Peter Tregunno, B. Eng, P. Eng

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

Master of Engineering

Ottawa-Carleton Institute for Electrical and Computer Engineering

Faculty of Engineering

Department of Systems and Computer Engineering

Carleton University

Ottawa, Ontario, Canada, K1S 5B6

5th May 2003

©2003, D. Peter Tregunno

National Library
of Canada

Bibliothèque nationale
du Canada

Acquisitions and
Bibliographic Services

Acquisitions et
services bibliographiques

395 Wellington Street
Ottawa ON K1A 0N4
Canada

395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence

ISBN: 0-612-88980-7

Our file Notre référence

ISBN: 0-612-88980-7

The author has granted a non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of this thesis in microform, paper or electronic formats.

L'auteur a accordé une licence non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de cette thèse sous la forme de microfiche/film, de reproduction sur papier ou sur format électronique.

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

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

Canada

Abstract

Bottleneck analysis forms the core of computer system performance analysis. Many practitioners have been exposed to bottleneck analysis based on facts from product-form queueing networks. Layered computing systems introduce constructs that invalidate product-form bottleneck analysis. To overcome this problem, the concept of the software bottleneck was introduced. This thesis extends a number of existing definitions and methods for the detection, diagnosis, and mitigation of software bottlenecks. Specifically, this thesis presents 1) an algorithm for the identification of a software bottleneck, 2) updates to the definitions used to define a software bottleneck, 3) a set of software bottleneck mitigation strategies, 4) a simple approximation for task utilization in a layered queueing model, and 5) a case study to demonstrate the above, based on an industrial system.

Acknowledgments

The writing of a thesis cannot be performed in isolation. My thanks first go to my supervisor, Professor Murray Woodside, for his guidance and knowledge. Secondly, my thanks go to everyone who encouraged me while completing this endeavour, especially Sara Wallace for both moral and editorial support, and my family and friends for their interest. Finally, thanks go to my professional colleagues for their suggestions.

Contents

1	Introduction	1
1.1	Motivation	4
1.2	Contributions	5
1.3	Thesis Organization	6
2	Background	8
2.1	Software Performance Engineering	8
2.1.1	SPE and the Software Development Process	9
2.1.2	(Perceived) Problems with SPE	13
2.2	Interactions in a Distributed Computing System	15
2.2.1	Synchronous	16
2.2.2	Asynchronous	17
2.2.3	Forwarding	18
2.3	The Layered Queueing Network Model for a Computing System	20
2.3.1	Other Performance Modeling Techniques	25
2.4	Bottlenecks in a Distributed Computing System	26

2.4.1	Bottleneck Analysis in a Traditional Queueing Network . . .	27
2.4.2	Bottleneck Analysis in a Layered Queueing Network	29
2.4.3	Bottleneck Mitigation	30
3	Software Bottlenecks	34
3.1	Identification of a Software Bottleneck	35
3.2	Classification of a Software Bottleneck	37
3.2.1	The Processor Supported Software Bottleneck	38
3.3	Bottleneck Mitigation Strategies	41
3.3.1	Increasing Threading Levels	42
3.3.2	Addition of Processors	44
3.3.3	Reducing Processor Demands at the Bottleneck	45
3.3.4	Decreasing Interactions with Lower Layers	46
3.3.5	Putting the Strategies Together	47
3.4	Optimistic Bounds Analysis	49
3.4.1	The OBA Calculations	50
3.4.2	The Algorithm for the OBA	54
3.4.3	An Example	55
3.4.4	Test Results	58
3.4.5	Limitations of the Better Quick Approximation	61
3.5	Conclusions	62
4	Further Approximation Issues	63
4.1	Contention and the OBA	66

CONTENTS

vi

4.2	Errors due to the Multi-threading Approximation	70
4.3	The Impact of Multiple Entries per Task on the OBA	74
4.4	Errors due to Second Phases	76
4.5	The OBA and Multiple Reference Tasks	82
4.6	Systems Near Saturation	87
4.7	Summary of Recommendations	90
5	A Case Study	93
5.1	The Need for Performance Engineering	95
5.2	The Distributed Telephone Switch	96
5.2.1	Current Voice Switches	96
5.2.2	Evolution to the NGN Model	98
5.3	Layered Queueing Model of the System	101
5.4	Model Solution Results	106
5.4.1	Bottleneck Identification	109
5.4.2	Bottleneck Mitigation	110
5.5	Validation With the Analytic Solver	111
6	Conclusions	114
6.1	Contributions	114
6.2	Future Research	116
	Bibliography	117
A	LQN Input File for Case Study	124

List of Figures

2.1	An example synchronous interaction.	17
2.2	An example asynchronous interaction.	18
2.3	An example of a forwarding interaction.	19
2.4	Lifecycle of Three Entries.	23
2.5	An Example of a Three Layer Deep LQN.	24
2.6	The set of tasks Θ	31
3.1	An example system for the Case 2 Processor Supported Software Bottleneck discussion.	39
3.2	Model for the OBA Test	58
4.1	The Funnel, Tower and Pyramid Systems.	68
4.2	A graphical comparison of the data from Table 4.7.	78
4.3	A graphical comparison of the data from Table 4.8.	81
4.4	Model with Two Reference Tasks.	83
4.5	Feasible Throughput Region for the Two Reference Task Model.	85

4.6	Comparison of the utilization estimates for Task 2 as user think time increases.	89
5.1	An abstract view of the current telephone network.	97
5.2	A simplified architecture of a current telephone switch.	98
5.3	Evolution of the telephone network with the VoP “class 4 switch”	99
5.4	Simplified Architecture of the Call Connection Agent	100
5.5	Simplified Architecture of the Trunking Gateway Switch	102
5.6	Layered Queueing Model of the VoP Distributed Class 4 Switch. .	102
5.7	Message Sequence Chart for the VoP Distributed Class 4 Switch. .	104

List of Tables

3.1	The model parameters for the Case 2 Processor Supported Software Bottleneck discussion.	39
3.2	Task and Processor Utilizations for the Case 2 Processor Supported Software Bottleneck discussion.	40
3.3	Model Parameters for the Optimistic Bounds Analysis Test.	59
3.4	Comparison of the 1qns solution, the Optimistic Bounds Analysis and the Quick Approximation for the Base Case.	59
3.5	Comparison of the 1qns solution, the Optimistic Bounds Analysis and the Quick Approximation for the Non-homogeneous case.	60
3.6	Comparison of the 1qns solution, the Optimistic Bounds Analysis and the Quick Approximation for the Update case.	60
3.7	Comparison of the 1qns solution, the Optimistic Bounds Analysis and the Quick Approximation for the More Centralized case.	60
3.8	Bottleneck Strength for all test variations, from the 1qns solution, the Optimistic Bound Analysis and the Quick Approximation.	61
4.1	OBA and 1qns Utilization Estimates for the Tower System	67

LIST OF TABLES

x

4.2	OBA and lqns Utilization Estimates for the Pyramid System . . .	67
4.3	OBA and lqns Utilization Estimates for the Funnel System . . .	69
4.4	Increasing the Number of Threads at T_3	71
4.5	Multi-threading in the Tower - OBA Utilization Estimates.	72
4.6	Multi-threading in the Tower - lqns Utilization Estimates.	72
4.7	A comparison of OBA and lqns utilization estimates for entries 3, 4 and 5 of the "Update" case while shifting service time from phase 1 to phase 2.	77
4.8	A comparison of Corrected OBA and lqns utilization estimates for entries 3, 4 and 5 of the "Update" case while shifting service time from phase 1 to phase 2.	80
4.9	System Parameters for the Two Reference Task Model.	83
4.10	Utilization Estimates for the Two Reference Task Model.	85
4.11	System Parameters for the Modified Two Reference Task Example.	86
4.12	Utilization Estimates for the Modified Two Reference Task Model.	87
4.13	Estimates from the Corrected OBA compared to lqns as a task enters saturation.	89
4.14	Utilization estimates for Task 2 when user think time exceeds 200	90
5.1	Case study per-task utilization figures from the OBA and lqns.	107
5.2	Case study processor utilization figures from the OBA and lqns.	108
5.3	Selected Utilization Results from Step 1 of the Bottleneck Miti- gation, obtained from the OBA.	111

5.4	Selected Utilization Results from Step 2 of the Bottleneck Mitigation, obtained from the OBA.	112
5.5	Selected Utilization Results from Step 1 of the Bottleneck Mitigation, obtained from 1qns.	112
5.6	Selected Utilization Results from Step 2 of the Bottleneck Mitigation, obtained from 1qns.	113

Chapter 1

Introduction

One of the keys to unlocking performance gains in computing systems is the identification of bottlenecks. A bottleneck is a single contention point that limits overall system performance. Once a bottleneck is identified, appropriate mitigation actions can be taken to reduce or eliminate the impact of the bottleneck. These mitigation actions range from structural changes (i.e. to increase processing capacity), to behavioural changes (i.e. to reduce workloads).

It has been shown that when performance is considered early in a design cycle, there is a higher probability that the final system will meet its performance requirements, on time and on budget. The incorporation of performance considerations throughout a design cycle is a key principle of Software Performance Engineering (SPE) [1]. SPE relies on models of system structure and behavior to provide quantitative measures of overall system performance in the absence of actual measured data. As such, when SPE methods are employed early in a

design cycle, modeling techniques that provide “accurate enough” results, and more importantly, insight into the performance problems are essential. One of the important insights possible is the location of the system’s bottleneck(s).

Most practical modeling techniques useable in the early stages of system design involve the creation and solution of a queueing network. A queueing network is a natural fit for most software designs involving more than one process, as the processes are modeled as queueing network stations, and the messages passed between the processes are modeled as jobs.

For systems that exclusively use asynchronous message passing, a product-form queueing network forms a useful model. Solution of product-form queueing networks is a well understood subject, and techniques such as Asymptotic Bounds Analysis or Balanced Job Bounds Analysis provide insight into the system’s throughput and response time. To obtain insight into the location of the bottleneck, techniques such as Buzen’s operational analysis [2] are useful.

Client-server systems are one class of commercially important systems that are not accurately modeled by a product-form network. The use of a request-wait-reply pattern between a client and a server in these systems leads to simultaneous resource possession. Queueing network models have been developed to account for this phenomena, starting with Extended Queueing Networks [3], and including Layered Queueing Networks (LQNs) [4]. All queueing networks used to model simultaneous resource possession share one important trait: they are not product-form¹. As such, the approximation techniques mentioned above do not apply, and

¹Simultaneous resource possession violates the product-form rule that a job can only occupy

the solution of the network is considerably more involved.

In a system with simultaneous resource possession, bottlenecks can appear for a variety of reasons ranging from client blocking, to critical sections, to overloaded server tasks. Using model results alone, it can be difficult to determine the location of the bottleneck. Furthermore, armed with only throughput and utilization figures, it is difficult to develop insight into the causes of the the bottleneck (i.e. the root cause of the system's performance problems) and to develop effective mitigation strategies. Finally, early in the design cycle, there is often a reluctance on the part of designers to invest heavily in a model that may just be discarded after one solution. This is one of the reasons behind the push for development of simple techniques that deliver the required insight.

One helpful concept that is simple and delivers the desired insight is the Software Bottleneck [5]. Given a queueing network analytic solution, simulation results or an approximation, a simple calculation is performed, leading to a numeric "strength" value for each task in the system that is a potential bottleneck. This "Bottleneck Strength", along with the throughput and utilization results lead a designer to the "trouble spot", and aid in the selection of appropriate mitigations.

With a known requirement for such simple, insightful techniques, the following goals for this thesis are established:

1. The further development and clarification of the software bottleneck concept.

one service station at any instant. With simultaneous resource possession, a job can occupy any number of service stations at any instant.

2. The development of “zero-th order” approximation based mitigation strategies, with guidelines for their use.
3. The further development of a simple approximation method for task utilizations, with sufficient accuracy to correctly identify a system’s software bottleneck.

1.1 Motivation

The work of this thesis is motivated by the author’s experiences as a professional software engineer in a large development organization, with exposure to a number of projects that suffered delays due to performance problems. These problems seem to appear late in the design cycle, with a resulting chaos similar to that described by Smith in [1]. Other projects have had designers who tried to be more proactive and model the software under development with a product-form queueing network, even though the system’s behavior violated the product-form rules. The net result is that either type of project ends up taking on the “fix-it-later” performance mentality.

The author asked other experienced software designers why “fix-it-later” was part of the development process, and obtained three types of answers:

1. We do not need to worry about performance - we just need to write tight code.
2. We know that SPE might improve our chances of meeting our performance

requirements, but we need to meet the functional requirements first, and we are running out of time.

3. Performance modeling of our software is hard, does not seem to give us the results we need, and can only be done by experts who do not understand our software.

This informal poll mirrors the results of Klein [6], whose survey of the state of the practice of SPE documents the need for simple tools that provide insight into the performance problems.

1.2 Contributions

Consistent with the goals discussed above, the contributions of this thesis are as follows:

1. An algorithm for the identification of a software bottleneck. Previous work by Neilson et. al. [5] provides a useful set of definitions and examples, and an informal method for identifying a software bottleneck. This thesis provides such an algorithm for the identification of a software bottleneck in Chapter 3.
2. The incorporation of the forwarding interaction in the definitions used to define the software bottleneck. The updates are discussed in Chapter 3.

3. A set of “zero-th order” software bottleneck mitigation strategies. The contributions, detailed in Chapter 3, are the classification of software bottlenecks into two types, and a set of guidelines on when to use a particular mitigation strategy.
4. The extension of a simple method for the approximation of task utilization in a layered queueing model. We term the approximation detailed in Chapter 3 the Optimistic Bounds Analysis (OBA). This approximation is based upon, and extends, the approximation presented in [5] and the bounding techniques of [7].
5. A case study involving an industrial system is presented to demonstrate the utility of the above contributions. The case study is presented in Chapter 5.

1.3 Thesis Organization

This thesis is organized as follows: background material covering Software Performance Engineering, inter-task interactions in a distributed system, an outline of the Layered Queueing Network model, and bottleneck analysis methods are presented in Chapter 2. Chapter 3 presents an algorithm for the location of a software bottleneck, the software bottleneck mitigation strategies, and the approximation details. A discussion of issues arising from the approximation is located in Chapter 4. The case study is developed in Chapter 5; this case study is used to demonstrate the techniques developed in previous chapters. Concluding remarks

are presented in Chapter 6.

Chapter 2

Background

2.1 Software Performance Engineering

Performance has long been a concern in the design and implementation of computing systems. There have been numerous examples of software development projects that have been delayed or canceled due to systems not meeting performance requirements/objectives/goals. It is from this history that the discipline of software performance engineering emerged. Smith, in [1] provides the canonical definition for Software Performance Engineering (SPE):

SPE is the systematic process for planning and evaluating a new system's performance throughout the life-cycle of its development. Its goals are to enhance the responsiveness and usability of systems while preserving quality.

In the trade press ([8] and [9] are examples), there are numerous examples of “Performance Tuning”; generally regarded as the modification of an implemented system’s parameters to improve performance characteristics. While this is an interesting area, performance tuning is typically not the focus of SPE. Smith classifies tuning exercises as examples of the “fix-it-later” mentality, although an SPE study could provide sample parameters or rules to guide the tuning.

SPE relies on established performance modeling techniques to predict the characteristics of software designs. There is no one modeling technique that is favored; rather it is left to the discretion of the modeler to choose a technique that provides “quick” detection of performance problems. After a potential problem has been identified, SPE encourages the modeler to use a more accurate and/or detailed study to further quantify the problem, and objectively compare proposed mitigations or solutions for the performance problem.

Smith, among others, believes that all software development projects can benefit from the use of SPE techniques. Smith, in [1], gives examples of SPE application in on-line transaction processing and database domains; the author of this thesis as applied SPE techniques in the development of telecommunications switching equipment.

2.1.1 SPE and the Software Development Process

Whether an organization develops software with a waterfall style process, a spiral process or an evolutionary process, there are a number of discrete stages in each process. The difference amongst these processes is the ordering and the repetitions

of the stages. The stages in a software development process are often generically described as follows:

1. **Requirements Definition:** A system's services, constraints, objectives and goals are defined, often in conjunction with an end customer. The output of this stage is a "complete" set of concise functional requirements.
2. **System and Software Design:** The requirements from stage 1 are partitioned into hardware and software subsystems, and specifications for each of these subsystems are developed. The output of this stage is these subsystem specifications and an overall system architecture.
3. **Implementation and Unit Testing:** The software design of stage 2 is realized as a set of program units. These units are then tested in isolation to verify conformance to the stage 2 specifications.
4. **Integration and System Testing:** Software and hardware units produced in stage 3 are integrated, usually piece by piece. Once integration is complete, the entire system is tested against the stage 1 requirements.
5. **Operations and Maintenance:** Quite often the longest stage in the life cycle, this is where the system is put into service, problems detected and corrected, and changes/enhancements requested.

Traditionally, the functional requirements are focused on defining what the system does¹; often performance issues such as throughput, response time or capacity are

¹Smith gives some interesting history on how functional requirements became more important than performance requirements in [1].

buried in the functional requirements. Smith's view is that performance requirements should be expressed alongside functional requirements, thus giving them equal emphasis. To ensure that performance is given such an equal weight, Smith proposed a set of additional considerations for each stage of the life cycle; a selection follows:

1. Requirements Definition:

- What are the performance requirements/goals² for the given set of use cases?
- Can the functional requirements be achieved with an acceptable level of performance?
- Approximately how much computing power (i.e. computing cycles per second) is required to support the proposed system?

2. System and Software Design:

- Does the expected performance of this architecture fully meet requirements?
- Does the proposed computing configuration provide enough power? Excessive? Cost effective?

3. Implementation and Unit Testing:

- How do the performance metrics of the implementation alternatives compare?

- What are the resource requirements of the critical components?

4. Integration and System Testing:

- Are the performance requirements met?
- Have any unforeseen problems arisen?

5. Operations and Maintenance:

- What is the performance impact of a proposed change/enhancement?
- What are the long term configuration requirements/options to support future use?

