INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each original is also photographed in one exposure and is included in reduced form at the back of the book.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

UMI

A Bell & Howell Information Company
300 North Zeeb Road, Ann Arbor MI 48106-1346 USA
313/761-4700 800/521-0600

Applying Layered Modelling to a Bridge/ Router Architecture

By: Peter Maly

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

Master of Engineering

Ottawa-Carleton Institute for Electrical Engineering
Faculty of Engineering

Department of Systems and Computer Engineering
Carleton University
Ottawa, Ontario, Canada, K1S 5B6

August 29, 1997

© Copyright 1997, Peter Maly

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

Our file Notre référence

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.

0-612-27002-5

Canada

The undersigned recommend to the Faculty of Graduate Studies and
Research acceptance of the thesis

Applying Layered Modelling to a Bridge/Router Architecture

submitted by Peter Maly, B. Eng.
in partial fulfillment of the requirements for
the degree of Master of Engineering

Chair, Department of Systems and Computer Engineering

Thesis Supervisor

Carleton University
August 29, 1997

Abstract

Designing real-time embedded computer systems to meet certain performance criteria is a significant problem. Not knowing where the performance problems in the system are, make it very costly to try to solve them. A methodology is proposed in this thesis that systematically looks at the relevant parts of the hardware and software architecture, to develop a layered performance model of the system. Once this model is derived, problem areas can be identified, and performance questions can be answered.

This thesis demonstrates the use of this performance methodology on a case study of the Expressway RLAN system. It demonstrates the application of the methodology step by step, and then shows examples of how the model can be exploited. Examples include using different traffic sources, and a hypothetical re-design of the system.

Acknowledgments

I would like to thank my supervisor Murray Woodside for guiding me through this research. I also thank Alex Hubbard, Greg Franks, Curtis Hrischuk, and others in the RADS lab who helped me in one way or another during various stages of this work.

I thank the following people at Gandalf Technologies for their help and support at the beginning of the thesis work:

Alan Spurdle, Ralph Mask, Zoltan Racz, Chuck Triemstra, Sukhvinder Parmar, Winga Ho, Mick Chawner, Kulrattan Bakhshi.

Financial support from Gandalf Technologies, and Software Performance Engineering Chair was greatly appreciated.

Last but not least I thank my parents for their understanding and support throughout the whole Masters program.

Table Of Contents

Abstract	I
Acknowledgments	II
Table Of Contents	III
List Of Figures	VII
List Of Symbols	X
Chapter 1:	
Introduction	1
1.1 The Purpose Of Performance Modelling	1
1.2 Motivation And Perspective	2
1.3 Problem Statement	6
1.3.1 Goals Of Thesis	7
1.3.2 Contributions Made By Thesis	7
Chapter 2:	
Background	9
2.1 Layered Queuing Network Modelling	9
2.2 Use Case Maps	12
2.3 Real-Time Embedded Systems	14
2.3.1 Obtaining Measurements From Real-Time Embedded Systems	15
2.4 Real-Time Embedded System Example: Bridges And Routers	17
2.4.1 Implementation Issues Of Bridges And Routers	19
2.5 Communication Protocols	21

Chapter 3:	
Outline of a methodology for modelling embedded systems	23
3.1 A Methodology For Modelling Embedded Real-Time Systems	23
3.2 Comparison Of Presented Methodology With Connie Smith's	29
 Chapter 4:	
Description of an embedded system	32
4.1 High Level Description Of System To Be Studied	32
4.2 Description Of The Hardware Architecture	33
4.3 Description Of The Software Architecture	38
 Chapter 5:	
Building a Base Model of the RLAN	45
5.1 Determining Scenarios In The RLAN System	45
5.2 Mapping Scenarios In The System To A Layered Model	51
5.3 Adding Behavioral Detail To Model	56
5.3.1 Main Bus Arbitration	56
5.3.2 Adding Detail To The Munich Chip	58
5.3.3 Polling Delay And Polling Overhead In Port Worker	59
5.3.4 Concurrency In Scenarios And Scaling Of The Model	61
5.3.5 Accounting For Munich Throttling	62
5.4 Calculation To Account For Munich Chip Polling	64
5.5 Calculation To Account For Polling By Software	66
5.6 Simplifications of the model	71
5.7 Summary	74

Chapter 6:	
Measurement Gathering And Model Validation	75
6.1 The CodeTEST Profiling Tool	75
6.1.1 Instrumenting in a RTOS	77
6.2 Goals Of Measurement Experiments Performed On The RLAN	78
6.3 Measurement Setup	79
6.4 Approach Taken To Make Measurements	81
6.5 Measurement Results For Software Functions	85
6.6 Measurement Results For Hardware Operations	85
6.7 Comparing Model Predictions With Real Observations	87
6.8 Discussion Of Utilization Results	90
6.9 Summary	92
 Chapter 7:	
Exploiting Model For Capacity Planning And Architectural Re-design	94
7.1 Goals Of The Performance Study	94
7.2 Predicting Capacity Of The Expressway RLAN With Poisson Arrivals	95
7.3 Polling Discipline	99
7.4 Selective Server	103
7.5 Bursty traffic analysis	104
7.5.1 Definition Of Three Traffic Profiles	105
7.5.2 Bursty Reference Task	107
7.6 Using New ParaSRVN Extensions To Model The RLAN	110
7.6.1 Defining The Port Worker Using The Poll Server	110
7.6.2 Comment On Model Modularity	112

7.7	Results Using Three Different Bursty Traffic Profiles	113
7.8	Approximations For Faster Solution Times With Bursty Traffic	117
7.9	Hypothetical Hardware Re-design Of The RLAN	120
7.10	Summary	126
 Chapter 8:		
	Discussion and Conclusions	127
8.1	Summary Of Results	127
8.2	Conclusions	127
8.3	Future Research	129
	 References	 131

List Of Figures

FIGURE 1.	Example of a layered model	11
FIGURE 2.	Illustrating some Use Case Map notation	13
FIGURE 3.	Example situation where a router and bridge would be used	19
FIGURE 4.	High level diagram of the RLAN's major input and output ports	32
FIGURE 5.	Hardware interconnection in the RLAN system	33
FIGURE 6.	Priority built into arbiter	36
FIGURE 7.	High level state machine diagram of the Munich chip arbiter	37
FIGURE 8.	High level software/hardware diagram of the Expressway RLAN	39
FIGURE 9.	High level pseudo code describing the Port Worker polling order	40
FIGURE 10.	Task scheduling employed in the RLAN	43
FIGURE 11.	Use Case Map of a packet going from LAN to WAN, singlelink	48
FIGURE 12.	Use Case Map of a frame going from WAN to LAN, singlelink	50
FIGURE 13.	Grouping of responsibilities to resources	52
FIGURE 14.	Addition of another layer to represent the main bus.	53
FIGURE 15.	Grouping responsibilities for the reception of an acknowledgment	54
FIGURE 16.	Merging the two models derived from the two scenarios	54
FIGURE 17.	Addition of two more scenarios to model	55
FIGURE 18.	Model with a more representative bus arbitration scheme	57
FIGURE 19.	With added detail of the Munich chip	59
FIGURE 20.	Model with software polling.	60
FIGURE 21.	Final structure of the RLAN model	62
FIGURE 22.	Polling orders for frames and packets in Port Worker	70

FIGURE 23. Simplified model of the RLAN system	73
FIGURE 24. Instrumentation process	76
FIGURE 25. Data gathering when code is running	77
FIGURE 26. Measurement setup in the lab	80
FIGURE 27. Histogram of a function being called in an experiment	81
FIGURE 28. Process trace data goes through to generate an execution profile	83
FIGURE 29. An example of a execution profile	83
FIGURE 30. Diagram of experiment setup for model validation	88
FIGURE 31. Defining operating range for an open system	94
FIGURE 32. Average waiting time experienced in the WAN queue	96
FIGURE 33. Average waiting time experienced at the LAN queue	97
FIGURE 34. Re-definition of LQN parameters for the polling discipline	101
FIGURE 35. Polling order implemented in the exhaustive polling discipline	103
FIGURE 36. Description of the Bursty traffic profile	105
FIGURE 37. Re-definition of LQN parameters for the Bursty client	108
FIGURE 38. Polling Task entries, note calls are asynchronous	110
FIGURE 39. A modified model of the RLAN with the new extensions	111
FIGURE 40. Another way of expressing polling in the model	113
FIGURE 41. Corporate profile, with bursts, plotting end to end delay	114
FIGURE 42. ISP profile, with bursts, plotting end to end delay	114
FIGURE 43. Worst Case, with bursts, plotting end to end delay	115
FIGURE 44. WAN to LAN delay curve with one poll cycle time reduced	116
FIGURE 45. Corporate profile, plotting delay results from 3 different models	118
FIGURE 46. ISP profile, plotting delay results from 3 different models	118