Smith's SPE proposes that a set of models for the system are incrementally developed as a part of each stage. Without even a rudimentary model, providing answers to Smith's questions is next to impossible in stages 1 and 2, and difficult in stage 3. Furthermore, since most organizations incorporate a feedback mechanism in the life cycle, a model refinement in phase n can modify the answer to a question asked in phase $n - 1$. Often, due to increased system knowledge, such a feedback mechanism leads to a phase $n - 1$ requirements change. With organizational discipline, a "one stage" requirements change should not be too costly in terms of effort. However, a "two stage" or greater requirements change often has a large impact, even in the most mature organizations.

²The author has found that the use of "performance goal" is prevalent with those who specify software. Typically a performance goal causes the performance requirement to be relaxed (i.e. the goal is to meet the throughput requirement 99% of the time). In other cases, the goal is a future requirement (i.e. for Release 1.0, the requirement is a 10 msec latency, and the goal is a 5 msec latency. For Release 2.0, the requirement is a 5 msec latency).

One of the objectives of SPE is to arrive at stage 4 with an (almost) complete understanding of the system's characteristics. In contrast, developments not using an SPE approach (what Smith calls the "fix-it-later" approach) often first develop an initial understanding of the system's performance characteristics in stages 4 and 5. In most cases, the result is that the system does not meet the performance requirements. Developers then have to "tune" what they believe to be the causes of the performance problems. The performance improvements from this tuning are, at most, modest when compared to a design that considered performance from the beginning. The outcome of fix-it-later is often a combination of cost and time overruns, and performance requirement non-compliance.

While SPE is appropriate early in the design lifecycle, there is a need for guidance in stage 4 (Integration and System Testing). The bottleneck mitigation strategies developed in Chapter 3 are targeted towards this phase, especially for software designers in the fix-it-later crunch. However, the mitigation strategies can be used in any phase of a development. In general, the bottleneck location algorithm and optimistic bounds analysis can be used in any phase of a software development, as long as reasonable workload estimates are available.

2.1.2 (Perceived) Problems with SPE

A number of authors have described "experiences" with software performance engineering ([10][11][12] are examples). These reports detail the experiences of SPE specialists with industrial problems. The common thread is that the specialists themselves find SPE difficult! It is not surprising to find that the level

of adoption of SPE techniques by software designers who do not have advanced training in the field is very low. A frequent comment heard from designers is “I know performance is important, but I know that performance engineering is difficult”³.

In a 1996 survey [6], Klein investigated the reasons behind the low rate of adoption of SPE techniques. The key conclusion from this report is that, unlike most other engineering disciplines, software engineering does not have a “standard of practice” when it comes to performance.

Of interest in this thesis are the observations in the performance models and measures category. Klein describes these as “observations about the use of performance models and measures to predict performance, and how trends toward heterogeneous, distributed systems compromise the efficacy of existing methods”.

In the models and measures problem area, Klein observed that designers are looking for a modeling technique that abstracts the often complex underlying mathematics. Software designers want to model software components as software components, not as queueing network stations. Klein found that queueing networks were perceived as the only technique for modeling software, and that designers were not able to gain much insight from these models. Thus, the designer did not see much benefit from the modeling exercise. To increase the SPE adoption rate, Klein contends that “simple conceptual models” that provide insight into the behavior of the actual software are required.

³This is common response from an informal poll of the author’s professional colleagues when the topic of SPE is discussed.

Klein found that when distributed systems were modeled with queueing networks, the semantics of the queueing network (most often a product-form network) did not match the semantics of the actual system. This, of course, leads to a reduction in value of the model and results. When these model results are compared to the actual system, discrepancies arise. Klein observed that these discrepancies lead designers to discount SPE techniques. To overcome these problems, Klein suggests educating designers in advances in modeling techniques such as LQN.

Much research has been performed and is currently underway to address the issues that have kept SPE from being adopted. As an example, the LQN model and associated tools ([13][14] are but two examples) all combine to address these issues. It remains to be seen what effect this research will have on the adoption rate; CASE tools themselves have moved between the research lab and the designer's desk for the past 20 years, with most designers still not using them.

2.2 Interactions in a Distributed Computing System

There are a number of terms used to describe how distributed software components communicate. For the purposes of this thesis, communication between software components is called an interaction. Interactions themselves have a number of names, depending on the operating system and programming language being used. It is convenient to group interactions into three patterns: synchronous, asynchronous and forwarding. Andrews, in [15], provides a survey of the first two; the

forwarding interaction is not well discussed in the literature, even though it does have practical applications. The following sections provide background information for each of these patterns.

2.2.1 Synchronous

The synchronous interaction, also known as a remote procedure call (RPC) [16], forms the base for distributed client-server computing. This interaction pattern is supported by most modern programming languages, whether via a sequential language with an additional library (e.g. C++ with an RPC library), or via high level language that supports both parallel and sequential programming (e.g. Ada).

At the heart of the synchronous interaction is the concept of the “rendezvous”. A client requests a rendezvous with a server, and the server accepts the rendezvous. The client is blocked from the request time until the server completes the service phase; this is denoted “phase 1”. The server may have a subsequent autonomous phase, during which the server is not able to accept a rendezvous, but the client is not blocked. This service phase is denoted “phase 2”. An example of a synchronous interaction is given in Figure 2.1.

From a software perspective, the synchronous interaction has the following major advantages: simplicity and reliability. The synchronous interaction is simple to understand as it can be abstracted as a procedure call. It can be considered reliable since the client is unblocked only when the server has completed the service phase. However, the synchronous interaction has one major disadvantage, the client blocking. As we will see later, this is one of the sources of software

Figure 2.1: An example synchronous interaction.

bottlenecks.

A refinement of the synchronous interaction is the asynchronous RPC. We classify the asynchronous RPC as a synchronous interaction as the client waits for the server to accept the rendezvous. However, the client does not wait for the server to reply. The asynchronous RPC can be thought of as the interaction shown in Figure 2.1, where the server's phase 1 execution time is zero. With a zero length phase 1, the server generates a reply to the client immediately upon accepting the rendezvous, thus unblocking the client.

2.2.2 Asynchronous

The asynchronous interaction pattern is also provided by (or can be constructed with primitives of) most modern languages listed for the synchronous interaction.

The asynchronous interaction is also known as a “send-no-reply” interaction, as the client sends a message to a server with no expectation of a reply. The client does not assume that the server is ready to accept messages; the server often provides a finite queue for messages that arrive whilst the server is in use. A syn-

Figure 2.2: An example asynchronous interaction.

chronous interaction may be implemented as two asynchronous interactions, but most modeling techniques have difficulties with this, as loops may be introduced into the interaction graph. An example of an asynchronous interaction is given in Figure 2.2.

From a software perspective, the asynchronous interaction has the advantage that it does not introduce client blocking. However, since the client is not synchronized with the server, there is no guarantee that a message will reach the server for processing; a message queue could overflow, for example. Furthermore, there is no reply path, although this may not always be a disadvantage.

2.2.3 Forwarding

The forwarding interaction is similar to the synchronous case in that the client *A* sends a rendezvous request to a server *B* and waits for a reply. However, in the forwarding interaction, server *B*, the “forwarding” server, sends the rendezvous request onto a server *C*. Server *C* could forward this request on, or it could be

Figure 2.3: An example of a forwarding interaction.

the “replying” server. The replying server generates the reply to the initial client without the involvement of the forwarding servers. Note that the forwarding server does not block on the replying server’s response to the client. An example of a forwarding interaction is given in Figure 2.3.

From a software perspective, the forwarding interaction has all of the advantages of the synchronous interaction, but allows for service location abstraction. This is useful in systems that have middleware components (COBRA). As with the synchronous interaction, the major disadvantage is the client blocking.

2.3 The Layered Queueing Network Model for a Computing System

The Layered Queueing Network (LQN) model [4] originated as a performance model for multi-layered client-server type systems that communicate via rendezvous. The LQN model has been extended to allow the modeling of systems using all three of the interaction patterns discussed above. As well, there is support for defining detailed execution sequences as activity graphs [17] to aid in the specification of behavior.

An LQN is composed of three entities: tasks, entries, and processors.

Tasks: a task is an object which has a single thread of control and initiates and/or accepts service requests. There are three types of tasks:

- pure clients, which are tasks that only initiate service requests. Pure clients are analogous to customers in a queueing network.
- active servers, which are tasks that both initiate and accept service requests.
- pure servers, which are tasks that only accept service requests. Pure servers are analogous to servers in a queueing network, with the exception of service phases.

A task may be used to represent any resource within the system; typically a task models a software process or a hardware device.

In an LQN, the task is composed of one or more entries. The task has one common input queue for all incoming rendezvous requests for all of the task's entries. A rendezvous request is selected based upon the task's queue discipline (examples are: FIFO or priority). The destination entry is then invoked by the task to complete the rendezvous.

Entries: an entry is an endpoint, within a task, of a rendezvous. Each entry has its own execution parameters. An entry is analogous to a C++ method (where a task is analogous to a C++ object). The entry contains the major structural (interaction is amongst entries) and behavioural parameters (service demands) of the model.

Processors: the processor is an artificial concept in an LQN. A processor is modeled as a pure server task where the requests for service come from the tasks assigned to the processor. An entry, during invocation, requests processor service one "slice" at a time; each request for service is implicitly modeled as a rendezvous between the entry and the processor's "pseudo-entry". The transformation of processors into tasks and pseudo-entries is fully described in [4].

An entry e is characterized by its phases of execution, the number of which is denoted by P_e . To simplify calculations, an entry can have at most $P_e = 3$ phases of execution, but for many models, $P_e = 2$ is sufficient. Figure 2.4 shows the lifecycle of a set of entries. In this figure, we see that during phase 1, Entry A conducts a rendezvous with Entry B. Note that Entry A is blocked until the

rendezvous with Entry B completes. Entry B is a two phased entry, conducting a rendezvous with Entry C, during its second phase.

The following definitions from [4] for the parameterization of an entry are repeated here for completeness:

- Phase (e, p) denotes ‘phase p of entry e ’
- s_{ep} denotes the mean total execution time of (e, p)
- y_{edp} denotes the mean number of requests to entry d , during (e, p)
- x_{ep} denotes the mean duration of (e, p) , also referred to as the service time of (e, p) . This mean duration is the given by $x_{ep} = s_{ep} + \sum_{d \in \epsilon^*} y_{edp}(t_{ed} + \delta_{ed})$, where t_{ed} is the mean delay to e when it sends a rendezvous request to d , ϵ^* is the set of all entries, and δ_{ed} is mean “delay”⁴ in communicating the rendezvous requests from e to d .

The modeler provides s_{ep} , y_{edp} and δ_{ed} , the solution to the underlying queueing network provides the t_{ed} ’s.

Figure 2.5 gives a graphical example of a three layer LQN. Here we see that tasks T_1 , T_2 and T_3 each have one entry, denoted as E_1 , E_2 , and E_3 , and task T_4 has two entries, E_4 and E_5 . For the purpose of the example, the processors are shown, and that tasks T_2 and T_3 share processor P_2 . For the diagram, we see that

⁴This mean communications delay assumes that the delay from e to d is dominated by factors outside of the software being modeled, such as physical propagation delays in a communications network.

Figure 2.4: Lifecycle of Three Entries.

Figure 2.5: An Example of a Three Layer Deep LQN.

synchronous interactions are used from entry E_1 to E_2 , E_1 to E_3 , and E_2 to E_4 . An asynchronous interaction is used between E_3 and E_5 .

Task T_1 is considered a pure client task, or a “reference” task. All load for the model comes directly from this task’s entry E_1 . Tasks T_2 and T_3 are active servers, as their entries both accept and generate interactions. Finally, task T_4 is considered a pure server task, as it’s entries only accept interactions.

The solution of LQN models is supported by both a queueing network simu-

lator, *parasrvn*, and an analytic solver *lqns*[18]. In most cases, the analytic solver is many times quicker than the simulator, and as such is the preferred model solution method. It has been shown that results from the simulator are reasonably close to actual measurements, given accurate entry characterization [18]. Furthermore, the performance metrics from the analytic solver are considered accurate enough; error bounds are typically 10%. However, there are many documented difficulties with the analytic solver and models that are highly saturated (significantly more offered load than the system can handle). In these cases, numeric convergence is the main problem. A large amount of effort has been put into solving these convergence problems [17]. Throughout this thesis, most models results are obtained through *lqns*, with any difficulties noted.

2.3.1 Other Performance Modeling Techniques

There exist a number of techniques other than layered queueing for the analysis of software performance models. Balsamo, in [19], gives an overview of the three major classes of stochastic modeling techniques: Queueing Networks (of which LQN is a member), Stochastic Timed Petri Nets and Stochastic Process Algebras. Further references to these techniques can be found within [19].

Stochastic Process Algebra (SPA) based modeling techniques rely on the specification of system functionality through a Process Algebra. Process Algebras are a set of techniques used to provide a formal specification of communicating entities that use synchronization to communicate. SPAs attach performance related information (e.g. service time) to this formal specification. Techniques exist to

transform the result into a Markov chain. This Markov chain can then be solved to yield performance metrics. According to [19], drawbacks of SPA techniques include the difficulty of Markov chain solution (due to state explosion), and the requirement for a formal specification. Advantages cited revolve around the creation of a formal specification.

Stochastic Timed Petri Nets (STPN) are a direct extension of Petri Nets. Petri Nets can be used as a formal method for the specification and verification of software. A Petri Net can be extended to a STPN with the addition of temporal information to the Petri Net places and transitions. With this information in place, the STPN can be transformed into a Markov chain; the Markov chain is solved to yield performance metrics. According to [19], advantages of Petri Net techniques include their close relationship with state machine models; techniques exist to transform UML statecharts into Petri Nets. Disadvantages cited for Petri Net techniques include the state explosion seen in large Markov chain solution.

2.4 Bottlenecks in a Distributed Computing System

One of the keys to performance improvement is the reduction or removal of bottlenecks from a system. Jain [20], for one, suggests that the identification of a bottleneck should be the first step in any performance improvement study. In a computing system, a bottleneck is a resource that is the key limiting factor in improving system performance.

In this section, we review methods for the identification of bottlenecks, first

for “traditional” queueing networks, and then for LQNs. We then discuss findings on the reduction or removal of bottlenecks.

2.4.1 Bottleneck Analysis in a Traditional Queueing Network

For the purposes of this section, we define a traditional queueing network as an interconnected network of queues, where jobs travel from queue to queue; a job can only be in one queue at a given instant, giving a product-form network. If one uses such a queueing network to create a performance model for a computing system, as is suggested by in [21], [1], or [20], there are a number of simple calculations that lead to the identification of a bottleneck device.

Jain’s “Operational Laws”[20]⁵ are a set of relationships for single class queueing networks that require no assumptions about the distribution of service times nor the inter-arrival time, and form the basis of these simple calculations. We now present the result required for bottleneck analysis without proof.

Forced Flow Law: The utilization of the i th device $U_i = XD_i$, where $D_i = V_i S_i$ is the total service demand on the i th device for all visits V of a job with service time S , and X is the system throughput. The important consequence, for the current discussion, of the forced flow law is that device utilization is proportional to service demand, such that $U_i \propto D_i$.

Using the forced flow law, we see that the device with the highest total service demand D_i will also have the highest device utilization U_i . Saturation is defined

⁵Jain states that these relationships were first identified by Buzen[2], and extended by Denning and Buzen[22].

as device i having $U_i = 1$. To identify the bottleneck, we must compute D_i for each device in the system. Let us assume that $D_j = \max\{D_1, D_2, \dots, D_j, \dots, D_M\}$; device j is the bottleneck. The system throughput is then bounded by

$$X(N) \leq \min\left\{\frac{1}{D_j}, \frac{N}{D+Z}\right\} \quad (2.1)$$

and the response time is bounded by

$$R(N) \leq \min\{D, ND_j - Z\} \quad (2.2)$$

where $D = \sum D_i$ is the total service demands across all devices except users, N is the number of users, and Z is the user “think time”.

These bounds have proven to be highly useful in practice, as they identify the bottleneck without the solution of a queueing network. However, their utility is reduced for the following reasons:

- The assumption of a single class of jobs at all stations. Multi-class bounding techniques put forward by Serazzi and Balbo ([23] and [24]), and Majumdar et. al. ([25]) remove this restriction.
- The implied lack of support for simultaneous resource possession. Simultaneous resource possession enables more realistic models of distributed computing systems. We discuss this further in Section 2.4.2.

2.4.2 Bottleneck Analysis in a Layered Queueing Network

One of the principle advantages of the synchronous interaction is its simplicity when compared to a set of asynchronous interactions. However, the simplicity comes with a price; while the client is waiting for a response from the server, there is a loss of potential concurrency. If a task spends too much time in this waiting state, tasks in higher layers see this waiting task as a bottleneck. In [5], Neilson et. al. term such a bottleneck a “software bottleneck”. Smith’s “One Lane Bridge” software antipattern [26] can also be considered a match for this phenomena.

The analysis presented in Section 2.4.1 is not able to detect the waiting task as a bottleneck. This is due to the lack of support in the analysis for simultaneous resource possession. The layered queueing model does exhibit this simultaneous resource possession; in the simple case, the active server task is holding its server as well as blocking rendezvous for other clients. The second element at play in a software bottleneck is blocking. In a traditional queueing network, blocking occurs when there is a lack of buffer space. In a layer queueing network, blocking occurs when a client requests service from a server that is currently busy.

To aid in the analysis, Neilson [5] provides two observations and two definitions:

Observation 1: When a task is a software bottleneck, it is saturated but its servers are not. Note that Neilson considers a device with a utilization of 1 to be saturated.

Observation 2: A saturated task or processor tends to saturate (or block) tasks which use it by rendezvous, so bottlenecks tend to spread saturation in a rendezvous net back along the request arcs.

Definition 1: A software bottleneck occurs when a task (or a set of task threads) exhibits a high utilization which is also high relative to the utilization of each of its servers, either direct or indirect.

Definition 2: A measure for the bottleneck, called the bottleneck strength, is the ratio of its utilization to that of its most highly utilized server, direct or indirect.

From the last definition, we obtain the following formula for the calculation of the bottleneck strength:

$$B_b = \frac{U_b}{\max_{i \in \Theta}(U_i)}. \quad (2.3)$$

We have denoted by b the candidate bottleneck task, and Θ the set of all tasks that are direct or indirect servers of b . This set Θ is depicted in the LQN fragment shown in Figure 2.6.

2.4.3 Bottleneck Mitigation

One of the principle results from [5] was the investigation into the utility of “task cloning” as a means of bottleneck mitigation. The authors note that they may not have been the first to use task cloning to provide bottleneck relief; however, they also did not find much formal analysis in the literature. Task cloning, or multi-

Figure 2.6: The set of tasks Θ .

threading⁶, is a purely software solution that increases the number of servers available. Multithreading may be directly provided by an operating system via kernel-level or user-level threads, or by the forking of new tasks. In reality, threads may be created (tasks forked) as required to fulfill a service request, or they may be statically created and assigned to fulfill a service request. The analysis assumes a finite number of statically assigned threads. This is reasonable since most operating systems provide a cap on the number of concurrent threads (or tasks).

The results from increasing the threading levels at the bottleneck task were positive. With the addition of a number of threads (the optimum number is found by experiment) the user level throughput was increased. As well, the bottleneck strength was reduced. In some cases, the addition of threads led to processor

⁶Several practical methods for task cloning exist; examples are light-weight threads, task pools or transaction control blocks.

saturation at the bottleneck. In summary, the results show that multithreading does mitigate the effects of a software bottleneck, albeit with non-obvious side effects.

Smith, in [26], proposes (with little analysis) bottleneck mitigation through the reduction of service time at the bottleneck, or the re-routing of load around the bottleneck.

Serazzi et. al., in [27], looked at the impact of a software bottleneck on the scalability of a distributed system. Their definition of a software bottleneck is slightly different from the one given above; they define a software bottleneck as a server that has reached its utilization limit. The utilization limit is a constraint that the modeler places on a particular resource; for a software server the utilization limit is allowed to be greater than 1. Serazzi's team proposes the use of iterations of asymptotic bounding techniques to obtain utilization estimates, then a reconfiguration step for resources that exceed their utilization limits. The reconfiguration options considered are: replication or reallocation of capsules (the LQN equivalent is a task), clusters (there is no direct LQN equivalent; a cluster is a logical grouping of objects), or objects (the LQN equivalent of a single entry), or the partitioning of the work performed by a capsule, cluster, or object. The major drawback of this technique is that queueing delays are only considered once a system configuration that meets all of the constraints have been found. Furthermore, no guidance is given with respect to the applicability of a given reconfiguration option to a bottleneck.

Litoiu et. al., in [28], continue the work of [27], in the investigation of the

maximum number of useful threads in the mitigation of a software bottleneck. Operating with the same definition for a software bottleneck as before, this team uses nonlinear (and linear) programming techniques coupled with analytic modeling software to determine the maximum number of threads. This removes one of the drawbacks of the previous work; queueing delays are now taken into account in the search steps. An important point, implicit in this work, is that mitigation of a software bottleneck does not always lead to its removal. If the system performance is acceptable with a bottleneck, the bottleneck will remain. It is interesting to note that, for the examples given by Litoiu, the simple “threading rule”⁷ of Woodside[29] obtains similar results (useful threading levels) without the solution of a performance model.

⁷Woodside’s simple threading rule states that a useful number of threads at a particular server is the sum of the number of threads in the server’s servers plus one.

Chapter 3

Software Bottlenecks

As was noted in Section 2.4.1, an effective strategies for performance improvement is the discovery and reduction of bottlenecks. We noted that, for a traditional queueing network model, the identification of a bottleneck can be done with a simple set of calculations, detailed in Equations 2.1 and 2.2. Unfortunately, bottleneck detection in a model that contains simultaneous resource possession is not so easy. Furthermore, as noted above, active server entities in such a model can exhibit two different types of bottlenecks, hardware (device) bottlenecks, or software (logical) bottlenecks. This chapter further explores the following areas:

- Bottleneck detection methods
- Categorization of software bottlenecks
- A measure for potential improvement
- Bottleneck reduction or elimination methods

- An approximation of task utilization that is useful in the detection of a software bottleneck.

3.1 Identification of a Software Bottleneck

As was discussed in Section 2.4.2, in [5], Neilson et. al. provided a set of definitions useful in characterizing the software bottleneck phenomena. A procedure for software bottleneck detection was informally defined through a number of examples. We now present a procedure for software bottleneck detection, as follows:

1. Obtain utilization estimates for every task and processor in the system. This can be done via measurement, simulation, analytic solution of the model, or an approximation technique. One such approximation technique that has produced reasonable results is given in Section 3.4.
2. Locate each task that is saturated. A task deemed saturated has a utilization greater than a predefined threshold, U_{sat} . The selection of U_{sat} at the discretion of the modeler. One typical value for U_{sat} is 75%. Denote this set of tasks by Φ . If Φ is an empty set, there are no software bottlenecks in the system for the given input load and the chosen U_{sat} .
3. Remove all tasks from Φ that are pure servers. From the definition of a software bottleneck, a pure server cannot be software bottleneck since it does not hold more than one resource¹.

4. For each task b that remains in Φ , compute its bottleneck strength B_b using equation 2.3. The task with the largest B_b is the software bottleneck.

When [5] was written, the underlying LQN model did not include the concept of a forwarding interaction. Forwarding interactions are now supported by the LQN model, and do appear in a number of systems. We now examine the bottleneck strength equation, considering the addition of forwarding interactions to the model.

Equation 2.3 defines the bottleneck strength B_b to be the utilization ratio between task b and the server of task b with the highest utilization. The bottleneck strength gives us an estimate of the maximum performance gain achievable if the cause of the bottleneck is eliminated.

Recall that the set of servers for task b is defined as Θ . If a task b performs synchronous interactions with tasks c , d , and e , the server set Θ is $\{c, d, e, P_b\}$ where P_b is the processor for task b . If we assume that there is enough offered load at b , the performance gains from bottleneck elimination are limited by the server in Θ which saturates first. The bottleneck strength measure assumes that the server which saturates first is the server in Θ that has the highest utilization.

In the case of a forwarding interaction, the set of servers Θ is expanded to include both the forwarding servers and the replying server. We include the forwarding servers due to the fact that the client task b waits for each forwarding server to accept the interaction. Thus a forwarding server with a high utilization,

¹These tasks are important in the overall performance analysis, as they can prove to be “traditional” bottlenecks.

relative to the other members of Θ , will limit the maximum performance gain from bottleneck elimination. In most practical cases, we have found that forwarding servers have low execution times. However, the forwarding server does block until the rendezvous is accepted at the next server, causing the forwarding server's task utilization to be greater than the service time suggests.

3.2 Classification of a Software Bottleneck

Through experiments, we have found two variations on the software bottleneck, a processor supported software bottleneck, and a server supported software bottleneck. The two classes of software bottlenecks are separated due to the differences in mitigation strategies for each class, as described in Section 3.3.

With the bottleneck detection method outlined above using the bottleneck strength measure of equation 2.3, the difference between the two is clear. We denote by θ_b , the member of server list Θ that has the highest utilization. A processor supported software bottleneck has θ_b being the processor hosting the task b . The server supported software bottleneck has task θ_b being a server of b .

The notion of two classes of software bottlenecks appears to be new; in [5], Neilson et. al. present a number of examples of what we are calling server supported software bottlenecks. One processor supported software bottleneck does appear in the multi-threading cases, however it is not identified as being different from the rest of the software bottlenecks. We refer the reader to [5] for a number of examples of the server supported software bottleneck.

3.2.1 The Processor Supported Software Bottleneck

The processor supported software bottleneck relies on the utilization of task b 's processor being less than the utilization of task b , but greater than the utilization of all other members of Θ . A high processor utilization indicates that task b either has a large service demand, or other tasks are making use of this processor. In many practical cases, this high processor utilization leads to longer blocking times for tasks that are clients of b , and as such, these client tasks could become software bottlenecks.

This being said, we have found two common design situations where task b becomes a processor supported software bottleneck:

1. High levels of threading at task b , with lightly loaded client of task b .
2. More than one processor-intensive task sharing the same processor, with the bottleneck task having a low number of synchronous interactions with lightly loaded servers.

Case 1 appears as the result of the use of threading for bottleneck mitigation, as discussed below. When a processor supported software bottleneck appears due to the increasing of the threading levels, it may indicate the need for another mitigation technique.

An example of case 2 is shown in Figure 3.1, with the model parameters given in Table 3.1. Here we have a system where T_3 and T_4 share a processor, P_3 . To ensure that T_1 or T_2 do not become software bottlenecks, the design of T_3 and T_4

Figure 3.1: An example system for the Case 2 Processor Supported Software Bottleneck discussion.

Parameter	Value	Parameter	Value
M_{users}	10	y_{24}	1
Z_{users}	15	s_{31}	0.25
$y_{users,1}$	1	s_{32}	1
$y_{users,2}$	1	s_{41}	0.25
s_{11}	0.25	s_{42}	1
s_{12}	0.75	y_{35}	1
s_{21}	0.5	y_{46}	1
s_{22}	0.5	s_{52}	0.25
y_{13}	1	s_{62}	0.25

Table 3.1: The model parameters for the Case 2 Processor Supported Software Bottleneck discussion.

Processor	Utilization	Task	Utilization
1	0.292	1	0.829
2	0.292	2	0.816
3	0.729	3	0.779
5	0.0729	4	0.779
6	0.0729	5	0.0729
		6	0.0729

Table 3.2: Task and Processor Utilizations for the Case 2 Processor Supported Software Bottleneck discussion.

moved a large amount of the processing into the second phase. This model was solved, yielding the utilization shown in Table 3.2. With U_{sat} set to 0.70, the bottleneck strengths for the tasks were found to be:

$$B_1 = \frac{0.829}{0.779} = 1.06 \quad B_2 = \frac{0.816}{0.779} = 1.05 \quad B_3 = \frac{0.779}{0.729} = 1.07 \quad B_4 = \frac{0.779}{0.729} = 1.07$$

Numerically, tasks 3 and 4 are the software bottlenecks, and processor supported. However, they are quite weak!

Making some slight changes to the model (setting $y_{351} = 3$, $y_{352} = 1$, $y_{461} = 3$, $y_{462} = 1$, and $s_{51} = 0.1$, $s_{52} = 0.5$, $s_{61} = 0$, $s_{62} = 0.6$) to increase the load on tasks 5 and 6 yields the following bottleneck strengths:

$$B_1 = \frac{0.818}{0.744} = 1.10 \quad B_2 = \frac{0.780}{0.720} = 1.08 \quad B_3 = \frac{0.744}{0.545} = 1.37 \quad B_4 = \frac{0.720}{0.545} = 1.32$$

Here we see that task 3 is a slightly stronger bottleneck, and is still processor supported.

We have found that processor supported software bottlenecks tend to be quite

weak, most often having bottleneck strength values below 1.5. Typically, the strength level will increase if the task in question is the client in a large number of synchronous interactions. However, this task will “push back” on its client tasks, often causing them to saturate as well, reducing throughput. One particularly nasty case is where the client of the bottleneck task uses asynchronous interactions with the bottleneck task, and the bottleneck task is the client in a number of synchronous interactions. This can cause the bottlenecked task to become saturated without any “push back” towards the client. If the bottlenecked task spends a large percentage of time blocked in synchronous interactions, the processor utilization will be low, and the bottleneck strength will be correspondingly high.

3.3 Bottleneck Mitigation Strategies

Let us assume that the system throughput levels do not meet requirements, and that we have found task b to be the software bottleneck. From the discussion in Section 2.4.2, we know that throughput is directly limited by the software bottleneck. Thus, reduction of the software bottleneck strength, B_b , or complete elimination of the software bottleneck, will lead to an increase in throughput. Mathematically, reduction of the bottleneck strength is achieved by either increasing U_b or decreasing U_{θ_b} .

In this section, we consider four mitigation strategies:

1. increasing threading levels at the bottleneck task b ,
2. replication of the bottleneck task and its processor,

3. reduction of the bottleneck task's phase 1 processor demands, and
4. decreasing the number of interactions that the bottleneck task has with lower layers.

For each of these mitigation strategies, we provide an estimate of the reduction of the bottleneck strength, based on values obtained from model solution, estimation techniques, or system measurement.

It is important to note that each of these strategies may cause a different task to become the software bottleneck. In general, without experimentation (either using a model or a live system), it is quite difficult to determine what the exact impact of a change is on the system. This is the reason why the mitigation strategy selection process discussed below is a single step process; repetition of selection of a strategy and experimentation, until requirements are met.

In the following sections, we use the adjectives strong and weak to describe a software bottleneck. Quantifying strong and weak is somewhat arbitrary; we say a software bottleneck is strong when the bottleneck strength $B_b > 2$, and a software bottleneck is weak when $1 < B_b < 2$.

3.3.1 Increasing Threading Levels

As discussed in Section 2.4.3, it has been shown that the increasing of a bottleneck task's threading levels reduces the bottleneck strength B_b . In a first approximation, for a strong software bottleneck with sufficient client load, the potential throughput gain at task b has an upper limit of $f_b \times B_b$, where f_b is the throughput at task

b before the increase of task b 's threading levels. This result is documented in [5].

We now consider the impact of an increase in threading levels on the bottleneck strength measure. The objective of this mitigation strategy is to increase the utilization of members of the server set Θ , without a large increase in task b 's utilization. For a strong server supported software bottleneck, with enough client load (i.e. all threads are busy), the per-thread utilization at b remains approximately U_b . Additional waiting time introduced by thread contention for the processor of task b , which would increase U_b , is not considered in this first approximation.

In an attempt to have b not remain a software bottleneck, we must keep $U_{\theta_b} > U_b$. This should be possible since $U_{\theta_b} < 2 \times U_{sat}$ for a strong software bottleneck. However, it is possible that the increase in threading levels at b causes θ_b to become a software bottleneck.

The cost of increasing the threading levels at a bottleneck task is often minimal, given a modern operating system. There may be some overhead introduced by "dispatching" software that accepts the rendezvous and forwards it to an idle or newly created thread. The drawback of increasing threading levels is the corresponding increase in processor contention levels.

Increasing threading levels at the bottleneck task is useful for all strong software bottlenecks. For a strong server supported software bottleneck, there is a large amount of unused processing capacity in the servers below the bottleneck, and an even larger amount of unused processing capacity at the bottleneck task's processor. In the case of a strong processor supported software bottleneck, there

is a large amount of unused processing capacity at the task's processor.

A weak software bottleneck indicates that there is not enough unused processing capacity at θ_b to double U_{θ_b} without potentially making θ_b a software bottleneck. It is for this reason that we recommend the increasing of a weak server supported bottleneck task's threading levels with care. In the case of a weak processor supported software bottleneck, the increasing of threading levels has a neutral to negative impact on performance. The negative impact is due to increases of per-thread utilization due to processor contention.

3.3.2 Addition of Processors

In the case of a weak processor supported software bottleneck, one often finds that the bottleneck task b is blocked waiting for processor time. This is can be the case when task b is sharing a processor with other processor intensive tasks of equal or higher priority. The addition of processing resources can alleviate this problem; however the impact is not often dramatic.

In the first approximation, assume that the addition of n processors reduces processor contention to $\frac{1}{n}$ the original value. If we let p be the percentage of the total service time x_b that task b spent blocked due to processor contention, then the maximum decrease in total service time is $\frac{p}{n}$. This gives an estimated new total service time $x'_b = x_b - \frac{(n-1)p}{n}$, or a decrease of $\frac{(n-1)p}{xn}$ in total service time. Since the task utilization is proportional to x_b , we could at most expect the utilization to decrease by the same factor. To ensure that b is no longer a software bottleneck, the task utilization has to decrease below U_{sat} , since we are not causing

the processor utilization to increase.

In reality, the addition of processors does not cause the utilization to decrease in a linear fashion as described in the previous paragraph, and as such, the impact is not as great as suggested. Thus the addition of processor for the bottleneck task is recommended only for weak processor supported software bottlenecks. Furthermore, the addition of processors often is expensive and involves sophisticated software modifications.

3.3.3 Reducing Processor Demands at the Bottleneck

One of the most often suggested solutions to software performance problems is “write faster code”. This seems like a sensible solution, but when faced with a strong server supported software bottleneck, the resulting performance improvements are often less than spectacular.

In the case of a strong server supported software bottleneck, the service time of task b is dominated by waiting at lower layer servers (since $U_{P_b} < U_{\theta_b} < 2U_b$). Let s be the percentage of the total service time x of task b that is processor demand. Let n represent the percent reduction in s that is proposed. We then have a new total service time $x' = x - ns$. In the first approximation, this gives a reduction of $\frac{ns}{x}$ in the total service time. Again, using the fact that the utilization of task b is proportional to its service time, we expect the same reduction in utilization.

To ensure that task b is no longer a software bottleneck, $\frac{ns}{x}U_b < U_{sat}$, or $\frac{ns}{x} > \frac{U_{sat}}{U_b}$. In the case of a weak processor supported software bottleneck, this might be achievable, since s is a large percentage of x . Thus an n percent reduction

in s is almost an n percent reduction in x (and U_b). However, in the case of a server supported software bottleneck, the total service time x is dominated by waiting, and an n percent reduction in s results in a negligible reduction in x . This is the reason for our recommendation that the reduction of phase 1 demands is only useful as the primary method for software bottleneck removal when a weak processor supported software bottleneck is found.

Furthermore, our experience has shown us that service demand reductions of more than 10% are difficult to realize. Modern development tools already perform a large number of optimizations on the code. The only practical ways to drastically reduce service demand are to either use a more efficient algorithm or to discover a bug².

3.3.4 Decreasing Interactions with Lower Layers

The final method for bottleneck mitigation that we are considering addresses one of the root causes of a software bottleneck; excessive waiting for a resource. If a task b has 2 or more rendezvous with a lower layer task d , it should be possible to “batch” those rendezvous together, and in the process b incurs only one “wait” for service at d . If the number of rendezvous is large, and the waiting time great, a large decrease in the utilization of b should be possible.