FIGURE 47. Worst Case, plotting delay results from 3 different models	119
FIGURE 48. Hypothetical hardware/software re-design of the RLAN	122
FIGURE 49. Layered model of a hypothetical hardware re-design	123
FIGURE 50. Average waiting time experienced at the LAN queue	124
FIGURE 51. Average waiting time experienced in the WAN queue	125

List Of Symbols

Symbol	Definition	Typical Usage
i	Represents an index to an active channel.	64
f	Maximum transmission speed of the DS0 channel, being 8000 bytes/sec.	64
$P(i)$	Transmission link utilization of the Munich chip on channel i .	64
V_i	Average size of the frame to be transmitted on channel i .	65
λ_i	Frames per second to be transmitted on channel i .	65
$Z_{poll}(i)$	Derived average cycle time when channel i gets polled to see if there is anything to transmit.	65
C_{munich}	Munich chip cycle time to check all of its ports to see if there is anything to transmit.	65
U_{poll}	Percentage of time the Munich chip occupies the bus with polling.	65
A	Time to check if there is something to transmit.	65
N	Number of active channels on each Munich chip. $N < 33$ because one Munich chip can only handle 32 channels.	66
$U_{polltotal}$	Percentage of time all Munich chips occupy the bus with polling.	66
j	Represents an index to a Munich chip.	66
$U_{poll}(j)$	Percentage of time the j^{th} Munich chip occupies the bus with polling.	66
M	Number of Munich chips in the system.	66
$E[C]$	Average time to complete one cycle of a polling loop.	67
k	Set of software functions to process packets/frames.	67

$d_{i,k}$	Switchover time to go from queue i,k to queue $i,k-1$. This is polling overhead of queue k .	67
$\rho_{i,k}$	Offered load of queue i,k .	67
b_k	CPU cost of servicing a job waiting in queue associated with k .	67
$\lambda_{i,k}$	Arrival rate of jobs going into queue i,k .	67
$E_{adjusted}[C]$	Average time the Port Worker takes to complete one cycle, with compensation for DMA activity.	67
U_{LAN}	Utilization fraction used by the LAN chip on the main bus copying data to/from memory.	68
U_{Munich}	Utilization fraction used by the Munich chip on the main bus copying data to/from memory.	68
U_{poll}	Utilization fraction used by the Munich chip on the main bus for polling TX descriptors.	68
B_{LAN}	Service time for the LAN chip to perform a 32 bit read/write operation.	68
B_{Munich}	Service time for the Munich chip to perform a 32 bit read/write operation.	68
V_{LAN}	Average size of the packet incoming on the LAN side.	68
V_{Munich}	Average size of frame incoming on the WAN side.	68
λ_{LAN}	Average throughput of packets incoming on the LAN side.	68
λ_{Munich}	Average throughput of frames incoming on the WAN side.	68
$E_{poll}[W]$	Average polling delay seen by frames/packets in system.	71
$E_{entry}[W]$	Average time to service a request at an entry.	71
$E_{total_delay}[W]$	Total delay seen by a packet/frame in a queue serviced by the Port Worker.	71
$\Phi_{i,k}$	The mean number of messages served per visit to queue i,k .	71

Chapter 1

Introduction

1.1 The Purpose Of Performance Modelling

Designing computer systems to meet certain performance criteria is a significant problem. Depending on measurement data from the system alone is not an effective way to ensure a computer system will meet its performance targets. To make measurements on a system, it has to be already implemented. It will cost significant amount of time and money to solve performance problems when the system is already built. Measurements alone are not capable of telling a designer where the performance problems are in the system. This is because measurements only provide a snap shot of the system performance, without any explanation to why the system is behaving a certain way. Not knowing where the problems are in the system, how can they be fixed the first time?

Performance modelling at design or re-design stage substantially increases the chances of creating a computer system that will meet performance expectations. It does this by incorporating all the relevant knowledge spread out among many engineers working on different aspects of the system. The model will then be able to tell the designers which factors most influence its performance. This will focus designing or re-designing efforts on relevant factors within the system that have significant impact on various performance concerns.

The performance model also provides other useful information like predictions of throughputs and delays, which can provide insight on system capacity. Model predictions

can give designers confidence the current design will meet the required performance expectations, or give early warnings to potential performance problems. If preliminary analysis indicates some performance problems in the design, the model can allow designers to rapidly explore alternatives to correct the problems before they arise in the implemented system.

Overall performance analysis is partly a risk reduction technique, and partly a useful engineering practice. From a business perspective, performance analysis is a risk reduction technique because it forces engineers to address performance issues before they become problems. With systems becoming more complicated everyday, it is easy to overlook certain factors which can affect performance. Therefore performance analysis is a risk reduction technique, and not a method that guarantees success all the time. Performance analysis is also useful engineering practice because it gives a global perspective on how the system works. It provides feedback on how well the design performs, which gives the designers more insight into the system they are building.

This thesis demonstrates through a case study, how performance modelling can be used to reduce risk, and show its usefulness as a engineering practice.

1.2 Motivation And Perspective

There is a wide variety of standard techniques for analyzing the performance of a computer system, for example [12], [18], [21]. Regardless of the study or solution technique, the three results normally obtained from the model are delays, utilizations, and throughputs for various resources or for the whole system. Utilization of resources can tell the per-

formance analyst where the bottleneck(s) in the system are. This knowledge will focus the efforts of the designers to work on these bottleneck(s) to either fix the performance problem in the existing system, or prevent a performance problem from happening in the future. Delay results from the model can answer performance questions about quality of service. For example it can answer how many users the system can simultaneously handle, given a metric for quality of service. Throughput results can predict how many transactions per unit time the system is capable of handling under various system loads.

Services the system is capable of performing, are invoked by external stimuli. When building a model, it is necessary to understand how the system fulfills each service to the end user. A scenario is a trace of causality through the system to determine how a transaction is completed. Performance models usually represent more than one scenario, and the number of scenarios in the model is usually determined by the scope of the performance analysis. When tracing scenarios through the system, the amount of detail captured can vary, and to some degree is dependent upon the solution approach.

Solution approaches for models usually fall into two categories, which are simulation and analytic. Models that use simulation tend to have more detail than analytic models. Many properties and interactions within a system can be captured very precisely because simulation is very general and robust. This generality comes at the cost of setting up the model, even if using a simulation package. Getting results from a simulation can take significant amount of computing resources and time until acceptable confidence intervals are achieved. Analytic models are not as general as simulation, and abstract more away from detail within the system. This means that analytic models are simpler, and are easier to

construct than simulations, assuming the analytic solution technique does not have to be invented. The greatest advantage analytic modelling possesses is, that fairly accurate results can be obtained many times faster than from a simulation.

When performance studies of computer systems are done, typically either the hardware of the system is well described, and the software is approximated, or vice versa. Performance studies on hardware architectures are usually done with simulation. The reason for this is hardware architectures that are normally modelled are too complex for an analytic solver to handle accurately. When an analytic approach is possible, a queueing network model is constructed. The queueing network model is usually solved using Mean Value Analysis (MVA), which is a relatively fast solution technique for queueing models. To obtain an exact solution, the queueing model has to obey rules that make the model product form. Sometimes the MVA rules are too restrictive in describing the system being modeled. In this case MVA can be extended by including the necessary features to better capture the elements of the system. This is done by modifying the delay equations in the MVA algorithm, therefore not guaranteeing an exact solution because the model is no longer product form. The extension to MVA is called Customized Mean Value Analysis (CMVA) [14], [24]. A weakness that the CMVA approaches have is there is no concept of simultaneous resource possession. Even the most simple hardware architectures exhibit simultaneous resource possession. For example when a CPU wants to access memory, it must acquire the bus, and then acquire memory. Not properly representing a system in a model, may mislead a performance analyst to where the true trouble spots in the system are. There are analytic techniques like [11] which can solve for simultaneous resource

possession. It does not modify the MVA algorithm in the manner as CMVA does, but still is not a product form solution. Compared to layered modelling, these types of approaches are often complicated and difficult to use when dealing with simultaneous resource possession.