For this first approximation, let us consider a single entry task b . The phase 1 service time of b is given by $x_{b1} \approx s_1 + \sum_{d \in \mathcal{E}^*} y_{bd1} t_{b1d}$, where $\mathcal{E}$ is the set of tasks

²A number of the author’s colleagues believe that correcting the misuse of object-oriented methods is another effective method to reduce service demands.

with which b interacts. The term y_{bd1} represents the number of rendezvous between b and d ; t_{bd1} represents the waiting time that b experiences for a rendezvous between b and d . The t_{bd1} term includes the waiting time for the rendezvous and the service time of d ($t_{bd1} = rd_d + x_{d1}$). For simplicity, let us consider only one task d that either has a large delay for a rendezvous, and/or is a server for 2 or more rendezvous from b . If the rendezvous requests from b to d are batched together, the $y_{bd1}t_{bd1}$ term from b_1 becomes $rd_d + y_{bd1}x_{d1}$. This is a conservative approximation; it does not take any optimizations that the server could make based on the batching. The new estimate of x_{b1} is given by $x'_{b1} = x_{b1} - (y_{bd1} - 1)rd_d$, leading to a decrease of $\frac{(y_{bd1}-1)rd_d}{x_{b1}}$ in the service time and utilization of b .

To ensure that b is no longer a software bottleneck, $\frac{(y_{bd1}-1)rd_d}{x_{b1}} U_b < U_{sat}$. In practical terms, this means that the product of y_{bd1} and rd_d must have been a large portion of x_{b1} . Large rendezvous delays are most often seen when a server has a high utilization. If b is a software bottleneck, and a server of b has a high utilization, the bottleneck is weak. Thus, batching rendezvous together is only effective when b is a weak server supported software bottleneck, and the product of y_{bd1} and rd_d is a large percentage of the service time of b .

3.3.5 Putting the Strategies Together

The following algorithm gives a systematic method for the removal of a software bottleneck.

1. Obtain performance metrics for the system, and locate the software bottle-

neck.

2. Determine the type of software bottleneck. The four choices are: strong server supported, weak server supported, strong processor supported, and weak processor supported.
 - (a) If the bottleneck is strong, increase the threading levels at the bottleneck task as per Section 3.3.1. Goto step 3.
 - (b) If the bottleneck is weak and processor supported, and there is a large amount of processor contention, increase the number of processing resources the bottleneck task has access to as per Section 3.3.2. Goto step 3.
 - (c) If the bottleneck is weak server supported, and there are a number of rendezvous with a server that has a large rendezvous delay, try batching rendezvous requests as per Section 3.3.4. Goto step 3.
 - (d) If the bottleneck is weak processor supported, try reducing the phase 1 service demands of the bottleneck task as per Section 3.3.3. Goto step 3.
3. Re-evaluate the system's performance. If not satisfactory, go back to step 1 with the current performance metrics.

3.4 Optimistic Bounds Analysis

Quantitative software performance engineering relies on either the solution of the mathematical relationships underling the model, or the simulation of the system being modelled. Currently, there exists both an analytic solver and simulation toolkit for LQNs. The analysis presented in this section provides a lightweight, quick approximation of a layered model's performance; it is meant to be used as a "back of the envelope" calculation method.

As we have discussed in Section 2.1.2, SPE will have a higher likelihood of being performed by practitioners if simple, quick, and accurate approximation techniques exist. Klein, in [6], emphasizes the need for simple techniques that deliver insight into potential performance problems. As has been mentioned before, such simple techniques exist for systems without simultaneous resource possession (see Section 2.4.1 for one such method), but simple, accurate methods for realistic distributed systems are few and far between.

In this section, we provide a bounding calculation for layered models. As was discussed in Chapter 2, performance metrics can be obtained from a layered model through simulation or analytic techniques. However, for the purposes of bottleneck identification, a simplified analysis is often sufficient. The calculation presented in this section, called the Optimistic Bounds Analysis (OBA), simplifies the layered model solution by ignoring waiting time when an entry conducts a rendezvous with another entry.

The OBA is not entirely new; Majumdar et. al. presented an upper and lower

bound analysis for layered models in [7], Woodside et. al. compute an upper bound on throughput in [4], and Neilson et. al. perform a similar analysis in [5], applying the results to software bottleneck discovery. Where the OBA differs from these is:

- in the treatment of tasks with multiple entries,
- tasks with a multiplicity (threading level) greater than one,
- and processors with multiple tasks.

The upper bound calculation of [7] only considers single entry tasks, but does permit multiple tasks per processor. The “type one” bounds analysis from [4] does consider multiple entry tasks, but does not permit a task multiplicity greater than one. The quick approximation presented in [5] considers single entry tasks with a multiplicity equal to one, and with all service time in the first phase. The OBA removes all of these restrictions. Furthermore, the quick approximation constrains the model to one symmetric set of reference tasks. The general OBA calculation removes this constraint.

The current OBA does not take interactions other than rendezvous into account, nor does it provide support for the interactions described by precedence expressions.

3.4.1 The OBA Calculations

For the OBA, we consider the following entities from the LQN model:

- A set of tasks T_i , where $i = 1 \dots T$. We assume that the first R tasks are the reference tasks T_r , where $r = 1 \dots R$.
- Processors are modeled according to the “processor transformation”, as discussed in Chapter 2. The artificial tasks that arise from this transformation are denoted by T_t , where $t = T + 1 \dots P$, where P is the number of processors in the model.
- A set of entries E_e , where $e = 1 \dots E$. Entries $E_1 \dots E_R$ belong to the reference tasks $T_1 \dots T_R$, which by definition are single entry tasks. All of the other tasks have one or more entries; the set of entries $\mathcal{E}_i$ are the entries that belong to task T_i .
- Entry e has one or more phases p . For simplicity, the maximum number of phases that any one entry can contain is three.
- The execution demands of entry e , phase p , (e, p) , is denoted s_{ep} , and the mean number of requests from entry e to entry d in phase p is denoted by y_{edp} .

The objective of the OBA is to compute the utilization, U , for each task and processor. To compute utilization, both service time and throughput must be approximated for each entry.

The first quantity to be computed is the optimistic service time for (e, p) , denoted by x_{ep} . This optimistic service time is given by its processor execution

demand s_{ep} plus the optimistic phase 1 service time for each entry that is called by (e, p) , such that:

$$x_{ep} = s_{ep} + \sum_{d \in D} y_{edp} x_{dl} \quad (3.1)$$

where D is the set of all entries that are called by (e, p) . This value is computed recursively, starting at the the reference tasks, following links to pure servers. By definition, pure servers have $x_{ep} \doteq s_{ep}$.

Throughputs of the entries are related to each other by the traffic equations from [4]. In the current notation, the throughput of entry e is given by

$$f_e = \sum_{d \in D} f_d Y_{de} \quad (3.2)$$

where D is the set of entries that use e via rendezvous, and $Y_{de} = \sum_p y_{dep}$. Recursive application of this relationship gives an entry's throughput in terms of reference throughput, such that:

$$f_e = \sum_r f_r a_{re} \quad (3.3)$$

where a_{re} is a constant to be determined. To compute a_{re} , set $f_r = M_r$, where M_r is the multiplicity of reference task r , and set $f_{n=1 \dots R, n \neq r} = 0$. Equations 3.2 and 3.3 can now be used to compute a_{re} .

The last quantity to be computed is the task utilization; based on f_e and x_{ep} , the task utilization is

$$U_i = \sum_{e \in \mathcal{E}_i} f_e X_e$$

where $X_e = \sum_p x_{ep}$. Since a task's utilization must be less than or equal to its multiplicity M , we have

$$U_i = \sum_{e \in \mathcal{E}_i} f_e X_e = \sum_r f_r \sum_{e \in \mathcal{E}_i} a_{re} X_e = \sum_r f_r K_{ri} \leq M_i. \quad (3.4)$$

Thus, for a model with i tasks (including the reference tasks), there will be i inequalities that bound the R reference task throughputs. The goal now is to determine the reference task throughputs, f_r , using the bounding equations.

With one reference task, it is clear that $f_r \leq \frac{1}{\max_i K_{ri}}$. We consider the problem of two or more reference tasks in Chapter 4.

The bounding equation for an artificial processor task is

$$U_q = \sum_{e \in \mathcal{E}_i, i \in \mathcal{P}} \sum_r f_r a_{re} \sum_p s_{ep} \leq N_q. \quad (3.5)$$

where $\mathcal{E}_i$ is the set of entries belonging to task i , N_q is the number of processors (if q is a processor with multiplicity greater than 1, otherwise, $N_q = 1$) and $\mathcal{P}$ is the set of tasks hosted on processor q . Equation 3.5 can be derived from the entry service time equation 3.1, noting that the artificial processor task has artificial entries with service times corresponding to the task's real entries. Equations 3.2, 3.3 and 3.4 can then be combined to arrive at the the bounding equation for an

artificial processor task. Note that, for a processor task, $K_{rq} = \sum_{e \in \epsilon_i} a_{re} \sum_p s_{ep}$.

3.4.2 The Algorithm for the OBA

We now present the algorithm used for the Optimistic Bounds Analysis.

1. Order the tasks from 1 to T , such that, if task T_i has an entry that performs a rendezvous with an entry of task T_j , $i < j$. As with the LQN (SRN) algorithm in [4], if this ordering cannot be performed, there is a loop in the task request group, and the analysis should exit. With such an ordering, the reference tasks will be tasks $1 \dots R$.
2. From the ordered task list, create the ordered entry list. The ordering of tasks in step 1 ensures that entries from task T_i will appear before entries from task T_j in the ordered entry list, if $i < j$.
3. For each entry in the ordered entry list, starting at the end of the list, compute x_{ep} using equation 3.1.
4. For each reference task, compute a_{re} for each entry, starting at the first non-reference task entry (e_{R+1}) in the ordered entry list, using Equations 3.2 and 3.3.
5. Determine the bounding equation for each task, using Equation 3.4.
6. Determine the bounding equation for each artificial processor task, using Equation 3.5.

7. If there is only one reference task in the model, the maximum reference task throughput $f_1 \leq \frac{1}{\max_i K_{ri}}$. With this estimate for f_1 , per task utilization values can be determined.

3.4.3 An Example

To demonstrate the use of the OBA, we use the “Basecase” system from [5], as shown in Figure 3.2, with parameters in Table 3.3.

1. Task ordering: We note that the tasks as currently labeled are correctly ordered.
2. Entry ordering: We note that the entries as currently labeled are correctly ordered.
3. Service time computation: The optimistic service times for each entry are

found to be:

$$x_{51} = s_{51} = 1$$

$$x_{52} = s_{52} = 1$$

$$x_{41} = s_{41} = 1$$

$$x_{42} = s_{42} = 1$$

$$x_{31} = s_{31} = 1$$

$$x_{32} = s_{32} = 1$$

$$x_{21} = s_{21} + y_{231}x_{31} + y_{241}x_{41} + y_{251}x_{51} = 5.5$$

$$x_{22} = s_{22} + y_{232}x_{32} + y_{242}x_{42} + y_{252}x_{52} = 5.5$$

$$x_{12} = s_{12} + y_{122}x_{22} = 15.5$$

4. Traffic relationships:

$$a_{11} = 10$$

$$a_{12} = a_{11}Y_{12} = 10$$

$$a_{13} = a_{12}Y_{23} = 30$$

$$a_{14} = a_{12}Y_{24} = 30$$

$$a_{15} = a_{12}Y_{25} = 30$$

5. Task bounding equations:

$$155f_1 \leq 10 \quad \text{Task 1}$$

$$110f_1 \leq 1 \quad \text{Task 2}$$

$$60f_1 \leq 1 \quad \text{Task 3}$$

$$60f_1 \leq 1 \quad \text{Task 4}$$

$$60f_1 \leq 1 \quad \text{Task 5}$$

6. Artificial Processor Task bounding equations:

$$20f_1 \leq 1 \quad \text{Processor 2}$$

$$60f_1 \leq 1 \quad \text{Processor 3}$$

$$60f_1 \leq 1 \quad \text{Processor 4}$$

$$60f_1 \leq 1 \quad \text{Processor 5}$$

7. From the bounding equations, $f_1 = 1/110$. Per task utilization estimates are:

$$U_1 = 0.141$$

$$U_2 = 1.0 \quad U_{P2} = 0.182$$

$$U_3 = 0.546 \quad U_{P3} = 0.546$$

$$U_4 = 0.546 \quad U_{P4} = 0.546$$

$$U_5 = 0.546 \quad U_{P5} = 0.546$$

Note that the processor that hosts T_1 is an “infinite server”. For such a processor, the utilization figure is unimportant.

Figure 3.2: Model for the OBA Test

A comparison of the estimates from the OBA and the analytic solver `lqns` for a number of cases is presented in the next section.

3.4.4 Test Results

As a further test of the OBA, and to compare the OBA to the Quick Approximation of [5], we again use the system shown in Figure 3.2 was used, with the parameters in Table 3.3. Results from the `lqns` solution, the OBA and the Quick Algorithm are shown in Tables 3.4, 3.5 and 3.6.

The comparison in Table 3.8 shows that the OBA performs well in most cases, except for the “Update” case. The large error in this case is due to the heavily weighted second phase in entries 3, 4, and 5. Errors due to a second phase are

Parameter	Base Case	RPC	Update	Non-homogeneous	More Centralized
$T_1 = Z_1 = s_{12}$	10	10	10	10	10
y_{122}	1	1	1	1	1
s_{21}	1	1	1.3	1	6
s_{22}	1	1	0.7	1	2
y_{231}	3	1.75	1	2	1
y_{232}	0	1.25	0	0	0
y_{241}	0	0.50	1	0.20	1
y_{242}	3	2.50	0	0.80	0
y_{251}	1.50	0	1	1.50	1
y_{252}	1.50	3	0	1.50	0
s_{31}	1	2	1.40	1	3
s_{32}	1	0	4.60	2	1
s_{41}	1	2	1.4	5	3
s_{42}	1	0	4.60	1	1
s_{51}	1	2	1.40	1	3
s_{52}	1	0	4.60	1	1

Table 3.3: Model Parameters for the Optimistic Bounds Analysis Test.

Performance Measure	1qns Solution	OBA	QA
U_2	1.00	1.00	1.00
U_3, U_4, U_5	0.394	0.546	0.300
U_{P2}	0.135	0.182	N/A
U_{P3}, U_{P4}, U_{P5}	0.394	0.546	0.300
B_2	2.54	1.83	3.33

Table 3.4: Comparison of the 1qns solution, the Optimistic Bounds Analysis and the Quick Approximation for the Base Case.

Performance Measure	1qns Solution	OBA	QA
U_2	1.00	1.00	1.00
U_3, U_4, U_5	0.387	0.5	0.300
U_{P2}	0.130	0.167	N/A
U_{P3}, U_{P4}, U_{P5}	0.387	0.5	0.300
B_2	2.58	2.00	3.33

Table 3.5: Comparison of the 1qns solution, the Optimistic Bounds Analysis and the Quick Approximation for the Non-homogeneous case.

Performance Measure	1qns Solution	OBA	QA
U_2	1.00	1.00	1.00
U_3, U_4, U_5	0.484	0.967	0.300
U_{P2}	0.161	0.323	N/A
U_{P3}, U_{P4}, U_{P5}	0.484	0.967	0.300
B_2	2.58	1.03	3.33

Table 3.6: Comparison of the 1qns solution, the Optimistic Bounds Analysis and the Quick Approximation for the Update case.

Performance Measure	1qns Solution	OBA	QA
U_2	1.00	1.00	1.00
U_3, U_4, U_5	0.231	0.235	0.200
U_{P2}	0.464	0.471	0.400
U_{P3}, U_{P4}, U_{P5}	0.231	0.235	0.200
B_2	2.16	2.12	2.50

Table 3.7: Comparison of the 1qns solution, the Optimistic Bounds Analysis and the Quick Approximation for the More Centralized case.

Variation	B_2 from lqns	B_2 from OBA	B_2 % difference between OBA and lqns	B_2 from QA	B_2 % difference between QA and lqns
Base Case	2.54	1.83	-28.0	3.33	31.1
Non-homogeneous	2.58	2.00	-22.5	3.33	29.1
Update	2.07	1.03	-50.2	3.33	60.9
RPC	3.33	3.33	0.00	3.33	0.00
More Centralized	2.16	2.12	-1.85	2.50	15.7

Table 3.8: Bottleneck Strength for all test variations, from the lqns solution, the Optimistic Bound Analysis and the Quick Approximation.

discussed in Section 4.4.

3.4.5 Limitations of the Better Quick Approximation

As has been briefly noted in Section 3.4, the OBA does not support all of the LQN constructs. The following model constructs are not supported:

- Asynchronous Interactions
- Forwarding Interactions
- Interactions described by precedence expressions
- Open Arrivals
- Service time defined with a coefficient of variation
- Infinite threading

3.5 Conclusions

The concept of a software bottleneck is not new; the authors of [5] provided a useful set of definitions, and a number of examples of the location of a software bottleneck. The contributions of Section 3.1 are two-fold: 1) the formalization of the previous work on software bottlenecks into an algorithm for the location of a software bottleneck, and 2) the extension of the definitions used to define a software bottleneck to include the forwarding interaction.

Once a bottleneck is located, the next step is mitigation. Section 3.2 contributes the concept of classification of a software bottleneck. These two classes, server supported and processor supported, are useful due to different mitigation strategies for each. Section 3.3 presents four such mitigation strategies: 1) increasing threading levels, 2) the addition of processors, 3) reducing phase 1 processor demands at the bottleneck task, and 4) decreasing interactions with lower layers. The first and third mitigation strategies have appeared in the literature. However, the current chapter appears to be the first time all four have been discussed in the context of bottleneck mitigation, with pros and cons, and maximum potential benefit estimated.

The last contribution of this Chapter is the OBA. The OBA is a lightweight approximation technique, useful for estimating task utilization for the purposes of bottleneck location. Although the absolute error in the OBA estimates is large, the OBA generally allows for the identification of the correct bottleneck. Limitations of the OBA are the subject of Chapter 4.

Chapter 4

Further Approximation Issues

As discussed in Chapter 3, the OBA is intended to be used to obtain estimates of task level utilization and reference task saturation throughput. To obtain these figures without the solution of queueing network model, the OBA uses a refinement of the *no contention* upper bounds. These no contention bounds ignore the delay introduced by queueing at a server. Thus, for tasks which are a client in a client-server relationship, service times are underestimated. This underestimation of service times leads to an underestimation of task level utilization.

The absolute accuracy of the utilization figures for all tasks from the OBA is not a strict requirement. To be useful in software bottleneck identification, the OBA must be able to:

- Accurately identify the set of saturated tasks, denoted above as Φ .
- Provide accurate relative utilization figures for the set of tasks that are servers of a saturated task b , denoted above as Θ .

Throughout this chapter, the term error¹ is used to indicate the difference between the utilization figure obtained by the OBA and the utilization figure obtained from lqns.

The errors in the OBA's utilization estimates are, in large part, due to the fact that the approximation ignores contention, causing three problems. First, the OBA sets the queueing delay term in the LQN waiting equation to zero. Secondly, the OBA does not account for the contention at the task level due to multiple entries being contained within a single task. Finally, the OBA disregards processor contention when there is more than one task assigned to a processor. These factors all contribute to the OBA estimating service times lower than lqns would, and causing the utilization figures to be corresponding lower.

Further errors are introduced by the simplistic multi-threading calculation used by the OBA. The utilization figure for a multi-threaded task can be viewed as the mean number of busy threads. The OBA does not consider practical bounds on the mean number of busy threads that are introduced by the static structure of a system (i.e. constraints due to a lack of client threads). As a result, the OBA can over-estimate the mean number of busy threads and the task's utilization.

The last major source of error is due to the OBA is the second phase estimation. Previous authors have noted the difficulties that a second phase of services causes in the estimation of service time and utilization. In the tests of Section 3.4, large errors due to dominant second phases were noted. A correction term and a second iteration of the approximation is considered in an attempt to reduce the

¹Error being defined as $\frac{U_{OBA} - U_{lqns}}{U_{lqns}} \times 100$.

OBA's errors.

Minor sources of error in the OBA stem from the stochastic nature of the LQN model. The service times and request rates are assumed by LQNS to be geometrically distributed, with the value specified as the mean. The current LQNS solver allows the user to alter the service time distribution through the specification of a coefficient of variation, or the use of a deterministic phase. The OBA disregards the statistical nature of these parameters. In this regard, the OBA assumes that all entries have deterministic phases.

This chapter focuses on six areas of difficulty for the OBA. The first three are centred around contention, with a general discussion of contention-causing architectures, followed by a discussion of the impact of multi-threading, and the impact of multiple entries per task. We then look at the errors caused by the second-phase treatment in the OBA, and propose a correction. In general, the OBA allows for multiple reference tasks; a discussion of the impact of multi reference tasks on the accuracy and usability of OBA estimates is given. The last approximation issue is the handling of systems near saturation.

For each of these six approximation issues, we give general recommendations for the use of the OBA; most of these recommendations suggest cases where the OBA does not produce usable results. A summary of these recommendations is given at the end of this chapter.

4.1 Contention and the OBA

The main simplification of the OBA when compared to the analytic solver `lqns`, is the removal of the computation of a waiting time. When waiting time is low, we expect the the OBA to return utilization estimates that are reasonably close to those of `lqns`. As waiting times increase, we expect that the OBA will return poorer utilization estimates. In this section, we use three common distributed architectures, the funnel, pyramid, and the tower to explore the impact of contention on the OBA. For the purposes of this section, we equate an increased waiting time with contention; client waiting time at a server is used to measure contention for that server.

The *funnel*, *tower* and *pyramid* systems under consideration are shown in Figure 4.1. In the context of the current discussion, the following differences between these architectures is important²:

Tower The tower architecture describes a set of layered servers, which are piled one on top of the other. With the exception of the top layer (representing the user tasks) and the bottom layer, each layer has an equal number of tasks.

Funnel The funnel architecture describes a set of layered servers, where at each server layer, there are fewer server tasks than client tasks in the layer above.

In such a system, rendezvous requests are “funneled” into the lowest layer.

The funnel often exasperates server contention.

²Throughout this Chapter, we refer to the $M : N$ ratio; the ratio of the total number of active entities (tasks/threads) at layer n and total number of active entities at layer $n + 1$. For a funnel, $M > N$, for a pyramid, $M < N$, and for a tower, $M = N$.

	OBA Utilization Estimate	1qns Utilization Estimate	Relative Difference
T_2	1.000	0.844	18.5%
T_3	0.833	0.704	18.3%
T_4	0.667	0.563	18.5%
T_5	0.167	0.141	18.4%
T_6	0.167	0.141	18.4%
T_7	0.167	0.141	18.4%

Table 4.1: OBA and 1qns Utilization Estimates for the Tower System

	OBA Utilization Estimate	1qns Utilization Estimate	Relative Difference
T_2	1.000	0.927	7.87%
T_3	0.417	0.386	8.03%
T_4	0.417	0.386	8.03%
T_5	0.167	0.155	7.74%
T_6	0.167	0.155	7.74%
T_7	0.167	0.155	7.74%

Table 4.2: OBA and 1qns Utilization Estimates for the Pyramid System

Pyramid The opposite of the the funnel, each server layer in the pyramid architecture has more server tasks then client tasks in the layer above. With more servers then clients, server contention is usually reduced.

In each system, the amount of contention for lower layer server resources is a function of the client layer; in the tower, we expect that the contention level is similar at each level, whereas the amount of contention increases as one goes lower in the funnel, and decreases as one goes lower in the pyramid.

For each system, each entry has one unit of phase one execution time. As well, each entry makes one visit to each of its direct servers, with the exception of the funnel system, where $y_{122} = y_{132} = y_{142} = 0.333$, $y_{251} = y_{261} = y_{351} = y_{361} = y_{451} = y_{461} = 1$, and $y_{571} = 1.5$. Utilization estimates for the three systems are

(a) The Funnel System

(b) The Tower System

(c) The Pyramid System

Figure 4.1: The Funnel, Tower and Pyramid Systems.

	OBA Utilization Estimate	1qns Utilization Estimate	Relative Difference
T_2	0.645	0.624	3.37%
T_3	0.645	0.624	3.37%
T_4	0.645	0.624	3.37%
T_5	0.806	0.537	50.1%
T_6	0.806	0.537	50.1%
T_7	0.967	0.555	44.0%

Table 4.3: OBA and 1qns Utilization Estimates for the Funnel System

shown in Tables 4.1, 4.2 and 4.3. As expected, the OBA produces reasonable results for the systems where the contention is lower (the tower and pyramid). The interesting system is the funnel, where the OBA estimates do not help in the location of the software bottleneck. The 1qns results suggest that software bottlenecks exist at T_2 , T_3 and T_4 , with a strength of 1.16; a weak software bottleneck. However, these tasks are quite some distance from saturation, if we consider the typical U_{sat} of 0.75. As such, these tasks are not true software bottlenecks. The OBA suggests that T_7 is saturated with a utilization near 1.0, while the 1qns solution gives a utilization estimate for T_7 that is near half of the OBA's.

In the funnel case, we have 2:1 contention at task T_7 . Without a waiting time calculation, the OBA will underestimate the utilization of T_5 and T_6 . Similar effects are seen at tasks T_2 , T_3 and T_4 . In general, as the funnel ratio increases, the OBA estimates worsen even when the server task is lightly utilized. In effect, the no blocking assumption of the OBA creates a task with an infinite number of threads; in an infinite threading case, there is always a thread ready to accept a rendezvous.

The overriding recommendation is that the OBA should be used with caution

when a steep funnel ($M \ll N$) appears in an architecture, due to the lack of contention estimation. In these cases, the analytic solver is recommended. For other architectural patterns, where contention is not as much of an issue, such as the pyramid or tower, the OBA generally produces reasonable results.

4.2 Errors due to the Multi-threading Approximation

The second contention variation that we explore is the impact of multi-threading on the accuracy and usability of the OBA estimates. To examine the impact of multi-threading, we use the simple tower system from Figure 4.1. In the basecase, we set all service times to one, and each server is invoked once per response. We set the number of threads at T_1 to be 10, representing 10 users.

The OBA gives the following utilization estimates: $U_2 = 1.0$, $U_3 = 0.833$, $U_4 = 0.667$ and $U_5 = U_6 = U_7 = 0.167$. From these results, we locate the bottleneck at task T_3 , with $B_3 = 1.25$. Our first experiment increasing the threading levels at the bottleneck task T_3 . The OBA utilization estimates from this experiment are given in Table 4.4.

From the results in this table, it is clear that the OBA does not generate correct utilization estimates. However, in each of these cases, the OBA does identify T_2 as the bottleneck; the concern is that the strength estimates are far too large.

The main reason for the errors that we see in Table 4.4 is that the increasing of threading levels at T_3 has transformed the tower into a funnel. In the previous

Number of Threads	OBA Utilization Estimate	lqns Utilization Estimate
2	0.417	0.913
3	0.278	0.792
4	0.208	0.767
5	0.167	0.757
8	0.104	0.747
10	0.083	0.745

Table 4.4: Increasing the Number of Threads at T_3 .

section, we saw that the OBA does not product reasonable throughput estimates for funnel-like systems. Again, the error is attributable to the lack of a waiting time computation in the OBA.

An interesting set of tests for the OBA use the Tower Pattern tests from [29]. Both the OBA and lqns results for these tests are shown in Tables 4.5 and 4.6. It is interesting to note that the OBA locates the correct bottleneck in both the (2,1,1) and (3,2,1) cases, even though the utilization estimates appear to have large errors. Using the “Theading Rule of Thumb” [29], the ideal threading configuration is (6,5,4). It is interesting to note that the OBA estimates all utilization values equal to one in this case. However, the OBA estimate of the utilization of the pure server threads is approximately twice the lqns estimate. As the threading levels continue to increase, the funnel effect becomes more pronounced. This leads to the results being unusable for bottleneck identification; in the cases where the threading level is higher then the ideal, the pure server tasks are identified as the bottlenecks.

Is the all-ones result from the ideal threading level case a fluke? To demonstate that this result is not general, we create a four level pyramid, with four tasks in the

	(10,10,10)	(9,8,7)	(8,7,6)	(7,6,5)	(6,5,4)	(3,2,1)	(2,1,1)
U_2	0.600	0.667	0.750	0.857	1.00	0.500	0.600
U_3	0.500	0.625	0.714	0.833	1.00	0.625	1.00
U_4	0.400	0.571	0.667	0.800	1.00	1.00	0.800
$U_{5,6,7}$	1.00	1.00	1.00	1.00	1.00	0.250	0.200

Table 4.5: Multi-threading in the Tower - OBA Utilization Estimates.
The threading levels are defined by (m_2, m_3, m_4) .

	(10,10,10)	(9,8,7)	(8,7,6)	(7,6,5)	(6,5,4)	(3,2,1)	(2,1,1)
U_2	0.996	1.00	1.00	1.00	1.00	1.00	1.00
U_3	0.869	0.935	0.941	0.947	0.953	0.980	0.826
U_4	0.744	0.876	0.888	0.900	0.911	0.909	0.661
$U_{5,6,7}$	0.705	0.668	0.636	0.596	0.544	0.227	0.165

Table 4.6: Multi-threading in the Tower - lqns Utilization Estimates.
The threading levels are defined by (m_2, m_3, m_4) .

pure server level ($T_5 \dots T_8$), two tasks above these servers (T_3, T_4), and one task at the top (T_2), with 10 client tasks. Again, all of the service times are 1, and the visit rates are also 1.

This configuration has a ideal threading level of $(m_2, m_3, m_4) = (7, 3, 3)$. The OBA yields the following estimates: $U_2 = 0.785$, $U_3 = U_4 = 0.833$ and $U_{5,\dots,8} = 1.0$. The lqns estimates are $U_2 = 1.0$, $U_3 = U_4 = 0.806$ and $U_{5,\dots,8} = 0.598$. Obviously, the OBA results lead to an incorrect bottleneck location.

The effect that is often seen in multi-threaded systems is “push-back”. Push back refers to inflated utilization in layers above a saturated task. In this pyramid example, we have all pure servers saturated. Since the servers exclusively execute in phase one, the clients of these servers should also be saturated. The OBA is incapable of detecting conditions under which push back occurs. As such, the

OBA utilization estimates for clients of saturated servers are not correct. This is one of the principal weaknesses of the OBA.

To perform an accurate estimate of the utilization at a multi-threaded server (or for a client of a multi-threaded server), one has to be able to estimate the queueing delay. Rolia [30] provides a simple estimate for the probability of finding all servers busy, but to use this to determine the queueing delay, a population (queue length) estimate is needed. For simple systems, such as the ones discussed in this Chapter, it is possible to accurately estimate the population, but in general, a queueing network solution is required. It is for this reason that there is not an attempt to improve the OBA's estimation of utilization when the model contains multi-threaded servers.

The results from these simple threading experiments lead us to two recommendations on the use of the OBA:

1. Do not rely on the utilization estimates from the OBA when the threading levels approach or exceed the threading levels recommended by the "Threading Rule of Thumb".
2. The user of the OBA is cautioned with regards to the validity of the estimates when the results indicate that there should be "push-back"³. In these cases, the OBA will not accurately estimate utilization. Multi-threading exacerbates the push-back problem.
3. Do not rely on the utilization estimates from the OBA when there is excessive "funneling" created by multi-threading. Excessive funneling is often

seen when condition 1) is violated.

4.3 The Impact of Multiple Entries per Task on the OBA

The final contention variation that we examine is the impact of multiple entries per task on the accuracy and usability of the OBA utilization estimates. When there are more clients than servers in a layered system, the potential for resource contention increases. Combining multiple entries into a single task increases the “funnelness” of the system. As we have seen above, the greater the funnel effect, the greater the error in the OBA utilization estimates.

Consider a system with a layer of n client tasks and m server tasks, where $n > m$, and each of the server tasks has one entry. Assume that each client makes one visit to some percentage p of the m servers. If the utilization of each of the m servers is low, the interference between the tasks will be corresponding low. As the utilization of the server increases, the probability that a client attempts a rendezvous with a server that is busy increases. Now, consider the same n clients performing rendezvous with m entries hosted at one server task. In effect, this creates an $n : 1$ funnel; a classic software bottleneck.

If the utilization of each of the m entries is low, such that $U_i = \sum_m U_m$ is small (for example less than 0.5) the OBA will most likely produce usable utilization

³“Push-back” refers to blocking seen in layers above a saturated task, which causes inflated utilizations for the blocked tasks. The saturated task is said to be pushing back on its clients.

estimates. As the total utilization increases, the probability that a client will have to wait also increases. The OBA utilization estimate for the client will suffer due to the lack of a waiting time estimate. In effect, the OBA treats a multi-entry task as if it had an infinite number of threads. Two effects are seen; the first is push back as discussed above, and the second is that the maximum throughput estimate will be too high (due to the underestimation of waiting time in the task bounding equation), since the maximum throughput is the inverse of the maximum bounding equation coefficient. This overestimation of throughput causes OBA utilization estimates to be artificially increased below a contention point, since more traffic is admitted than would be realistic.

We have found many models where the n clients and m entries hosted in one task produces lower than expected contention. This is due, in most part, to traffic dependencies, where a client makes p visits to a server task. These p visits are in series (if they are rendezvous interactions), and do not contend with each other. A more general discussion of this problem, termed *interlocking*, is provided in [4].

The following are recommendations for the use of the OBA when multiple entries are hosted by a single task:

1. When a multi-entry server task's utilization estimate is high (greater than 0.5, for example), assume that the estimate for the client utilization will be low, due to the lack of contention estimation in the OBA. In these cases, there are two options: use of the analytic solver, or adjustment of the service times to account for the waiting due to contention .

2. Conversely, the OBA appears to produce reasonable estimates when a multi-entry server task has a relatively low utilization.
3. In general, the multi-entry task recommendations hold for multi-task processors. This follows from the artificial task treatment of processors in the OBA.

4.4 Errors due to Second Phases

Second phases of service have caused many authors trouble; Franks, in [17], gives a good summary of the current second phase computation currently in use in the *lqns* solver, as well a history of analytic second phase approximations. From the “Update” case in Section 3.4, we have seen that the OBA is far too optimistic in the prediction of an entry’s throughput when the entry that is dominated by a second phase is heavily utilized. We explore the optimism of the OBA in this section.

To quantify the degree of optimism that the OBA exhibits, we vary the second phase service time percentage in the “Update” model from 0 to 100 % while maintaining a constant total service time of 6 units. The utilization estimates given by the OBA and *lqns* for entries 3, 4 and 5 are given in Table 4.7, and are plotted in Figure 4.2.

The first indication of trouble with the OBA is at the 50% phase 2 level, where the error from the approximation is greater than 20%. More troubling are the OBA results at the 80% phase 2 level and up, where the approximation indicates 100%

Phase 2 Percentage	OBA Estimate	1qns Estimate	Relative Difference
0	0.300	0.293	2.33
10	0.330	0.317	3.94
20	0.365	0.340	6.85
30	0.411	0.365	11.2
40	0.469	0.390	16.8
50	0.545	0.415	23.9
60	0.652	0.441	32.4
70	0.811	0.467	42.4
80	1.00	0.492	50.8
90	1.00	0.516	48.4
100	1.00	0.540	46.0

Table 4.7: A comparison of OBA and 1qns utilization estimates for entries 3, 4 and 5 of the “Update” case while shifting service time from phase 1 to phase 2.

utilization of these entries, while 1qns indicates utilizations around 50%. Furthermore, the OBA results indicate that the system no longer contains a software bottleneck (as the pure server entries are at 100% utilization), while the 1qns results still indicate a software bottleneck at entry 2. These difficulties are direct result of the OBA first assumption; in this case, as the phase 2 percentage is increased, the amount of blocking that entry 2 experiences at entries 3, 4 and 5 increases.

In an attempt to correct this deficiency of the OBA, we introduce a correction factor to account for the overtaking behaviour. For a two phased entry j , Woodside, in [4], presents the following expression for a request’s waiting time:

$$W_j = x_{j1} + \Pr\{\text{OT}\}_j x_{j2} + x_j L_j (N - 1)$$

Figure 4.2: A graphical comparison of the data from Table 4.7.

where the $x_j L_j (N - 1)$ term is the delay due to requests from other entries at this entry. For the OBA, we ignore this term. The term of interest is the $\Pr\{\text{OT}\}_j$ term; this is the “correction” term that we desire.

The most simple correction term is presented by Woodside as

$$\Pr\{\text{OT}_j\} = \frac{x_{j2}}{x_{j2} + 1/\lambda_j - W_j}.$$

The $1/\lambda_j$ term represents the mean arrival rate of requests from an particular client entry i to the server entry j . The W_j term represents the mean total delay at entry j ; again, for the purposes of the OBA, we ignore this term.