A method of software performance analysis is layered modelling [27], which is an analytic method for solving such systems. Layered modeling can also account for simultaneous resource possession, which allows the model to capture mutual exclusion in software. The concept of layered modelling is to capture the software interactions from a software architecture, and then determine how these interactions pose various demands on the underlying hardware. How this hardware functions can almost always be abstracted out, because it usually is not necessary to understand it. The main reason for this is, the hardware architecture which runs the software was either simple, or provided no significant effect on the system within the scope of the analysis. Thus most layered models have not represented hardware detail.

Embedded systems which are studied in this thesis, form a class of computer systems where abstracting from the software or hardware may not always work. The reason for this is, hardware and software are intimately coupled because they are designed for each other, rather than for general use. Therefore these systems will tend to have more complex hardware architectures, because they try to deliver high performance at low cost. To achieve high performing embedded systems, designs commonly have multiple CPUs, shared bus(es), shared slave device(s), DMA, and Application Specific IC chips (ASIC). With these elements present in a system, one has to wonder how much effect do all these ele-

ments have on the execution of software. DMA activity, ASICs, and other CPUs that all share a bus will experience contention. From a software perspective, this contention translates into slower execution of code, and ultimately leads to longer execution times of scenarios within the system. Clearly ignoring the hardware architecture or accounting for it in a non-precise way will not yield to very good performance predictions. Ignoring the software architecture is not a very good strategy either. The software model will give no insight into what kinds of demands the software puts on specific areas of the hardware, and how the hardware or software can be optimized for maximum performance. Average processor throughput of a hardware architecture, which is a common hardware performance metric, is not a good enough to optimally tune hardware and software as a complete entity.

The approach taken in this thesis to modelling embedded systems is, applying layered modelling in such a way that it encompasses both the hardware and software architecture in one model. Layered modelling provides the simplicity and speed of analytic modelling, and many of the interactions that exist at the software level also exist at the hardware level. As a result of this approach, this introduces layered hardware resources.

1.3 Problem Statement

The primary goal of this thesis is to present a generic methodology to apply layered modelling to embedded systems, where including the hardware and software architecture in a performance model is equally important. This methodology will be applied to a real case study to demonstrate how to apply it, and to show how well it works.

1.3.1 Goals Of Thesis

The goals of the thesis are the following:

- Outline a methodology for applying layered modelling to embedded systems.
- Apply this methodology to a case study, which is a bridge/router embedded system.
 - Describe and derive a model of the case study.
 - Describe how measurements were done on the embedded system.
- Show how the model can be exploited for several purposes and explore faster solution techniques.

1.3.2 Contributions Made By Thesis

The contributions of this thesis are the following:

- Demonstrate a new application of Layered Queuing Network Modelling by modelling an embedded system. In previous studies using layered modelling, it has never been important to include the hardware architecture to such extent. This explores the notion of layered hardware resources, which is something new.
- Document a case study of modelling a bridge/router embedded system.
- The case study has polling at the hardware and software levels. A system with any form of a polling discipline has not been studied before with layered modelling, so this is an addition to the modelling framework.
- Addition of three new components to ParaSRVN which is a simulator for layered models. The three new components are the selective server, exhaustive polling discipline,

and bursty reference task. These components can be specified freely in a model and are automatically incorporated for simulation. They are used to refine the layered model to yield more accurate and realistic results.

Chapter 2

Background

2.1 Layered Queuing Network Modelling

Layered Queuing Network modelling [26], [27] is a convenient way of modelling distributed software client server systems. The two advantages it possesses are, it has semantics to express common interactions found in distributed client server systems, and the layered models can be solved both by simulation and analytic modelling.

One of the fundamental components in a layered model is a “task” which can represent a process, task, or object. A task is an entity that executes some work, and can only do so if it has acquired access to a processor. There are only two classes of tasks in layered modelling which are client and a server. A client task (also called a reference task) is a task which never receives any requests to do work. The server task is just the opposite of a client task because, other tasks make requests to it. Just like the client, the server optionally can make requests to other servers, but only when the server task has to service a request.

Each task can impose different work on the processor, by serving different requests which are represented by entries. An entry is a service interface offered by the server task to its clients. All the requests for each task are put into the same queue. The order in which the requests are served is dependant upon the queueing discipline selected for the task. Layered modelling supports First In First Out (FIFO), and Head Of the Line (HOL) queueing disciplines for entries.

There are three ways in which service invocations from one task to another can occur. These are by synchronous, asynchronous, and forwarding calls. An asynchronous call is one in which the client task makes a request to a server, and does not wait for a response from the server. On the other hand, a synchronous call is one in which the client makes a request to the server, and the client will do nothing until the server has responded back with a reply. The forwarding call is similar to a synchronous call, except that after the server finishes servicing the request, it will transfer the request to another server which will be responsible for sending the reply back to the client.

There are three types of servers that can exist in a layered model. The number of copies of a task on each processor(s) is what determines the server type, which can be a single, multiple, or infinite server. A single server is a single task, and can only handle one request at a time. A multiserver is a copy of N tasks that take requests from one queue, and can handle up to N requests simultaneously. Finally the infinite server is an infinite number of tasks, which can handle an infinite amount of requests simultaneously. To add a constant delay in the model, an infinite number of tasks and processors have to be specified, so that the server can handle infinite requests simultaneously and never wait for the processor.

Server tasks that receive synchronous requests may optionally implement a second phase. This is a phase of execution the server does work after sending a reply back to the client. Figure 1 shows an example of a layered model using the notation to represent the various concepts discussed.

FIGURE 1. Example of a layered model

Each entry has a service time which is specified by “s:” followed by a list of times, one for each phase. The number of service times in the list specifies the number of phases an entry has. For example E4 in figure 1 has two phases, with a service time of 0.2 in phase 1, and 0.5 in phase 2. The multiplication factor near a task indicates task multiplicity, and the multiplication factor near a request indicates number of times the entry is called. The number of visits to an entry is another way of saying the number of times an entry is called. A visit ratio is the relative number of visits made to other entries. For example entry E1 in figure 1 visits E2 once, and E3 5 times. The visit ratio for E1 visiting E2 and E3 is 1:5 which means that for every visit to E2 there will be 5 visits to E3.

For all the layered model diagrams in this thesis, three features found in figure 1 have been left out for simplicity. First simplification is that service times are not shown. Second simplification is most entries associated with a task are not shown, and only a single parallelogram is shown to represent the task. The third simplification is processors are not shown, and any time a task is copied, it is always assumed that there is an equal number of processors, unless otherwise specified.

Below is a summary of the assumptions and limitations of the layered modelling:

- synchronous, asynchronous, and forwarding call types
- random or deterministic messaging
- exponential distributed or deterministic service times
- task will always accept a request on any entry
- three different server types: single server, multiserver, infinite server.
- scheduling policies: First In First Out (FIFO), Head Of the Line (HOL), Random, Processor Sharing, and Pre-emptive Priority (PP).

2.2 Use Case Maps

This thesis will use Use Case Maps (UCM) as a means to document and describe the processing of various services in the system before a model is built. Use Case Maps [5] is a high level design notation that traces causality through a computer system given a certain set of stimuli. This trace of causality is called a scenario, and is drawn on top of a component context diagram. The component context diagram is a high level diagram, showing the major components in the computer system. Use Case Maps are drawn over a

component context diagram, to show the sequence of events that occur to complete a scenario. Components in the system are usually teams or objects. A Team represents some functionality in the system. It can consist of a group of objects, but the actual composition of the team in terms of hardware or software may not be known. When the UCM goes through a component, this means the component has some role in fulfilling a scenario. Figure 2 illustrates a typical UCM diagram.

FIGURE 2. Illustrating some Use Case Map notation

Scenarios also have the capability of forking and joining. Two types of forks that can occur in a UCM are the AND fork and OR fork. The AND fork indicates that the scenario has concurrency, which means the scenario splits into two parallel executing paths. The OR fork indicates the scenario will only execute one of possible scenarios. This does not mean the OR fork represents a logical decision, but rather a convenient way of expressing a group of scenarios that have similar causality. Similarly two joins exist in UCM notation, which are the AND join and OR join. The AND join is a synchronization point for scenar-

ios involved. No scenario can continue executing until the rest of the scenarios have reached the AND join. In figure 2, after executing responsibility D, that scenario will have to wait until responsibility B has completed, or vice versa. The OR join just indicates the merging of two scenarios. It has no logical implications, and reduces the complexity of the UCM diagram, by only drawing one path for two scenarios which have some common responsibilities. The basic UCM notation discussed can be found in figure 2. More detailed information on Use Case Maps can be found in [5].

2.3 Real-Time Embedded Systems

A real-time system is defined as any system where the time between a stimulus and response is of importance [4]. The target value for the response delay varies from system to system, and depending on the type of system, the target response delay can range from nanoseconds to seconds. There are two categories of real-time systems, which are hard and soft real-time systems. A hard real-time system is defined by the need for a response by a certain deadline. If the response comes after the deadline, a failure will occur. An example of a hard real-time system is a missile guidance system. Trajectory calculations have to be fed into a missile by certain times, otherwise the missile may never reach its intended destination. Soft real-time systems do not have strict requirements on response time, because in these systems it is acceptable to have some variation. A database system is an example of a soft real time system, because if the response for a query takes 2 seconds, or 3 seconds it has no catastrophic repercussions. Response requirements for a soft real-time system can be expressed by mean and variance on the response time, or by a percentage of responses occurring within a certain time.

Embedded computer systems are systems in which a computer plays a role within a larger engineering system to process information. Embedded systems have two distinct features which are, they are designed to perform specific tasks, and the system interacts with other hardware devices which interface to the external world. In these applications, the timeliness of processing data can be crucial to the operation of the entire system, hence making them real-time systems. Embedded computing is also a cost effective way of building a computer system to perform specific operations very quickly, like DSP boards for example.

2.3.1 Obtaining Measurements From Real-Time Embedded Systems

When modelling any system, some parameters will be required to come from measurement data. Obtaining measurements is very similar to testing, in the sense that both require to execute some scenario within the system. However measurement and testing have different goals. The intent of measurement gathering is to determine how long it takes to execute parts of a scenario. Testing is concerned with verifying the proper execution of the scenario against the design. Testing, and therefore measurement gathering for embedded systems is more difficult than a PC or a workstation [7], [20]. The reason for this is, embedded systems have just enough hardware and software to perform their intended task(s). This means many features that makes a PC or a workstation convenient to use like file storage, video display, user memory, and keyboard are not present in an embedded system. User interfaces found in embedded systems range from shorting a jumper, to a command line or menu driven interface, accessed through a serial cable. This is the oppo-

site to what can be found on a PC or a workstation, which has a general command line interface with many commands, and batch scripting to automate work.

Usually the only way to get measurements out of an embedded system is by some custom hardware/software solutions, like CodeTEST [1]. This means that the measurement infrastructure built for one embedded system is only good for that system, and is not very portable. This is because measurements are largely dependant upon specific low level hardware details of the system, which boil down to monitoring signals at various points on a printed circuit board. To perform such low level monitoring of system behavior, this will require electronic equipment like a frequency counter, logic analyzer, or an oscilloscope. Equipment of this nature usually have no provisions to be interfaced with a PC or a workstation, hence making it difficult to automate the measurement process. Achieving some degree of automation in measurement analysis is important because sometimes many measurements have to be obtained. Even little procedures like manually recording measurements can become monotonous and error prone because embedded systems usually do not have the memory to store much information.

Embedded systems have interfaces for input and output, and the only way the system will do useful work is by driving these interfaces. This means more specialized hardware will be required to properly interface with the system. The extra equipment may be custom built, which probably will be another embedded system, or some commercial product. Again this comes back to the issues mentioned earlier about the lack of automation, and general facilities to make the process of measurement gathering less tedious. Interface equipment needed for measurement experiments will probably have no provisions for

automation or a interface to a PC or workstation to govern its operation. This means a human operator will be required to properly set up experiments.

Real-time kernels found in embedded systems may lack high resolution timing facilities (in tens of microseconds) to make measurements on software and other important events happening on a small time scale. Solaris and HP-UX provide the `getrusage()` function that has a resolution within 10 microseconds. Real-time kernels require 1 to 10 millisecond timers to exist in the target system for the kernel's internal use. In order for a real-time kernel to provide an equivalent to `getrusage()`, the kernel would have to keep track of which task is currently executing, and how long the task ran for to get proper readings. All of this extra tracking would take up memory, and kernels cannot consume a lot of memory, because they are designed to work in as many systems as possible.

2.4 Real-Time Embedded System Example: Bridges And Routers

Bridges and routers [8], [22] are devices that make ubiquitous inter networking possible. The primary purpose of these devices is to connect different networks together. From an OSI [22] communication protocol stack perspective, bridges and routers implement the bottom 2 (up to Data Link Layer) and 3 (up to Network Layer) layers respectively. OSI defines these devices as relays, because they support enough of the protocol stack to forward information to other networks.

A bridge connects two networks that use the same networking protocol. Since bridges only implement the bottom two layers of the OSI model, they are less complex than routers. All bridges employ address filtering, which means the user of the bridge can program

which packets addressed to various destinations should not be forwarded to the other side. This is primarily done so a LAN connected with more than one bridge may not get flooded with redundant packets. Some bridges also have the capability of dynamically changing their records to keep track of when nodes are added or removed on different LANs. The bridge will add a new address to its records when a packet destined for a new address is successfully transmitted. This implies that a new node has been added to the network. When a node is removed from the network, the bridge will erase this entry from its records so that it does not forward packets addressed to that node.

Routers are more complex because they encompass all the functionality found in a bridge, and extrapolate this functionality by connecting more than one network together. Each network connected to the router does not necessarily have to support the same networking protocol as the other connected networks. This means the router must be designed to support all the possible networking protocols, and be able to convert packets from one protocol to the other. Since routers generally do more processing on each incoming packet, there is always a chance that the rate of incoming packets may be too great at a given time. In circumstances like these, it is natural for the router to drop packets, and allow the higher level protocol to re-transmit the packet.

Figure 3 shows an example network where a bridge and router would be used. Below each LAN number, is the higher level protocol, and following the slash is the networking protocol used. Notice that the bridge only links two networks with the same networking protocol, yet the router joins many different LANs with different networking protocols.

FIGURE 3. Example situation where a router and bridge would be used

2.4.1 Implementation Issues Of Bridges And Routers

All routers must be capable of manipulating, constructing, fragmenting, and merging packets for all the protocols the router supports [8]. Manipulating and constructing packets is important for sending acknowledgments that a packet(s) has been received, or to convert from one packet format to the next. Packet fragmenting is required for networking protocols that have an upper or fixed limit on packet size. Packet merging is needed for those network protocols that have either a minimum packet size, or a fixed larger size of packet. It can also be used to tune the performance of the protocol by reducing the packet overhead for each frame, which is done by decreasing the payload to packet header ratio.

FIFO queues or mailboxes are used to buffer packets between layers of the protocol stack, travelling either up or down the stack. Despite having provisions for buffering packets between layers, there has to be flow control between layers. This is so that certain lay-

ers do not become saturated with packets, or to accommodate a layer which does not have enough resources to deal with the packet at a given time.

Both bridges and routers have system management tasks to manage the protocol layers, general system management, and system maintenance. An example of general system management would be to report and collect statistics on the various protocols, such as errors, and the number of frames received and transmitted. Setting up various operational parameters for the protocol stacks in the router would be considered as protocol layer management. System maintenance management would be considered as having the capability to perform low level maintenance operations like loading in new code for the bridge or router.

Another type of task commonly found in bridges and routers is some form of a timer task that awakens after being set to sleep for a certain amount of time. This gives the bridge or router the capability to time out and/or recover on certain operations. An example of recovery or timing out would be to update certain finite state machines or take appropriate action within the various layers of the protocol stack.

Scheduling of tasks in the bridge or router is the responsibility of the real-time kernel. The kernel is also responsible for handling inter-task communication, semaphores for critical sections, interrupt handling, and timing facilities for tasks. The kernel however does not necessarily have any memory management, which is usually the programmer's responsibility. Fortunately memory managers are standard in programming libraries. Buffers are usually allocated once, and are either static or obtained from a buffer pool. This is to save on allocation and de-allocation of memory which can become costly. When pack-

ets are being passed along from layer to layer, a pointer to the packet will almost always be passed around to eliminate the cost of copying packets from one region of memory to another.