To “correct” the OBA, we perform the calculation loop a second time. The calculations are modified as follows:

- Equation 3.1 is modified to be: $x'_{ep} = x'_{ep} + \sum_{d \in D} y_{edp} x'_{d1} \delta_d$, where $\delta_d = \frac{x_{d2}}{x_{d2} + 1/\lambda_d}$. We use the throughput estimate generated by the first loop in the correction term δ_j .
- The bounding equations for each entry are recomputed based on the x'_{ep} values.

The following steps are added to the OBA algorithm presented in Section 3.4:

1. Steps 1 through 7 are as before.
8. If an estimate for f_1 cannot be generated exit the algorithm.

Phase 2 Percentage	Corrected OBA Estimate	lqns Estimate	Relative Difference
0	0.299	0.293	2.01
10	0.329	0.317	3.65
20	0.365	0.340	6.85
30	0.407	0.365	10.3
40	0.457	0.390	14.7
50	0.512	0.415	19.0
60	0.571	0.441	22.8
70	0.626	0.467	25.4
80	0.674	0.492	27.0
90	0.741	0.516	30.4
100	0.813	0.540	33.9

Table 4.8: A comparison of Corrected OBA and lqns utilization estimates for entries 3, 4 and 5 of the “Update” case while shifting service time from phase 1 to phase 2.

9. Otherwise, for each entry in the ordered entry list, starting with the last entry in the list, compute x'_{ep} .
10. Determine the new bounding equation, using Equation 3.4 substituting x' values for x values.
11. Determine the maximum reference task throughput, and with this value, and the bounding equations, compute task and processor utilization estimates.

We now use the corrected version of the OBA with the “Update” case, again varying the second phase service demand from 0 to 100 percent. The corrected OBA values are give in Table 4.8, and shown graphically in Figure 4.3.

Comparing Figures 4.2 and 4.3, we see that the Corrected OBA produces utilization estimates that are closer to those from lqns. More importantly, the Cor-

Figure 4.3: A graphical comparison of the data from Table 4.8.

rected OBA does not allow entries 3, 4 and 5 to become saturated as was the case in the original OBA. Furthermore, the Corrected OBA always estimated the utilization at entry 2 as 1.0, where the original OBA did not as the utilization of entries 3, 4 and 5 approached 1.0. Due to this problem, using the results of the OBA for bottleneck location would identify the wrong task as the bottlenecked task. The original OBA would have suggested that tasks 3, 4 and 5 were the bottlenecks, whereas both the corrected OBA and *lqns* identify task 2 as the software bottleneck.

The divergence between the *lqns* estimates and the Corrected OBA estimates is expected. Franks, in [17], showed that the server based overtaking calculation often produced large errors. He presented a more complex analysis of overtaking, with a correspondingly more complex expression for $\Pr\{\text{OT}\}_j$. For the purposes of an approximation, we feel that the simple server based expression for overtaking probability is sufficient. We do, however, append a caution on the use of the Corrected OBA. We do not recommend the use of the Corrected OBA for systems with a number of entries with dominant second phases. Rather, we recommend the use of the analytic solver *lqns*.

4.5 The OBA and Multiple Reference Tasks

Up to this point, we have only considered using the OBA on models with a single reference task. In Section 3.4, the OBA algorithm terminated at step 6 (before the task utilizations were estimated) if the model contained more than one reference

Figure 4.4: Model with Two Reference Tasks.

task. We now consider the two reference task case.

Throughout this section, we use the model shown in Figure 4.4 as the basis for our discussions. The model parameters are given in Table 4.9.

For this system, the OBA produces the following bounding equations:

Parameter	Value	Parameter	Value
s_{12}	10	y_{132}	1
s_{22}	5	y_{232}	2
s_{31}	0.1	y_{341}	2
s_{41}	0.1	y_{451}	2
s_{51}	0.2	y_{461}	4
s_{61}	0.1	y_{471}	2
s_{71}	0.2		

Table 4.9: System Parameters for the Two Reference Task Model.

$$\begin{array}{ll}
12.7f_1 \leq 1 & \text{Task 1} \\
10.4f_2 \leq 1 & \text{Task 2} \\
27f_1 + 43.2f_2 \leq 1 & \text{Task 3} \\
26.0f_1 + 41.6f_2 \leq 1 & \text{Task 4} \\
8f_1 + 12.8f_2 \leq 1 & \text{Task 5} \\
8f_1 + 12.8f_2 \leq 1 & \text{Task 6} \\
8f_1 + 12.8f_2 \leq 1 & \text{Task 7}
\end{array}$$

The feasible throughput region is shown graphically in Figure 4.5. From this figure, we see that the feasible throughput region is bounded by $f_1 \geq 0$, $f_2 \geq 0$ and $27f_1 + 43.2f_2 \leq 1$. To obtain utilization estimates, the OBA requires the maximum feasible values for both f_1 and f_2 . Our first approach to determine f_1 and f_2 uses the ratio $Y_{13} : Y_{23} = f_1 : f_2 = 10 : 16$, or $f_2 = \frac{10}{16}f_1$. Solving the Task 3 bounding equation with this ratio gives $f_1 = 1/54$ and $f_2 = 1/86.4$. Using these values to estimate task utilizations gives the results in Table 4.10. From these results, we see that the OBA produces reasonable estimates for a two-reference task system with a single task as the bottleneck. In this case, Task 4 is the software bottleneck.

To make the system more interesting, we set the system's parameters to those shown in Table 4.11. In this sample system, we have moved Task 4 up to be a peer of Task 3, and we have directed all of the requests originating from Task 1 to Task 3, and from Task 2 to Task 4. The pure server tasks have not changed. The OBA gives the following bounding equations:

Figure 4.5: Feasible Throughput Region for the Two Reference Task Model. The feasible region is shaded grey. Note that the bounds for tasks 1, 2, 5, 6 and 7 are well outside of the range, and not shown.

	OBA Utilization Estimate	lqns Utilization Estimate	Relative Difference
Task 3	1.00	0.987	1.32%
Task 4	0.963	0.951	1.26%
Task 5	0.296	0.293	1.02%
Task 6	0.296	0.293	1.02%
Task 7	0.296	0.293	1.02%

Table 4.10: Utilization Estimates for the Two Reference Task Model.

Parameter	Value	Parameter	Value
s_{12}	10	y_{132}	1
s_{22}	5	y_{242}	2
s_{31}	0.1	y_{351}, y_{451}	4
s_{41}	0.1	y_{361}, y_{461}	8
s_{51}	0.2	y_{371}, y_{471}	4
s_{61}	0.1		
s_{71}	0.2		

Table 4.11: System Parameters for the Modified Two Reference Task Example.

$$125f_1 \leq 10 \quad \text{Task 1}$$

$$80f_2 \leq 8 \quad \text{Task 2}$$

$$25f_1 \leq 1 \quad \text{Task 3}$$

$$40f_2 \leq 1 \quad \text{Task 4}$$

$$8f_1 + 12.8f_2 \leq 1 \quad \text{Task 5}$$

$$8f_1 + 12.8f_2 \leq 1 \quad \text{Task 6}$$

$$8f_1 + 12.8f_2 \leq 1 \quad \text{Task 7}$$

Solution of this system provides an estimate of $f_1 = 0.025$ and $f_2 = 0.04$. Substitution of these estimates gives the task utilization estimates shown in Table 4.12.

From the OBA results, we compute bottleneck strengths for Task 3 and Task 4 of 1.56; from the lqns estimates, we compute B_3 and B_4 of approximately 2.06.

Why does the OBA give very reasonable estimates for the single bottleneck case, and less reasonable estimates when there is more than one bottleneck?

- Contention is increased.

	OBA Utilization Estimate	lqns Utilization Estimate	Relative Difference
Task 3	1.00	0.987	1.32%
Task 4	1.00	0.992	0.81%
Task 5	0.64	0.486	31.7%
Task 6	0.64	0.486	31.7%
Task 7	0.64	0.468	31.7%

Table 4.12: Utilization Estimates for the Modified Two Reference Task Model.

- Push back from Task 5, 6 and 7 to Tasks 3 and 4 is not modelled by the OBA.

From these two sets of results, we recommend caution in the use of the OBA when a model has multiple reference tasks. Specifically, we suggest that the OBA is only relied upon when the results show only one software bottleneck task. Where there is more than one software bottleneck, the OBA does not model the contention accurately enough; we recommend the use of the analytic solver in these cases.

4.6 Systems Near Saturation

For the OBA to be useful in the location of software bottlenecks, it must be able to identify tasks that are in saturation. All of the models considered thus far have had one task that is well beyond the saturation point U_{sat} . In this section, we examine the OBA's behaviour when used with a system that has a task near U_{sat} .

For discussion, consider the “Basecase” model from Section 3.4.4. With the parameters presented above, the Corrected OBA generates the following bounding equations:

$$182.98f_1 \leq 10 \quad \text{Task 1}$$

$$147.62f_1 \leq 1 \quad \text{Task 2}$$

$$66.429f_1 \leq 1 \quad \text{Task 3}$$

$$66.429f_1 \leq 1 \quad \text{Task 4}$$

$$66.429f_1 \leq 1 \quad \text{Task 5}$$

Solving these bounding equations for f_1 identifies Task 2 as the bottleneck task, with a utilization of 1.0. To bring Task 2 out of saturation, we need to lower its utilization to below U_{sat} ; for the purposes of this section, assume $U_{sat} = 0.75$. In this model, there are a number of parameters that can be changed to bring Task 2 out of saturation; we will limit ourselves to two: increasing user think time, and lowering the number of users.

From the bounding equations, it is clear that increasing the user think time will increase K_1 ; for the system to become “source limited” and to bring Task 2’s utilization below U_{sat} , we require $U_{sat}K_1/M_1 \geq K_2/M_2$. In the Basecase, $K_1 = s_{12} + x_{21}$, and $s_{12} = 10$. Thus, we need to increase the user think time s_{12} to 188.529; a very large increase indeed.

To obtain an understanding of the estimating ability of the OBA as a task enters saturation, we decrease s_{12} from 200 to 10 in 20 unit drops. The user throughput and utilization estimates produced by both the Corrected OBA and 1qns are given in Table 4.13, and shown graphically in Figure 4.6.

From Figure 4.6, we see that as the utilization of Task 2 decreases, there is

s_{12}	OBA f_1 Estimate	lqns f_1 Estimate	OBA U_2 Estimate	lqns U_2 Estimate	Relative Error
200	0.00483	0.00447	0.642	0.672	-4.46
180	0.00533	0.00486	0.721	0.731	-1.37
160	0.00597	0.00531	0.821	0.799	2.75
140	0.00676	0.00581	0.950	0.875	8.57
120	0.00694	0.00632	1.00	0.957	4.49
100	0.00677	0.00674	1.00	1.00	0.00
80	0.00677	0.00692	1.00	1.00	0.00
60	0.00677	0.00690	1.00	1.00	0.00
40	0.00677	0.00686	1.00	1.00	0.00
20	0.00677	0.00680	1.00	1.00	0.00
10	0.00677	0.00676	1.00	1.00	0.00

Table 4.13: Estimates from the Corrected OBA compared to lqns as a task enters saturation.

Figure 4.6: Comparison of the utilization estimates for Task 2 as user think time increases.

s_{12}	OBA U_2 estimate	lqns U_2 estimate
225	0.564	0.609
250	0.503	0.556
275	0.453	0.510
300	0.412	0.472

Table 4.14: Utilization estimates for Task 2 when user think time exceeds 200

a range in which the OBA overestimates Task 2's utilization. This poses an interesting problem, since the OBA should always underestimate utilization. This could be due to any number of reasons, such as stochastic variability in the lqns solution, or an underestimation of utilization by lqns. However, the differences between the lqns estimates and the OBA estimates are within a reasonable range, with the greatest being approximately 8.5%.

As the user think time increases past 200, we find that the OBA tends to estimate utilization with greater error, as shown in Table 4.14. The results from this simple example lead us to not recommend the use of the OBA when the system is source limited, and the largest non-reference task utilization found is less than 75%.

4.7 Summary of Recommendations

In this chapter, we have provided a number of examples of when the OBA does or does not provide useable utilization estimates. In summary, the OBA provides useable estimates when the system has one class of reference task and/or one software bottleneck. In all cases, checking of the estimates from the OBA with

the analytic solver *lgns* is recommended. The frequency of these checks can be reduced when software bottlenecks are a result of included service time, as in pyramid type systems.

The following is a summary of the recommendations and cautions for the user of the OBA. We wish to stress that the OBA is a tool for the estimation of task utilization, with these estimates used to locate software bottlenecks.

1. The user is cautioned against the use of the OBA when large contention causing or steep funnel task architecture are present in the system. Steep funnels can be a direct result of the architecture, or an indirect result of threading or hosting multiple entries at a single task.
2. The use of the OBA is recommended when the task architecture resembles the tower or pyramid.
3. The OBA appears to produce useable results (i.e. useful for bottleneck detection) when threading levels are less than those suggested by the threading rule of thumb.
4. The OBA does not accurately model push-back. The user must be aware of push-back, and either adjust task utilizations or use the analytic solver to obtain useable results. In many cases when a task is saturated, it is the software bottleneck; tasks above it will have only a slightly higher utilization, and a bottleneck strength slightly greater than one. In these cases, the OBA estimates correctly identify the true software bottleneck. However, when a pure server saturates, the OBA estimates are questionable.

5. With the second phase correction, the OBA produces reasonable utilization estimates provided the majority of the tasks in the model are not second phase dominant.
6. Caution is recommended when the OBA is used to estimate utilization for a model that contains two or more reference tasks. When there is more than one independent software bottleneck, the OBA does not model the contention correctly. In the example shown, this did not caused a mislocation of a software bottleneck.
7. The OBA appears to generate useable utilization estimates for systems near the saturation zone. However, the OBA does not always produce accurate utilization estimates when a system is source limited, and no task is near saturation. This is not a drawback, since the primary use of the OBA is for the estimation of utilization in systems that have at least one task in or near saturation.
8. The combination of two or more of these factors (e.g. dominant second phase and multi-threading) causes compound errors. In these cases, the OBA must be used with caution. Even the analytic solver has difficulties with some of the more extreme systems .

Chapter 5

A Case Study

One of the areas of software engineering where SPE has been successfully applied is telecommunications switching systems. Each of today's digital voice switches are the result of over 20 years of evolution of the switch's architecture. Singer, in [31], details the evolution of Lucent's 5ESS voice switch architecture, with emphasis on the following performance aspects: continual performance improvement, the tradeoffs between performance and other properties, call flows and delays, overload control, and software resource engineering.

The evolution of these systems is amazing - in twenty years, the 5ESS has been scaled from under 10 000 line terminations (individual telephone lines) to over 200 000 line terminations, and from 100 000 call attempts per hour to over 2 500 000 call attempts per hour. While these improvements are impressive, many telecommunications experts believe that, from an economical viewpoint, large telephone switches such as the 5ESS are at the end of their evolution. These

experts believe that using lower cost packet switching equipment in place of the large switch provides a more economical way to continue to expand the number of line terminations. The migration from separate voice and data networks to a single data network that transports voice is termed the “Next Generation Network” (NGN). Telcordia Technologies, a major telecommunication standards organization¹, published an architectural framework for the provision of voice services in the NGN in [32].

The creation of the NGN is more than the connection of a number of packet switches, routers and servers. It is expected that NGN elements will not cause a degradation in perceived user level quality and performance. Furthermore, to achieve the promised economic benefits, these elements must scale more effectively than current switches. We discuss the performance and scalability requirements in section 5.1.

There have been some commercial deployments of voice over packet (VoP). The remaining sections of this chapter provide further background material, leading to the development of a model for one solution. The techniques developed in Chapters 3 and 4 are then applied to this model. Finally, the analytic solver *1qns* is used to validate the improvements suggested by the OBA.

¹Telcordia Technologies, formerly known as Bellcore (Bell Communications Research), is the research and development arm of the (former) seven Bell Operating Companies in the United States. Telcordia also provides services to Canadian local telephone companies.

5.1 The Need for Performance Engineering

The provision of residential telephone services is still highly regulated. In Canada, the Canadian Radio and Telecommunications Commission (CRTC) is the government body responsible for setting standards for service providers. From the end customer's point of view (Decision 97-16 [33]), there is one explicit performance requirements that all providers must meet: a user must receive dial tone within 0.5 seconds, 98% of the time. Various CRTC decisions require service providers to follow a number of Telcordia specifications, most notably GR-511 [34], and GR-517 [35]. The user-level performance requirement that we will examine is the "post dialing delay". Post dialing delay is defined as "the time interval between the end of dialing and the occurrence of an audible network response, other than switching noise, to the caller". This delay should not exceed 0.4 seconds for 99.9% of call attempts at an average busy season busy hour load level².

Apart from directly measurable requirements, Telcordia specifies a number of requirements that are of interest to a performance engineer. One such set of requirement is economic system scalability in termination capacity, call capacity, and traffic usage capacity, per switching location. These requirements can be distilled into number of users, the number of calls per hour the user population can make, and the number of simultaneous calls. There are also requirements for

²Telephone system load is defined in units of hundreds of call seconds-per-hour (CCS). Traffic intensities are collected on an hour by hour basis, with the hour having the greatest intensity being called the busy hour. Various loading points can be derived from this data. One such loading point, the average busy season busy hour (ABSBH) load is defined as: "The three months (not necessarily consecutive) with the highest average traffic in the busy hour are termed the "busy season". The busy hour traffic level averaged across the busy season is termed the ABSBH load."

equipment vendors to supply call capacity estimates. Currently, these are provided based on lab measurements; in the future it is conceivable that these estimates could be generated with the help of a performance model, backed up by experimental data.

Further interconnection standards impose other performance requirements. As an example, the SS7 TCAP and ISUP layers have strict call processing time requirements.

In sharp contrast, there are no such regulations in the data switching world. One reason cited in [32] for the slow rate of adoption of VoP solutions is that performance and scalability have been neglected by equipment vendors with a data switching background. This may, in part, be attributable to lack of a requirement, until recently, for SPE in the development of data switching equipment, especially at the requirements stage.