2.5 Communication Protocols

A communication protocol is a definition of how two or more computer nodes exchange information. Below is a description of several communication protocols which appear in this thesis.

Ethernet protocol

The Ethernet IEEE 802.3 protocol [8] is a multipoint protocol that embraces the data layer of the OSI model. The concept of Ethernet is that every node is connected to a common link, and every node can transmit on that link. The protocol is capable of knowing when two nodes managed to transmit at the same time, and when another node is transmitting. If the link is busy, the node that wants to transmit backs off for a random period of time. Since all nodes are connected to the link, each node must always monitor the link. The Ethernet standard does support error checking for its packets, but does not have any provisions for positive or negative acknowledgments. If a packet is received and fails the error checking, that packet is discarded. It is then responsibility of the higher level protocol to re-transmit the packet.

HDLC protocol

The HDLC protocol [22] is both a point to point and point to multipoint protocol, which exists in the data layer of the OSI model. The HDLC protocol guarantees delivery

of frames, unlike the Ethernet standard. To achieve this, HDLC has error checking along with positive and negative acknowledgments. Therefore if a frame fails error checking, the recipient can send a NACK to the transmitter for a re-transmission. HDLC also provides flow control capability, which allows the transmitter to send frames only when the recipient is ready to receive them. This mechanism ensures that the receiver will not discard frames, if for some reason the receiver cannot process frames as fast as they come in.

Point to Point Protocol (PPP)

PPP is a standard for encapsulating network layer information (usually IP) for transmission on a point to point link [13]. This means that PPP is a protocol stack which defines the data and partly the network layers of the OSI model. The protocol at the data link layer used by PPP is HDLC. PPP exists on the same level as Ethernet IEEE 802.3. Therefore when TCP/IP packets have to be sent on a point to point link, PPP will be used, but if the packets have to be sent on a LAN, Ethernet IEEE 802.3 is generally used.

TCP protocol

TCP stands for Transmission Control Protocol [8], which occupies the transport layer of the OSI stack. The transport layer is one layer above the network layer. This is a higher level protocol which is transparent to bridges and routers. IP stands for Internet Protocol [8], which occupies part of the network layer in the OSI model. The main functions this protocol delivers is routing, and merging/fragmenting of packets, which have been discussed in sections 2.4, and 2.4.1.

Chapter 3

Outline of a methodology for modelling embedded systems

3.1 A Methodology For Modelling Embedded Real-Time Systems

This thesis presents a generic approach to applying layered modelling to implemented embedded real time systems. The performance methodology achieves this by systematically analyzing the relevant areas of the hardware and software architecture together. The two things gained from this analysis are, a high level understanding of the system, and a layered model to predict performance of the system.

The methodology is shaped to overcome certain difficulties when attempting to do performance analysis on embedded systems. The design of an embedded real time system can become hard to follow because of complex interactions between the software and hardware. These interactions can become more obscured over the life time of a product, because the software and hardware architectures are optimized.

1. Define the scope of analysis, by choosing relevant scenarios and workloads.

Before any analysis begins, it is important to understand the performance questions that must be answered. One example of a typical performance question is, how many users can this system handle simultaneously given some metric for quality of service. Another question would be how many transactions of a certain type per second can the system handle under certain loads.

Based on these questions, the analyst should discuss with designers of the system what relevant group of scenarios execute in the system. This is a very important point to follow, because this is really what defines the magnitude of the analysis. If a situation occurred where there are too many unimportant or closely similar scenarios being examined, the analysis may either fail or take very long to produce answers.

It is also important to determine the operating conditions of the system when answering the performance questions. Examples of operating conditions are, arrival rates of jobs going into a system, or background processing while a scenario of concern is being executed. Knowing this information may not be necessary until as far down as step 7, when the actual analysis begins.

2. Determine the sequences of responsibilities to accomplish the scenarios.

Now that enough of the scope of the analysis has been set, it is time to start building a model of the system. Choose a scenario from the group of scenarios in the system that contribute to the achievement of an overall goal. With the chosen scenario, determine the sequence of all the responsibilities to accomplish it. This will involve searching through existing documents, talking with designers, or maybe even looking through source code to reverse engineer the system. Then determine which devices execute the various responsibilities, and understand what resources each device requires in order to fulfill its responsibility. For embedded systems, this is determined from looking at the hardware architecture. The results of this work is documented in the form of a Use Case Map. This

serves two purposes, one as a reference for the performance analyst, and second supplementary design documentation for the system.

When tracing scenarios through the system, keep note of all the synchronization and queueing points in the software or hardware. These points are typically important to include in a performance model of a system. It is also important to include the number of times certain responsibilities are executed within the execution of a scenario. This information should be a supplement to the UCM, because there are no provisions in the notation to handle such detail. The only exception to this are synchronization points, because they can be expressed using AND joins. This step is demonstrated in section 5.1.

3. Build model from scenarios within scope of the analysis.

With all the scenarios relevant to the scope of the analysis documented in UCM format, along with any other relevant information from step 2, a first pass layered model can be constructed. The model is derived by iteratively adding one scenario at a time, until all relevant scenarios are represented in the layered model. The model derived in this step is not a complete model, but it does contain the final structure, along with all the proper flow of causality. This step is demonstrated in section 5.2.

4. Put in any behavioral details/approximations and simplifications into the model.

Usually a second pass is required to complete the model after the structure is derived from the previous step. Layered modelling is not a robust modelling technique yet. Therefore it is possible to find systems that have properties which cannot be expressed directly with the layered model paradigm. It is important to include these behavioral details and

approximations as best as possible in the layered model. This is because even if behavioral detail is poorly represented in the model, it is better to leave it in and make note of it, than not to include it. This follows a philosophy of something is better than nothing, that even a bad approximation can give some insight into the operation of a system.

First pass models generally stand a good chance of having too many unnecessary tasks and entries. Simplifications to the model should always be made, in the interests of faster simulation or solution times. One should strive to make simplifications without sacrificing any accuracy, but sometimes this is not possible. These simplifications should be documented, otherwise it can be difficult to understand how the model represents the system. A situation like this is not a good one because no one will believe the results from the model.

At the end of this step, every detail of the model structure has been taken care of. The only detail missing from the model are the values for the parameters. Because the model's structure and parameters are completely defined, this focuses the effort in obtaining relevant measurements from the system. This step is demonstrated in section 5.3.

5. Obtain measurements for execution of various responsibilities, and put parameters into model.

Carrying on from step 4, obtain the necessary values for the required parameters by making measurements on the system. Most of the time, some values might be already known before hand, and therefore reducing the measurement work. Procedures on how to obtain measurements are very system dependant, especially when dealing with embedded systems. In the end, one must somehow determine the correct value for usually a service time, delay, or visit ratio.

By this stage, it may be necessary to know the operating conditions under which certain performance questions have to be answered. This is a judgement call the performance analyst has to make, and could become another parameter in the analysis. Sometimes there may be a reason to put stress on a system when making measurements. For example effects like caching may emerge and give software measurements that are much different from when the system is not stressed.

6. Validate model against real measurements if possible.

Models are abstract representations of a system, and before using the model to make performance predictions, it is desirable to have the confidence that the model is correct. This is achieved by validating the model against actual system measurements. Comparing delay measurements from the real system with delay predictions from the model is the most important step in validation. Other results like throughputs and utilizations should also be compared as further proof that the model is a good representation of the system. Using predicted delays to compare with real measurements is the best validation, because delay predictions are the most difficult to obtain. This is because delays are usually a function of more than two parameters. Hence if any one of these parameters is wrong, this can be reflected in a incorrect delay prediction. Therefore comparison of delay results gives better coverage across the model. Unfortunately validating a model is a lot of work, and may not even be practical in many situations. Therefore one must settle with whatever is available or possible at the time. Though some kind of a sanity check should be made on the model to see if it represents some aspects of the system.

It may be wise to think about model validation at the beginning of step 5 (measurement gathering), while all the equipment is in one place, and ready to run experiments. This is assuming that the experiment setup is sufficient, or very close to what is required to make validation experiment runs. There are two reasons to concurrently think about validation experiments at the same time as measurement experiments. One reason is resources within a company are usually scarce, and the measurement experiments are probably using resources that can be used for development work which is higher priority. Since all the resources are together, it is best to make use of them, because all the resources may not be available for a significant amount of time. The second reason is loss of productive momentum, which naturally comes from loss of familiarity about the details of the experiment set up over time. If possible, it is always better to complete a task in one shot, instead of fragmenting the effort over time. This is because every time the task is restarted, there will always be a “start up and familiarize” phase before any progress is made.

7. Use model to answer performance questions.

The final step is to use the model to answer the performance questions defined in step 1. A typical example of a performance question is, how many users can this system handle simultaneously given some metric for quality of service? Another example performance question would be how many transactions of a certain type per second can the system handle under certain loads. The answer to these questions are obtained from the layered model, by observing various delay, throughput and utilization results.

If time permits, it can definitely be a worthwhile exercise for the performance analyst to briefly explore outside the scope of the study. For example prove some suspicions about

bad system design, which can highlight some potential problem in the system. Following hunches (within reason) may discover problems in the system which would not normally be found within the scope of the analysis.

3.2 Comparison Of Presented Methodology With Connie Smith's

The performance methodology that Connie Smith proposed in [21] is outlined below.

1. Define scenarios to model
2. Create model topology
3. Determine queueing disciplines and relative processing speeds
4. Derive classes for model from execution graphs
5. Determine visit ratios and service times
6. Determine missing model parameters from measurements
7. Solve model
8. Check for to see if results are reasonable
9. Evaluate model results and identify bottlenecks
10. Evaluate partial utilizations of software components on each device
11. Evaluate queuing delays for each software component
12. Identify and evaluate alternatives
13. Verify data and validate model results

The performance methodology presented in this thesis is essentially the same as the methodology shown above. One of the main differences between the two methodologies is the performance methodology presented in this thesis is geared toward reverse engineer-

ing, and measuring an implemented system. The methodology presented in [21] can be applied to both an implemented or not implemented system. The other difference is [21] does validation of the model after the performance analysis has been completed. This is the opposite to the methodology presented in this thesis which validates the model before any analysis begins.

The methodology presented above is more complicated than the methodology presented here, because there are more steps to follow. One of the reasons for this is [21] uses the concept of an execution graph. An execution graph is a flow chart showing software execution with attached information like number of disk I/Os, number of CPU instructions a software module has to execute, etc. Part of the methodology is to convert this flow chart into a Information Processing Graph (IPG) developed by Scientific and Engineering Software (SES) [19]. This IPG is an intermediate notation used to describe a queueing network model. The transformation itself converts parameters in the execution graph model to visit ratios and service times, which are needed in the IPG and eventually the queueing network model. On the other hand the methodology presented in the thesis uses Use Case Maps to document the “reverse engineering” of the system. This focuses the performance analyst to understand the system without thinking about performance, and the results of this work can be passed on to the project team involved with the system as design documentation. From the Use Case Maps, the responsibilities make it apparent what measurements should be made, and these measurements will map on to the layered model directly. Because of this, in the presented methodology there is no need for any calculations (step 4 in [21]) when going from the Use Case Map model to the layered model hence making it simpler.

With layered modelling, the trace of causality does not disappear in the model, hence making the design more understandable and easier to communicate to the others.

Another advantage of the presented methodology is that it uses Layered Queueing Modelling, and not Queueing Network Models. This makes the methodology in [21] more complicated because there is always post processing of results (steps 10 and 11 in [21]) from the queuing network solver to map back into software. Layered modelling does this already, and there is no need to do these calculations. This makes layered modelling better suited for modelling software than QNMs.

One of the biggest problems with QNMs is the fact they have no provision for simultaneous resource possession. This means that some resources will require more than one other resource to service a request. For example when a CPU executes code, it requires two other resources simultaneously to perform its duty, which are the address/data bus, and memory. In embedded systems, simultaneous resource possession occurs frequently at the hardware level when modelling buses. They can also occur at the software level when modelling mutual exclusion. When using QNM, and abstracting away from simultaneous resource possession, it is possible the QNM model can give incorrect conclusions to where the bottleneck resource(s) are in the system. Layered modelling has the provisions to express and solve models that have simultaneous resource possession, and hence making it a better choice for a solution technique than QNM which [21] uses.

Chapter 4

Description of an embedded system

4.1 High Level Description Of System To Be Studied

The case study will analyze Gandalf's Expressway RLAN system, where its general architecture is shown in figure 4. Users have an ISDN line in their home or office, connected to a PC or workstation. When users connect to the RLAN, the "Remote LAN" makes it appear that they are directly connected to the LAN. The most simple ISDN user link available is a DS0, which operates at 64Kbits/sec.

FIGURE 4. High level diagram of the RLAN's major input and output ports

On the ISDN side of the RLAN, there is a variable number of DS0 links available to handle incoming calls from remote users. A user can occupy one (singlelink connection) or two DS0 links per connection (multilink connection) to take full advantage of the ISDN line bandwidth. The number of DS0 links the RLAN can support is up to 124. On the

other side of the RLAN there is a 10 Mbits/sec LAN network which uses a standard Ethernet protocol. When a user wants to transmit a frame on the DS0 channel, the frame will be bridged over to the LAN side or routed to another DS0 connection. If a packet arrives on the LAN, it will be routed to the appropriate DS0 channel, provided there is an available connection with the same address as the packet destination. Each DS0 channel uses a synchronous PPP protocol with or without compression at the network layer, and a HDLC protocol at the data link layer.

4.2 Description Of The Hardware Architecture

Figure 5 shows a high level hardware interconnection diagram of the Expressway RLAN.

FIGURE 5. Hardware interconnection in the RLAN system

This is a system with 1 shared bus, and 6 bus masters being the Main CPU, LAN Chip and Munich Chip all contending for main memory. The operation of various components in figure 5 is discussed in more detail below.

Compression Card

Each DS0 channel can support a compressed PPP connection. Since the cost of compressing and de-compressing frames is high, a daughter card was designed for just that purpose. The daughter card has two CPUs, each having their own memory and bus, running the identical compression software. The software running on the main CPU communicates with the compression card exclusively through dual port memory. This thesis does not consider the compression feature, because it is proprietary.

Munich Chip

The Munich chip provides the RLAN with an ISDN interface, and the capability of receiving and transmitting HDLC frames. The name of the chip is derived from the city it was designed in, Munich Germany. There are four Munich chips in the RLAN system. Each chip has the capability of receiving and transmitting HDLC frames simultaneously on 32 DS0 channels. What is important to know about this chip is its basic behavior on the bus. The Munich chip stores incoming frames into buffers which reside in main memory. The chip also polls buffers in main memory to see whether it has any frames to transmit. These buffers have a special format known to the Munich chip, which are called a descriptors.

When there is nothing to transmit, the Munich chip will check its transmit descriptors every 125 μ s until a frame has to be transmitted on the DS0 link. When the Munich chip has the bus, even when checking the descriptors, it will try to fill up its on board transmit FIFO buffers, or empty its on board receive FIFO buffers. When the Munich chip has nothing to transmit, it will request the bus to check the descriptors every 3.9 μ s, for every active channel. In this mode the Munich chip must do roughly 25.2 μ s of work within 125 μ s period when 32 channels are active. The Munich chip has the capability of doing sixteen 32 bit read/write operations per bus request. The FIFO buffer on each Munich chip is configured, so each channel will have 8 bytes of buffer for each of the receiving and transmitting directions.

LAN Chip

There is only one LAN chip in the RLAN system, because there is only one LAN (which runs at 10 Mbits/sec). The LAN and Munich chips share several features which are, 1) they both have their own descriptor formats in main memory, and 2) both are designed to automatically put receiving data into their descriptors.

Despite these commonalities, the LAN chip does operate differently from the Munich chip. The one major difference without going into any detail is that the LAN chip cannot receive and transmit a packet simultaneously. Another difference is that the LAN chip is not designed to check its descriptors to see if it has anything to transmit. Instead the LAN chip is told by the main CPU when there is something to transmit via a command queue. When the LAN chip has been told to transmit, it will automatically fetch descriptors and

transmit the packet onto the LAN by itself. The LAN chip has an on board FIFO buffer of 32 bytes for each of the receive and transmit directions.