5.2 The Distributed Telephone Switch

5.2.1 Current Voice Switches

Current voice switches are organized into a two level hierarchy. At the lowest level, class 5 switches have subscriber line interfaces and network trunk interface. At the highest level, class 4 switches only have trunk interfaces to interconnect both class 5 and other class 4 switches. Inter-switch communication is almost

Figure 5.1: An abstract view of the current telephone network.

exclusively performed out of band via the SS7 protocol suite³; [36] provides an extensive survey of the protocol suite, and the services that the protocols enable. Also attached to the network, via SS7 protocols, are databases. These databases are the enablers for a number of modern telephone features, from “800” service, to local number portability, to call forwarding. Figure 5.1 provides an abstract view of the current network.

Current voice switches are large distributed systems; most architectures in-

³In Canada, there is a small percentage of switches that are not connected to the SS7 network. These switches use older “in-band” signaling techniques. In-band signaling refers to signaling that is carried in the voice channel, out of band signaling employs separate facilities to transport information.

Figure 5.2: A simplified architecture of a current telephone switch.

ternally distribute processing across a number of modules. As an example, a 5ESS switch will have an administrative module (AM), a communications module (CM), and a number of switch modules (SM). The 5ESS SM is responsible for about 95% of total call processing; the CM is responsible for the communications paths (both voice and control), and performs the rest of the call processing. The AM is the central control point, providing a switch configuration database, and an interface to external management systems. More details of the architecture are provided in [31]; a simplified architecture is presented in Figure 5.2.

5.2.2 Evolution to the NGN Model

As a first commercial step in the evolution to a NGN, class 4 switches and the TDM voice transport network are replaced with VoP switches (trunking gateways), VoP control points (call connection agents), and a packet based transport network. The packet technology employed is ATM; the trunking gateways are derived from an existing ATM switch. The call connection agent consists of a number of servers to perform much of the call processing, interfacing to the trunk-

Figure 5.3: Evolution of the telephone network with the VoP “class 4 switch”

ing gateways and interfacing to external voice switches. The IETF/ITU Protocol H.248 [37] is used for communications between the call connection agents and the trunking gateways.

To an external voice switch, the combination of the call connection agent and trunking gateway appear as a class 4 switch. Thus, in our models, we still see users attached to class 5 switches, however, we do not provide any model for the class 5 switch. The evolution of the traditional network (as depicted in Figure 5.1) is shown in Figure 5.3.

The call connection agent contains the following servers connected by an internal network:

Line Information Database: This database contains all of the information required about the attached gateways in order to make routing decisions.

Call Processing Servers: These servers do the bulk of the call processing work,

Figure 5.4: Simplified Architecture of the Call Connection Agent

making call routing decisions and coordinating connectivity.

SS7 Interface Processor: This server is the call connection agent's interface to the traditional telephone network. It converts SS7 protocol interactions into a set of internal call control messages.

H248 Interface Processors: This servers is the call connection agent's interface to the VoP network. It converts H.248 protocol interactions into a set of internal call control messages.

Figure 5.4 gives a graphical representation of the servers in the call connection agent.

The trunking gateway switch, as shown in Figures 5.3 and 5.5 consists of the following processing modules (cards), interconnected by an intra-switch network:

H248 Call Processing Card: This module terminates H.248 messages, converting them into internal "switched services" connection requests.

Trunk Interface Cards: These modules provide the trunk side interfaces. The trunk interface cards play a minor role in the call setup path (with which we are interested), and a major role in the voice/packetization path (with which we are not interested).

Network Interface Cards: These modules provide the network side interfaces. As with the trunk interface cards, these cards play a minor role in the call setup path.

Control Card: This module provides the overall control and connection database for the switch.

SVC Call Processing Card: This module terminates the “switched services” connection requests, and provides an interface to the ATM signaling network. All connections on the switch are created via ATM switched virtual circuit (SVC) signaling methods.

The architecture of the trunking gateway is given in Figure 5.5. The core packet switch is built using the SVC Call Processing, Control, and Network Interface cards from the gateway, and runs similar software.

5.3 Layered Queueing Model of the System

Figure 5.6 shows the task level LQN model of the VoP distributed class 4 switch. As described in earlier sections, the system has four major components; the Call

Figure 5.5: Simplified Architecture of the Trunking Gateway Switch

Figure 5.6: Layered Queuing Model of the VoP Distributed Class 4 Switch.

Connection Agent, the Core Switch, and the Originating and Terminating Trunking Gateway Switches.

The LQN model is constructed using design artifacts, such as the Message Sequence Chart of Figure 5.7. An ad-hoc technique similar to the one proposed by Kähkipuro [38] is used to transform these artifacts into the tasks and entries. Note that service times are estimated from experience with a similar system⁴. The LQN input for the case study is shown in Appendix A.

The Orig V task represents the Class 5 telephone switches that surround the distributed class 4 switch; originating a call load of approximately 500 000 calls per hour. It is with this task that we are able to measure the call throughput and setup latency. We expect a call throughput of at least 500 000 calls per hour, and a setup latency of no less than 200 msec. The Term V task represents the terminating Class 5 telephone switch. In this model, it is a pure delay server, with a number of threads approximately equal to the number of simultaneous call attempts.

The Call Connection Agent is comprised of five tasks. The SS7InIp and SS7OutIp tasks run on the same processor, and represent the front end SS7 message processors. Outside of the model, these two tasks could be combined into one task with input and output entries. However, the LQN model does not allow for a loop in the task graph. Threading levels in these tasks are set to the maximum number of simultaneous outstanding SS7 messages allowed. The COCO task is the heart of the Call Connection Agent. The threading levels at this task are dictated by the number of call attempts per second; each COCO task thread is

⁴For commercial reasons, the developer of this system wishes to remain anonymous.

Figure 5.7: Message Sequence Chart for the VoP Distributed Class 4 Switch. Note that only message flow between the four major components is shown.

expected to be able to handle 15 calls per second. The fourth task is the H248Ip, controlling the H.248 messaging interactions between the Call Connection Agent and the Gateways. Again, the threading levels at this task represent the maximum number of simultaneous outstanding H.248 messages allowed. The final task represents the Call Connection Agent's database with a simple single threaded delay server.

Each Gateway is also built out of five tasks. Running on the same processor, the GwH248 and CACO tasks represent the processing required for handling H.248 message exchanges and call control. Once again, the GwH248 task's threading level is set to the maximum number of simultaneous outstanding H.248 messages allowed. Due to a design restriction, the CACO task is limited to two threads. The SVC task is the central task in the Gateway's call setup path. Threading levels at this task represent the maximum simultaneous number of Q.2931 (ATM SVC) messages. This task requests call routing information from the CTL task, configures endpoint information on chosen interface and network line cards, and interacts with the Core Switch's SVC task. The CTL, ILC and NLC tasks are single threaded delay servers, representative of elements in the Gateway. Note that for the originating Gateway, there are no interactions between the SVC task and the CACO task.

The Core Switch is built from the same platform as the Gateways, with the absence of VoP signaling processors. As a core switch, the number of simultaneous SVC messages allowed is larger; thus the threading levels of the SVC task are higher than the SVC task of the Gateway. At the Core Switch, the SVC task re-

quests call routing information for the CTL task, configures endpoint information on two network line cards, and interacts with the originating Gateway's SVC task. The CTL and NLC tasks are identical to those on the Gateways.

5.4 Model Solution Results

The first use of the case study model is to compare the results obtained from the OBA with those of the layered model solver *lqns*. We are primarily interested in the per-task and processor utilization figures, as these are the inputs to the bottleneck location algorithm. The per-task utilization figures from the OBA and *lqns* are shown in Table 5.1, and the processor utilization figures are shown in Table 5.2.

Comparing the results of the OBA and *lqns* solution, we see that the processor utilization figures are consistent, with the OBA overestimating the utilization of each processor by approximately 7%. This is due to a 7% overestimation in the saturation throughput, f_1 . We do not believe that the variation in the differences amongst the processors is significant; we attribute this to the stochastic underpinnings of the *lqns* model and solution.

The per-task utilization figures are more interesting! For most tasks, the OBA underestimates the task utilization as it does not take any blocking into account, short of the phase two adjustment. The tasks that show a large difference in utilization between the OBA and *lqns* solution are active servers that have a non-zero mean delay for a rendezvous. However, the OBA does show the correct trends,

Task	OBA utilization	lqns utilization	% Difference
ss7IpIn	0.0665	1.000	
ss7IpOut	0.0550	0.0508	-7.64
h248Ip	0.0885	0.113	27.7
coco	1.000	1.000	0
dbServer	0.0206	0.0191	-7.28
gw1Control	0.175	0.172	-1.71
gw1Svc	0.0736	0.0979	33.0
gw1H248	0.154	0.213	38.3
gw1Caco	0.388	0.509	31.2
gw1Nlc	0.0550	0.0508	-7.64
gw1Ilc	0.0275	0.0254	-7.64
coreControl	0.175	0.214	22.3
coreSvc	0.00824	0.0114	38.3
coreNlc	0.0275	0.0348	26.5
gw2Control	0.175	0.177	1.14
gw2Svc	0.0258	0.0338	31.0
gw2H248	0.0087	0.0113	29.9
gw2Caco	0.0226	0.0261	15.5
gw2Nlc	0.0549	0.0562	2.37
gw2Ilc	0.0275	0.0290	5.45

Table 5.1: Case study per-task utilization figures from the OBA and lqns.

Processor	OBA Utilization	lqns Utilization	% Difference
ss7Ip	0.00989	0.00914	-7.58
H248Ip	0.0152	0.0141	-6.62
dbServer	0.0206	0.0191	-7.28
coco	0.0907	0.0838	-7.61
gw1Control	0.00989	0.00914	-7.58
gw1Svc	0.115	0.107	-6.96
gw1H248	0.0881	0.0831	-5.68
gw1Nlc	0.1165	0.152	-7.88
gw1Ilc	0.165	0.152	-7.88
coreControl	0.00989	0.00914	-7.58
coreSvc	0.181	0.167	-7.73
coreNlc	0.330	0.305	-7.58
gw2Control	0.00989	0.00914	-7.58
gw2Svc	0.181	0.168	-7.18
gw2H248	0.0989	0.0915	-7.48
gw2Nlc	0.165	0.152	-7.88
gw2Ilc	0.165	0.152	-7.88

Table 5.2: Case study processor utilization figures from the OBA and lqns.

with the Coco task being the one that stands out from a utilization point of view.

The ss7IpIn task is an anomaly, as it is the only task for which the OBA dramatically underestimates utilization. Knowing that the only server that ss7IpIn interacts with is Coco, and that the Coco task has a utilization of 1.00, it is reasonable to expect the ss7IpIn task to also have a utilization of 1.00. The OBA does not take blocking into account; from the point of view of the ss7IpIn task, there is always a Coco thread ready to accept a rendezvous. As such, the OBA does not accurately model “push-back”.

5.4.1 Bottleneck Identification

For the bottleneck identification, we use the procedure outlined in Section 3.1. We set U_{sat} to 75%, yielding the Coco task as the only task in the system that is saturated. The set Θ is {h248Ip, dbServer, ss7Ip, Coco’s processor}. Utilization figures from the OBA for this set are {0.0885, 0.0206, 0.0550, 0.0907}. Thus, the OBA gives a bottleneck strength $B_{coco,OBA} = 11.0$. As a comparison, using the lqns results yields a bottleneck strength of $B_{coco,lqns} = 8.85$. Note that using the OBA, we find that the Coco task is a strong processor supported software bottleneck, and using the lqns results, we find the Coco tasks is a strong server supported software bottleneck.

Using the reasoning from the section above, we should also consider the ss7IpIn task in Φ ; due to the “push-back” from the Coco task, $B_{ss7IpIn} = 1$.

Again, we emphasize that we are using the approximation only to identify the software bottleneck and to obtain a rough estimate of it’s strength. We consider

the OBA to be a “good enough” approximation in this case as it has led us to the correct bottleneck, and given us a reasonable estimate of the bottleneck’s strength.

5.4.2 Bottleneck Mitigation

We now use the methods of Section 3.3 to try to remove the bottleneck that we have found at the Coco task. Since $B_{coco} > 2$, we follow the recommendation to increase the threading levels at the bottleneck task. It is suggested that we increase the number of threads at the Coco task by a factor equal to the bottleneck strength. Using the result from the OBA, the recommended number of threads is 110. Strictly speaking, the threading rule of thumb suggests that the maximum number of useful threads at the Coco task is $M_{db} + M_{SS7OutIp} + M_{H248Ip} + 1$, or $1 + 150 + 120 + 1 = 272$. Looking at the system in the configuration presented here, it is clear the the threading levels at these tasks are too high, given the levels at lower tasks.

Inserting this value into the OBA and recomputing gives the results in Table 5.3. From these results, we see that the new bottleneck task is gw1Caco, with a bottleneck strength of $B_{caco} = 2.22$. We note that after step1, the OBA actually gives a utilization estimate for gw1Caco that is below of U_{sat} of 75%. However, we know that the OBA underestimates utilization, so we continue with the bottleneck strength calculation and bottleneck mitigation. Once again, the bottleneck strength is greater then 2, so we now increase the threading levels at the bottleneck task, giving a recommended number of threads at the gw1Caco task equal to 9. The OBA results from the resulting model are shown in Table 5.4.

Task	Utilization Estimate
Coco	0.151
H248Ip	0.142
gw1Caco	0.643
gw1Svc	0.122
gw1Control	0.290

Processor	Utilization Estimate
Coco	0.150
Gw1H248	0.146

Table 5.3: Selected Utilization Results from Step 1 of the Bottleneck Mitigation, obtained from the OBA.

We note that the utilization estimates do not change between steps 1 and 2, with the exception of the gw1Caco task, which has had an increase in its threading levels. Furthermore, we notice that the system has become “source-limited”; there is not enough offered load from the user level. This is a desired position!

5.5 Validation With the Analytic Solver

As a check against the OBA results, 1qns solutions were obtained for both steps 1 and 2, and selected results are shown in Tables 5.5 and 5.6. On first glance, the utilization figures are startlingly different from the figures from the OBA. However, once the effect of push-back is taken into account (which the OBA does not), the figures are not too different. From the 1qns solution, we find $B_{caco} = 3.60$ after step 1.

While it is true that the actual values obtained from the OBA and 1qns differ,

Task	Utilization Estimate
Coco	0.151
H248Ip	0.142
gw1Caco	0.286
gw1Svc	0.122
gw1Control	0.290

Processor	Utilization Estimate
Coco	0.150
Gw1H248	0.146

Table 5.4: Selected Utilization Results from Step 2 of the Bottleneck Mitigation, obtained from the OBA.

Task	Utilization Estimate
Coco	0.986
H248Ip	0.998
gw1Caco	0.953
gw1Svc	0.185
gw1Control	0.265

Processor	Utilization Estimate
Coco	0.125
Gw1H248	0.124

Table 5.5: Selected Utilization Results from Step 1 of the Bottleneck Mitigation, obtained from lqns.

Task	Utilization Estimate
Coco	0.182
H248Ip	0.310
gw1Caco	0.635
gw1Svc	0.278
gw1Control	0.340

Processor	Utilization Estimate
Coco	0.150
Gw1H248	0.148

Table 5.6: Selected Utilization Results from Step 2 of the Bottleneck Mitigation, obtained from lqns.

the OBA allows the designer to obtain the correct insight into the system performance. The OBA results, coupled with the bottleneck mitigation strategies led to the elimination of the two software bottlenecks in the system under study. One further note on the OBA is in regards to the ease of solution: for this case study, the OBA calculations were performed with a simple Java based application. This eases the exploration of options; for example, practical threading configuration spaces can be traversed quickly. This is compared to lqns, where the Step 1 and Step 2 model solutions each took about 90 minutes on an otherwise idle high-end engineering workstation.

Chapter 6

Conclusions

6.1 Contributions

The general contribution of this thesis is the extension of a number of existing methods for the detection, diagnosis, and mitigation of software bottlenecks. The specific contributions of this thesis are:

1. The algorithm for the identification of a software bottleneck. An informal method, using a set of definitions and examples, was presented in [5]. Section 3.1 uses these definitions in the formalization of the algorithm for the identification of a software bottleneck. This algorithm is used throughout the thesis to locate software bottlenecks.
2. Updates to the definitions for the definition of a software bottleneck. When the software bottleneck was defined in [5], the forwarding interaction was not defined. In Section 3.1, modifications to the software bottleneck strength

computation are presented, namely the inclusion of both forwarding server and the replying server in the set of tasks Θ .

3. A set of software bottleneck mitigation techniques. Section 3.3 presents the following four software bottleneck mitigation techniques: 1) increasing threading levels at the bottleneck task, 2) replication of the bottleneck task and its processor, 3) the reduction of the bottleneck task's phase one processor demands and 4) the decreasing of the number of interactions that the bottleneck task has with lower layers. Of these, the first and third mitigation technique have been briefly discussed in the literature. For all four mitigation techniques, an estimate of the reduction of the bottleneck strength is provided. These estimates can be used to guide the designer to the optimal mitigation technique without the need for further model solution. In defining the mitigation techniques, the classification of software bottlenecks into two types, processor supported and server supported, is considered in Section 3.2. Previous literature on software bottlenecks treaded all bottlenecks uniformly.
4. A simple method for the approximation of task utilization in a layered queueing system. The OBA of Section 3.4 is presented as a simple method for estimating task utilization. This approximation is based on a number of previous techniques; the main contribution of the OBA is the treatment of tasks with multiple entries, tasks with a multiplicity greater than one, and processors hosting multiple tasks. This extends the utility of the pre-

vious approximation techniques, allowing the approximation of more realistic systems. Chapter 4 is devoted to the exploration of the limitations of the OBA. This chapter gives a number of conditions under which the OBA produces a useable solution. The main focus of the OBA is the estimation of task utilization for bottleneck identification.

5. A case study of an industrial system is used to demonstrate the utility of the above contributions. The model of the distributed telephone switch, presented in Chapter 5 uses the OBA to estimate task utilization, followed by the software bottleneck location algorithm to identify software bottlenecks. The mitigation techniques are then put to use to improve system performance. A short comparison between the results from the OBA and the analytic solver *lqns* is presented; for the case study presented, the *lqns* solution validates the improvements derived from the OBA results.

6.2 Future Research

Future work may be done in the following two areas:

1. Further investigations into software bottleneck mitigation techniques. The four mitigation techniques developed in this thesis just scratch the surface. More work is required to fully understand the limitations of each mitigation technique. As well, better estimates on potential improvement for each technique would make them more useable. Finally, other software bottleneck

mitigation techniques not discussed in this thesis, such as the modification of layers below the bottleneck, are possible.