Bus Arbiter

The bus arbiter grants permission to various devices on the main bus which can use it, and ensures that permission is not given to more than one device a time. The arbitration scheme implemented in the RLAN is a priority based “hold till done”. Except for the CPU, any time a device has the bus, it can hold the bus until the device is finished, even if a request comes in from a higher priority device. The only exception to this rule is if the CPU has a lock on memory. For the duration of the lock, no device can pre-empt the CPU to give up access to memory. This is not accounted for in any of the analysis, because the duration of the locks are so small that they do not affect other devices on the bus. Figure 6 shows a high level state machine of how the priority in the bus arbiter works.

FIGURE 6. Priority built into arbiter

Munich Arbiter

The Munich arbiter manages requests from the 4 Munich chips. The main purpose of this arbiter is to ensure the Munich chips do not hold the bus indefinitely, starving the bus

from other devices. This is achieved by ensuring no Munich chip can request the bus more than once during a session. A session is the time from the first request from any of the four Munich chips, to when no Munich chip is requesting the bus that has not held it during the current session. Since at the end of a session no Munich chip is requesting the bus, the Munich chip arbiter will not issue a request for the main bus. This will let the other lower priority devices access the bus. Figure 7 gives a high level state machine on how the Munich Arbiter works.

FIGURE 7. High level state machine diagram of the Munich chip arbiter

Munich Chip Throttle

The Munich chip throttle is a mechanism that disables the Munich chip from requesting the bus for a certain amount of time. If a memory request occurs when the Munich chip cannot access the bus, the request is queued internally. A maximum of 16 requests can be queued before the Munich chip resets. As mentioned earlier, when the Munich chip has nothing to transmit, it has 25.2 μ s of work to complete within 125 μ s in each polling cycle. Therefore the Munich chip throttle cannot disable the Munich chip requests for more than 99.8 μ s, otherwise the Munich chip will crash the system.

Main CPU

The main CPU is the heart of the RLAN system. It executes the software to process a frame found in the Munich descriptors and transfers it into the LAN descriptors. Conversely the software also processes frames found in the LAN descriptors and transfers them into the Munich descriptors. The main CPU runs a real-time kernel that manages 11 tasks. These tasks fulfill other necessary functionality of the RLAN like management and maintenance of the system. It also communicates with all the devices in the system which are the Munich chips, LAN chip, main memory and dual ported memory for the compression card.

Main Memory

Main memory is a single centralized memory unit in the RLAN, where code executes by the main CPU resides, along with any data the main CPU manipulates. Also all the DMA devices (Munich chip, LAN chip) access main memory to store and retrieve incoming and outgoing frames.

4.3 Description Of The Software Architecture

Figure 8 shows a high level software/hardware diagram of the Expressway RLAN system, showing the major software modules and hardware components.

FIGURE 8. High level software/hardware diagram of the Expressway RLAN

This system runs on top of a real-time kernel, and one component not shown in figure 8 is the Port Worker task. The Port Worker is the main task in the system that performs 95% or more of all the processing done by the RLAN system. It polls all of the queues and descriptors shown in figure 8 through the software interface functions of each software module. If a packet or a frame is ready to be processed in one of the descriptors or queues, it will be processed, and sent to another queue or descriptor. The polling order of the Port Worker is shown in figure 9, where the function calls correspond to the software interface functions in figure 8. Each call may either accomplish useful work, or do nothing if there is nothing to service.

FIGURE 9. High level pseudo code describing the Port Worker polling order

```
while (TRUE)
{
    for(x=1; x <= number of active channels; x++)
    {
        hdlc_service_rx(x);
        hdlc_service_tx(x);

        if (x mod 12 == 0)
        {
            sc_cc_copy();
            cc_sc_copy();
        }

        rx_framer(x);
        tx_framer(x);
    }

    lan_service_cu();
    lan_service_ru();
    pt_service_int_rx();

    sc_cc_copy();
    cc_sc_copy();

    sleep until awakened by Null task or timer tick
}
```


RX_framer and TX_framer

Recall the Munich chip has its own descriptor format as mentioned in section 4.2. For reception, when the Munich chip has received a complete frame, the chip will mark all the descriptors associated with that frame as “finished receiving frame”. It is the job of RX_Framer to take these descriptors, assemble them into a frame, and pass them into the HDLC receive queue for further processing. For the TX_framer function, HDLC_service_TX passes frame(s) down to the TX_framer queue. TX_framer will take the frame and convert it into the Munich chip descriptors. The frame will be transmitted on the DS0 channel by the Munich chip via DMA.

HDLC_service_RX and HDLC_service_TX

These two functions implement most of the synchronous PPP protocol stack. This is where acknowledgments of packets occur, flow control, etc. The RX and TX at the end of HDLC_service designates packets entering and exiting on the WAN side respectively. For the reception case, the HDLC_service_RX function will call the bridge/routing function, which will then place it into the LAN queue for it to be sent off on the LAN. For the transmission case, HDLC_service_TX will take a frame from the WAN queue, and send it down to TX_framer, which will eventually make it out the WAN side of the RLAN.

LAN_service_RU and LAN_service_CU

These two functions support the LAN chip described in section 4.2. The LAN chip operates in such a way that it has two command queues, which are for receive

(LAN_service_RU) and transmit (LAN_service_CU). When a packet has to be sent out on the LAN, a command is issued on the transmit command queue. When a packet is received on the LAN, a notification of this event is placed on the receive queue. In the case LAN_service_RU has received a packet, it executes the bridge/routing, and then places this packet into the appropriate WAN queue. When LAN_service_CU is called, it checks the LAN queue to see if there is a packet to be transmitted. If so, LAN_service_CU will take a packet from the LAN queue and convert it into the LAN chip's descriptor format. Eventually the LAN chip will transmit the frame onto the LAN using DMA.

CC_SC_copy and SC_CC_copy

These two functions copy frames from dual port memory to main memory, and vice versa respectively. These functions are only active when compression is enabled.

Other Tasks in the system

Eleven other tasks beside the Port Worker task exist in the RLAN system. All of these tasks account for very little of the utilization of the main CPU, because they are outside the scope of the analysis, which deals with steady state data transfer. For an example of non steady state behavior, one of the eleven tasks manages the opening and closing of a ISDN connection.

Task Scheduling

The last item to be covered in the description of the RLAN is how the kernel schedules these 11 tasks. The Port Worker has the highest priority, and there is a Null task which has the lowest priority. The other ten tasks have priorities in between the two. The Port Worker will execute one complete polling cycle, and block on a semaphore. This gives the lower priority tasks time to execute if they have to. Eventually when the Null task starts executing, it signals the semaphore the Port Worker is blocked on, and the Port Worker will wake up and the whole cycle begins again. There is the possibility that some or all eleven tasks combined will require significantly more CPU time than usual, meaning that the Port Worker task may not execute for a long time. This problem is solved by the semaphore being signaled every 10 ms. This will insure that the Port Worker will not be starved for more than 10 ms, pre-empting any lower priority task currently running.

FIGURE 10. Task scheduling employed in the RLAN

As mentioned earlier, the Port Worker task is known to take 95% of the CPU time. In the analysis, the task scheduling described will be approximated by assuming the Port Worker will run on the CPU without any contention from the other 11 tasks. This means

the execution of the 11 other tasks is being ignored, because they don't pose a significant impact on scope of the analysis, which deals with steady state data transfer.

Chapter 5

Building a Base Model of the RLAN

A performance model of the RLAN system can be built with the system description found in chapter 4. This chapter will apply the first 4 steps in the methodology presented in chapter 3. The first step in the methodology is to define the scope for the analysis before any progress towards building the model begins. The scope of the analysis is to consider the PPP singlelink uncompressed connections with the RLAN in bridging mode, with variable packet/frame sizes, and no background LAN traffic. The analysis is limited to steady state data transfer, with no connections being added or taken away. The next 3 steps to follow are to trace the scenarios through the system, and from these execution paths build a layered model.

Constructing a layered queueing model of the RLAN system was not a trivial task. The reason for this was the layered model lacks a polling discipline, and that hardware devices make requests to resources in the same manner as software. This means one must creatively capture the workload and properties of the system, to get the model.

5.1 Determining Scenarios In The RLAN System

With the scope of the analysis defined, the methodology proceeds to step 2, which traces and documents the scenarios to be modeled. Within the scope of the analysis, there are two primary scenarios for traffic travelling in both directions. The first scenario starts at an arrival of an HDLC frame at a Munich chip and goes to the LAN. The second sce-

nario occurs when a packet arrives at the LAN chip and goes to a DS0 channel. The scenarios are described by the Use Case Maps in figure 11 and figure 12. The UCM traces a path through devices in the system that perform computation on a packet/frame.