2. The automation of design optimization. Various authors (Hesham [39], and Jogalekar [40]) have suggested heuristic-based design optimization techniques. The bottleneck location and the bottleneck mitigation techniques of this thesis could be viewed as heuristics in such design optimization techniques. Tool-based design optimization suggestions are of value to the software design community; with more intelligence in the tools, the modeler can then concentrate on model development rather than batches of experiments.

Bibliography

- [1] Connie U. Smith, *Performance Engineering of Software Systems*, Addison-Wesley, 1990.
- [2] J. Buzen, "Fundamental laws of computer system performance," in *Proceedings of SIGMETRICS'76*, Cambridge, MA, 1976, pp. 200 – 210.
- [3] C. Sauer, E. MacNair, and J. Kurose, "The research queueing package: Past, present and future," in *Proceeding of the National Computer Conference*, 1982, pp. 273 – 280.
- [4] C. Murray Woodside, John E. Neilson, Dorina C. Petriu, and Shikharesh Majumdar, "The stochastic rendezvous network model for performance of synchronous client-server-like distributed systems," *IEEE Transactions on Computers*, vol. 44, no. 1, pp. 20 – 34, Jan. 1995.
- [5] J. E. Neilson, C. M. Woodside, D. C. Petriu, and S. Majumdar, "Software bottlenecking in client-server systems and rendezvous networks," *IEEE Transactions on Software Engineering*, vol. 21, no. 9, pp. 776 – 782, Sept. 1995.

- [6] Mark H. Klein, "State of the practice report: Problems in the practice of software engineering," Tech. Rep. CMU/SEI-95-TR-020, Software Engineering Institute, Carnegie Mellon University, Feb. 1996.
- [7] S. Majumdar, C. M. Woodside, J. E. Neilson, and D.C. Petriu, "Performance bounds for concurrent software with rendezvous," *Performance Evaluation*, vol. 13, no. 1, pp. 207 – 236, 1991.
- [8] Edward Whalen and Mitchell Schroeter, *Oracle Performance Tuning*, Addison-Wesley, 2002.
- [9] Patrick Killelea, *Web Performance Tuning*, O'Reilly and Associates, 2002.
- [10] Pete Utton and Gino Martin, "Further experiences with software performance modelling," in *Proceeding of the First International Workshop on Software and Performance, WOSP'98*, Santa Fe, New Mexico, USA, Oct. 1998, pp. 14 – 15.
- [11] Svend Frølund and Pankaj Garg, "Design-time simulation of a large-scale distributed object system," Tech. Rep. HPL-98-158, Software Technology Laboratory, Hewlet-Packard Copmany, Sept. 1998.
- [12] Christiane Shousha, Dorina Petriu, Anant Jalnapurkar, and Kennedy Ngo, "Applying performance modelling to a telecommunication system," in *Proceeding of the First International Workshop on Software and Performance, WOSP'98*, Santa Fe, New Mexico, USA, Oct. 1998, pp. 1 – 6.

- [13] Dorin C. Petriu and C. M. Woodside, "Software performance models from system scenarios in use case maps," in *Proceedings of Tools02. 2002*, vol. 2324 of *Lecture Notes in Computer Science*, pp. 1141 – 1158, Springer Verlag.
- [14] C. M. Woodside, C. Hrischuk, B. Selic, and S. Brayarov, "Automated performance modeling of software generated by a design environment," *Performance Evaluation*, vol. 45, no. 2 - 3, pp. 107 – 123, 2001.
- [15] Gregory R. Andrews, *Foundations of Multithreaded, Parallel, and Distributed Programming*, Addison-Wesley, 2000.
- [16] R. Srinivasan, *RFC 1831 - RPC: Remote Procedure Call Protocol Specification Version 2*, IETF, Aug. 1995.
- [17] Roy Gregory Franks, *Performance Analysis of Distributed Server Systems*, Ph.D. thesis, Carleton University, 1999.
- [18] C. M. Woodside, S. Majumdar, J. E. Neilson, D. C. Petriu, J. A. Rolia, A. Hubbard, and R. G. Franks, "A guide to performance modelling of distributed client-server software systems with layered queueing networks," Tech. Rep. SCE-95-23, Department of Systems and Computer Engineering, Carleton University, Nov. 1995.
- [19] Simonetta Balsamo, Aninisca Di Marco, Paola Inverardi, and Marta Simeoni, "Software performance: State of the art and perspectives," Jan. 2003.

- [20] Raj Jain, *The Art of Computer Systems Performance Analysis*, John Wiley & Sons, Inc., 1991.
- [21] Leonard Kleinrock, *Queueing Systems Volume 2: Computer Applications*, John Wiley & Sons, 1976.
- [22] P. Denning and J. Buzen, "The operational analysis of queueing network models," *Computing Surveys*, vol. 10, no. 3, Sept. 1978.
- [23] Gianfranco Balbo and Giuseppe Serazzi, "Asymptotic analysis of multiclass closed queueing networks: Common bottlenecks," *Performance Evaluation*, vol. 26, no. 1, pp. 51 – 72, July 1996.
- [24] Gianfranco Balbo and Giuseppe Serazzi, "Asymptotic analysis of multiclass closed queueing networks: Multiple bottlenecks," *Performance Evaluation*, vol. 30, no. 3, pp. 115 – 152, Sept. 1997.
- [25] S. Majumdar, C. M. Woodside, J. E. Neilson, and D. C. Petriu, "Robust box bounds: Throughput guarantees for closed multiclass queueing networks with minimal stochastic assumptions," in *Proceeding of IEEE Infocom'92*, Florence, May 1992, pp. 2006 – 2015.
- [26] Connie U. Simth and Lloyd G. Williams, "Software performance antipatterns," in *Proceedings of the Second International Workshop on Software and Performance, WOSP'2000*, Ottawa, Ontario, Canada, Sept. 2000, pp. 127 – 136a.

- [27] Greg Hills, Jerome Rolia, and Giuseppe Serazzi, "Performance engineering of distributed software process architectures," in *8th International Conference on Modelling Techniques and Tools for Computer Performance Evaluation '95*, Heidelberg, Germany, Sept. 1995.
- [28] Marin Litoiu, Jerome Rolia, and Giuseppe Serazzi, "Designing process replication and activation: A quantitative approach," *IEEE Transactions on Software Engineering*, vol. 26, no. 12, pp. 1168 – 1178, Dec. 2000.
- [29] C. M. Woodside and Students, "Some results on software patterns and performance," Tech. Rep. SCE-97-04, Department of Systems and Computer Engineering, Carleton University, Mar. 1997.
- [30] J. A. Rolia and K. C. Sevcik, "The method of layers," *IEEE Transactions on Software Engineering*, vol. 21, no. 8, pp. 689 – 700, Aug. 1995.
- [31] Richard Singer, "Overview of 5ESS®-2000 switch performance," in *Proceeding of the First International Workshop on Software and Performance, WOSP '98*, Santa Fe, New Mexico, USA, Oct. 1998, pp. 7 – 13.
- [32] Telcordia Technologies, "Voice over packet in next generation networks: An architectural framework," Special Report SR-4717, Jan. 1999.
- [33] Canadian Radio and Telecommunications Commission, "Telecom decision 97-16: Quality of service indicators for use in telephone company regulation," July 1997.

- [34] Telcordia Technologies, "LSSGR: service standards," Generic Requirement GR-511-CORE, June 2000.
- [35] Telcordia Technologies, "LSSGR: traffic capacity and environment," Generic Requirement GR-517-CORE, Dec. 1998.
- [36] Travis Russell, *Signaling System # 7*, McGraw-Hill, 1998.
- [37] International Telecommunication Union, "ITU recommendation H.248: Gateway control protocol," Mar. 2002.
- [38] Pekka Kähkipuro, *Performance Modeling Framework for CORBA Based distributed Systems*, Ph.D. thesis, University of Helsinki, 2000.
- [39] Hesham El-Sayed, *A Framework for Automated Performance Engineering of Distributed Real-Time Systems*, Ph.D. thesis, Carleton University, 1999.
- [40] Prasad Jogalekar, *Scalability Analysis Framework for Distributed Systems*, Ph.D. thesis, Carleton University, 1997.

Appendix A

LQN Input File for Case Study

```
G
"Case Study"
0.0001
200
-1

P 0
p class5Switch.1 f i
p class5Switch.2 f i

# Processors in the Call Connection Agent

p ss7Ip f
p h248Ip f
p dbServer f
p coco.1 f

# Processors in the Gateway
```

```

p gwControl.1 f
p gwSvc.1 f
p gwH248.1 f
p gwNlc.1 f
p gwIlc.1 f

p gwControl.2 f
p gwSvc.2 f
p gwH248.2 f
p gwNlc.2 f
p gwIlc.2 f
# Processors in the Core Switch

p coreControl f
p coreSvc f
p coreNlc f
-1

T 0
t class5SwitchT.1 r class5SwitchE.1 -1 class5Switch.1 m 1000
t class5SwitchT.2 n class5SwitchIAM.2 class5SwitchREL.2 -1 class5Switch.2 m 500

t ss7IpInT n ss7IpInIamE ss7IpInRelE -1 ss7Ip m 150
t ss7IpOutT n ss7IpOutIamE ss7IpOutRelE -1 ss7Ip m 150
t h248IpT n h248IpAddE h248ModifyE h248SubtractE -1 h248Ip m 20
t cocoT.1 n cocoSetupE.1 cocoReleaseE.1 -1 coco.1 m 10
t dbServerT n dbServerGetE dbServerSetE -1 dbServer

# One control, one Voice CallP, one SVC, 3 network line, 6 interface line
t gwControlT.1 n gwControlRouteRequestE.1 gwControlDisconnectE.1 -1 gwControl.1
t gwSvcT.1 n gwSvcSetupE.1 gwSvcConnectE.1 gwSvcReleaseE.1 -1 gwSvc.1 m 20
t gwH248T.1 n gwH248AddE.1 gwH248ModifyE.1 gwH248SubtractE.1 -1 gwH248.1 m 10
t gwCacoT.1 n gwCacoAddE.1 gwCacoModifyE.1 gwCacoSubtractE.1 -1 gwH248.1 m 4
t gwNlcT.1 n gwNlcPt2PtSetE.1 -1 gwNlc.1
t gwIlcT.1 n gwIlcPt2PtSetE.1 -1 gwIlc.1

```

```

# One control, one Voice CallP, one SVC, 3 network line, 6 interface line
t gwControlT.2 n gwControlRouteRequestE.2 gwControlDisconnectE.2 -1 gwControl.2
t gwSvcT.2 n gwSvcSetupE.2 gwSvcConnectE.2 gwSvcReleaseE.2 -1 gwSvc.2 m 20
t gwh248T.2 n gwh248AddE.2 gwh248ModifyE.2 gwh248SubtractE.2 -1 gwh248.2 m 10
t gwCacoT.2 n gwCacoAddE.2 gwCacoModifyE.2 gwCacoSubtractE.2 -1 gwh248.2 m 4
t gwnlcT.2 n gwnlcPt2PtSetE.2 -1 gwnlc.2 m 3
t gwilcT.2 n gwilcPt2PtSetE.2 -1 gwilc.2 m 6

# One control, one svc, 12 line cards
t coreControlT n coreControlRouteRequestE coreControlDisconnectE -1 coreControl
t coreSvcT n coreSvcSetupE coreSvcConnectE coreSvcReleaseE -1 coreSvc m 125
t coreNlcT n coreNlcPt2PtSetE -1 coreNlc m 12

-1

E 0

# Gateway 1 is the terminating gateway
# Gateway 2 is the originating gateway

# we have 500 000 users, each making 1 call per hour. That means that
# there is 3600 seconds of sleep time
# for practical reasons, we have 1000 users, with 7.2 seconds of sleep time

s class5SwitchE.1 0 3.6 3.6 -1
y class5SwitchE.1 ss7IpInIamE 0 0 1 -1
y class5SwitchE.1 ss7IpInRelE 0 1 0 -1

# set the service time for the IAM at the terminating switch to
# be 50 msec
s class5SwitchIAM.2 0.050 0 -1
s class5SwitchREL.2 0.050 0 -1

s ss7IpInIamE 0.000030 0 -1

```



```
y ss7IpInIamE cocoSetupE.1 1 0 -1

s ss7IpInRelE 0.000030 0 -1
y ss7IpInRelE cocoReleaseE.1 1 0 -1

s ss7IpOutIamE 0.000030 0 -1
y ss7IpOutIamE class5SwitchIAM.2 1 0 -1

s ss7IpOutRelE 0.000030 0 -1
y ss7IpOutRelE class5SwitchREL.2 1 0 -1

s h248IpAddE 0.000050 0 -1
y h248IpAddE gwh248AddE.1 0.5 0 -1
y h248IpAddE gwh248AddE.2 0.5 0 -1

s h248ModifyE 0.000025 0 -1
y h248ModifyE gwh248ModifyE.2 1 0 -1

s h248SubtractE 0.000030 0 -1
y h248SubtractE gwh248SubtractE.1 0.5 0 -1
y h248SubtractE gwh248SubtractE.2 0.5 0 -1

s cocoSetupE.1 0.000800 0.000050 -1
y cocoSetupE.1 h248IpAddE 2 0 -1
y cocoSetupE.1 h248ModifyE 1 0 -1
y cocoSetupE.1 dbServerGetE 1 0 -1
y cocoSetupE.1 dbServerSetE 0 1 -1
y cocoSetupE.1 ss7IpOutIamE 1 0 -1

s cocoReleaseE.1 0.000200 0.000050 -1
y cocoReleaseE.1 h248SubtractE 2 0 -1
y cocoReleaseE.1 dbServerGetE 1 0 -1
y cocoReleaseE.1 dbServerSetE 0 1 -1
y cocoReleaseE.1 ss7IpOutRelE 1 0 -1
```

```
s dbServerGetE 0.000025 0 -1

s dbServerSetE 0.000100 0 -1

s gwControlRouteRequestE1 0.000040 0 -1

s gwControlDisconnectE1 0.000080 0 -1
y gwControlDisconnectE1 gwNlcPt2PtSetE1 1 0 -1
y gwControlDisconnectE1 gwIlcPt2PtSetE1 1 0 -1

s gwSvcSetupE1 0.000800 0.000050 -1
y gwSvcSetupE1 gwControlRouteRequestE1 1 0 -1
y gwSvcSetupE1 coreSvcSetupE 1 0 -1
y gwSvcSetupE1 gwNlcPt2PtSetE1 1 0 -1
y gwSvcSetupE1 gwIlcPt2PtSetE1 1 0 -1
y gwSvcSetupE1 coreSvcConnectE 1 0 -1

s gwSvcReleaseE1 0.000500 0.000050 -1
y gwSvcReleaseE1 gwControlDisconnectE1 1 0 -1
y gwSvcReleaseE1 coreSvcReleaseE 1 0 -1

s gwSvcConnectE1 0.000750 0.000050 -1

s gWH248AddE1 0.000020 0 -1
y gWH248AddE1 gwCacoAddE1 1 0 -1

s gWH248ModifyE1 0.000010 0 -1
y gWH248ModifyE1 gwCacoModifyE1 1 0 -1

s gWH248SubtractE1 0.000020 0 -1
y gWH248SubtractE1 gwCacoSubtractE1 1 0 -1

s gwCacoAddE1 0.000800 0.000100 -1
y gwCacoAddE1 gwSvcSetupE1 1 0 -1
```

```
s gwCacoModifyE.1 0.000100 0.000050 -1

s gwCacoSubtractE.1 0.000100 0.000050 -1
y gwCacoSubtractE.1 gwSvcReleaseE.1 1 0 -1

s gwNlcPt2PtSetE.1 0.001000 0 -1

s gwIlcPt2PtSetE.1 0.001000 0 -1

#End of Gateway number 1

#Start of Gateway number 2
s gwControlRouteRequestE.2 0.000040 0 -1

s gwControlDisconnectE.2 0.000080 0 -1
y gwControlDisconnectE.2 gwNlcPt2PtSetE.2 1 0 -1
y gwControlDisconnectE.2 gwIlcPt2PtSetE.2 1 0 -1

s gwSvcSetupE.2 0.000800 0.000050 -1
y gwSvcSetupE.2 gwControlRouteRequestE.2 1 0 -1
y gwSvcSetupE.2 gwNlcPt2PtSetE.2 1 0 -1
y gwSvcSetupE.2 gwIlcPt2PtSetE.2 1 0 -1

s gwSvcReleaseE.2 0.000500 0.000050 -1
y gwSvcReleaseE.2 gwControlDisconnectE.2 1 0 -1

s gwSvcConnectE.2 0.000750 0.000050 -1

s gWH248AddE.2 0.000020 0 -1
y gWH248AddE.2 gwCacoAddE.2 1 0 -1

s gWH248ModifyE.2 0.000010 0 -1
y gWH248ModifyE.2 gwCacoModifyE.2 1 0 -1

s gWH248SubtractE.2 0.000020 0 -1
```

```
y gwH248SubtractE2 gwCacoSubtractE2 1 0 -1

s gwCacoAddE2 0.000800 0.000050 -1

s gwCacoModifyE2 0.000100 0.000050 -1

s gwCacoSubtractE2 0.000100 0.000050 -1
s gwNlcPt2PtSetE2 0.001000 0 -1
s gwIlcPt2PtSetE2 0.001000 0 -1

# End of Gateway number 2

# start of the core switch
s coreControlRouteRequestE 0.000040 0 -1

s coreControlDisconnectE 0.000080 0 -1
y coreControlDisconnectE coreNlcPt2PtSetE 2 0 -1

s coreSvcSetupE 0.000800 0.000050 -1
y coreSvcSetupE coreControlRouteRequestE 1 0 -1
y coreSvcSetupE coreNlcPt2PtSetE 2 0 -1
y coreSvcSetupE gwSvcSetupE2 1 0 -1

s coreSvcConnectE 0.000750 0.000050 -1
y coreSvcConnectE gwSvcConnectE2 1 0 -1

s coreSvcReleaseE 0.000500 0.000050 -1
y coreSvcReleaseE coreControlDisconnectE 1 0 -1
y coreSvcReleaseE gwSvcReleaseE2 1 0 -1

s coreNlcPt2PtSetE 0.001000 0 -1
-1
```