There may be other sources of stimuli that cause work in the RLAN, like data arriving on the serial port. Frames and packets can trace out different scenarios other than the ones within the scope of the analysis, for example when a user telnets into the RLAN console to perform maintenance work. Another different scenario can occur when a frame is received on the DS0 link, and is transmitted on to a different DS0 link, never being forwarded to the LAN. The scenarios just described are out of the scope of the analysis going to be presented, but can be the scope of another performance study.

The UCM for a packet going from LAN to WAN is shown in figure 11, and will be the first scenario discussed. The LAN to WAN scenario begins when a LAN packet is received from the LAN. As the LAN chip receives the packet, the LAN chip will buffer the incoming bytes into its onboard FIFO buffer. When the FIFO buffer starts getting full, it will store the incoming bytes into one of the LAN receive descriptors which is located in main memory. Once the packet has been fully received, the LAN chip marks the packet as complete and switches to a new empty descriptor. The Port Worker will eventually check the LAN receive descriptors, and process any packets that are complete. The first part of the processing is to determine what DS0 link the packet is destined for, by calling the bridge/routing functions. Once the destination is determined, the packet is queued to its appropriate DS0 link queue (WAN queue). The packet will stay in the WAN queue until the Port Worker task will check it, and take it out of the queue given the appropriate conditions.

The HDLC portion of the PPP stack processes the packet to a frame, and sends it to TX_framer to convert the frame to the Munich chip descriptor format. The Munich chip polls these descriptors to see if there is anything to transmit. Once the Munich chip finds a new frame ready to transmit, it will DMA this information out onto the appropriate DS0.

The acknowledgment scenario is also shown in figure 11. After some time, partly dependent on the size of the HDLC flow control window, an acknowledgment will be received for the frame(s) received by the remote user. When the frame is being transmitted on the DS0 link, the Munich chip buffers the incoming bytes into its own FIFO buffer. When the onboard memory starts to become full, the Munich chip dumps the contents of its memory into one of the WAN receive descriptors which is located in main memory. The Port Worker task will eventually check these descriptors for that DS0 link through the RX_framer function, and assemble the frame. The frame is then sent to the HDLC_service_rx function, which processes the ACK from the remote user, updates any state variables, and discards the frame.

FIGURE 11. Use Case Map of a packet going from LAN to WAN, singlelink

The WAN to LAN scenario starts when a frame on the DS0 is transmitted by the remote user. When the frame is being received on the DS0 link, the Munich chip buffers the incoming bytes into its own FIFO buffer. When the onboard memory becomes full, the Munich chip dumps the contents of its memory into one of the WAN receive descriptors which is located in main memory. The Port Worker task will eventually check these descriptors for that DS0 link through the `RX_framer` function, and assemble the frame. The frame is then sent to the `HDLC_service_rx` function, which processes the packet, and updates any state variables. Depending upon certain conditions like the size of the sliding window on the HDLC protocol, an acknowledgment will be sent to the remote user for the successful reception of frame(s). Then the frame is sent to the upper layer of the PPP stack for bridging/routing, and is deposited into the LAN queue. The Port Worker will eventually check the LAN queue to see if there are any frames to be processed, and will process them if the proper conditions are met. When the frame is processed, it gets converted into the LAN chip transmit descriptors, and the LAN chip is instructed by the Port Worker to send the packet off to the LAN.

When an acknowledgment is sent to the remote user, the HDLC portion of the PPP stack creates an ACK frame, and sends it to `TX_framer` to convert the frame to the Munich chip descriptor format. The Munich chip polls these descriptors to see if there is anything to transmit, and will DMA the frame onto the appropriate DS0 link when the frame appears in the descriptors.

FIGURE 12. Use Case Map of a frame going from WAN to LAN, singlelink

5.2 Mapping Scenarios In The System To A Layered Model

The scenarios of interest have been defined in section 5.1, and this leads to building the layered model, which is step 3 of the performance methodology. The transformation of the UCM description into a layered model starts with identifying all the resources that have to do work, and binding all the responsibilities to these resources. Choosing to start with the LAN to WAN scenario from figure 11, three resources are identified that play a role in the completion of the scenario. These resources are the LAN chip, CPU, and Munich chip. It should be noted that the CPU task represents both the software and the hardware. Usually the processor would be the actual CPU, and the task would be called Port Worker, but the two have been collapsed. The responsibilities found in figure 11 are bound to the appropriate resource tasks as an entry.

To execute all the responsibilities in the scenario, an “operation task” one layer above is introduced. The operation task is a pseudo task that does nothing except call all the entries that belong in a scenario. To fix the rate at which these operations occur, one layer above the operation task is a driver task. The purpose of this task is to determine the rate at which particular scenarios are executed. The model developed so far is shown in figure 13.

FIGURE 13. Grouping of responsibilities to resources

The next step is to identify any contention these resources impose on each other, which falls out of the hardware architecture described in figure 5. From figure 5, all three of the devices are connected on the main bus, and all three of these devices use the bus in order to access main memory to fulfill their responsibilities. With this new information, the model is advanced to what is shown in figure 14. The new model shows the three devices all accessing the main bus, which then automatically acquires access to main memory.

FIGURE 14. Addition of another layer to represent the main bus.

This now completes the first pass derivation of the LAN to WAN scenario into the layered modelling paradigm. The next step is to build on this model by adding the ACK reception scenario, that is sent from a remote user on a DS0 link. Figure 11 shows the resources and responsibilities to complete the ACK reception scenario. Using the same reasoning for the LAN to WAN scenario, the first part of the model is derived and shown in figure 15.

FIGURE 15. Grouping responsibilities for the reception of an acknowledgment

Comparing models in figure 14 and figure 15, both use the CPU and the WAN chip resources. Using the same reasoning for the derivation of the LAN to WAN scenario, the two models are merged. This merging is done by grouping all the responsibilities from the two models into one device task, for each device. The result model is shown in figure 16.

FIGURE 16. Merging the two models derived from the two scenarios

The WAN to LAN and ACK transmission scenarios are the next scenarios to be captured into the layered model. The UCM of these scenarios can be found in figure 12. In the same manner that the LAN to WAN and ACK reception scenarios were derived and merged to construct the model found in figure 16, the same can be done with the two WAN to LAN scenarios. In the model, these two scenarios are called WAN_op for a data frame arriving at the WAN side, and TX_WAN_ACK for transmitting an acknowledgment that the RLAN successfully received the frame(s). The new model with all four scenarios in it is in figure 17.

FIGURE 17. Addition of two more scenarios to model

5.3 Adding Behavioral Detail To Model

So far the model presented has captured the workload explicitly described in the Use Case Maps found in figures 11 and 12. This is where step 4 of the performance methodology begins by adding behavioral details into the model missed in step 3. From the system description in chapter 4, there are several properties in the system that must be accounted for in order to achieve an accurate model.

5.3.1 Main Bus Arbitration

The arbitration mechanism for the main bus as described in section 4.2 is a priority based “hold until finished”. In figure 6 it is shown that the CPU, LAN chip, and Munich chip have lowest to highest priority respectively. Unfortunately layered modelling does not have a discipline that matches the bus arbitration scheme in the RLAN. Layered modelling does provide for pre-emptive priority scheduling for tasks that are assigned on the same processor. The bus arbitration scheme found in the RLAN can be approximated by pre-emptive priority, by assigning lowest to priority to the CPU and a higher and identical priority to the LAN and Munich chips. The reason for this is, when the LAN chip acquires the bus, the Munich cannot pre-empt the LAN chip, but both chips can pre-empt the CPU. The approximation here is that when the Munich and LAN chips request the bus at the same time, the model will randomly give it to either device. This is different from the RLAN where the arbitration scheme would always give the bus to the Munich chip.

FIGURE 18. Model with a more representative bus arbitration scheme

In figure 18, three tasks were added to the bus layer, but the number of processors remains unchanged at one. The memory ops task was eliminated along with the slave layer, because the time spent on the bus is time spent in memory. Now the memory operations are represented by the operations of a “memory” as depicted in figure 18. The priority pre-emption in layered modelling is a discipline in how the processor schedules the tasks, which is why the number of tasks increased to three. Also note each task at the bottom left hand side have a number allocated to them. This number is the priority of the task, with the smallest number being lowest priority, and larger numbers meaning higher priority. Each entry in the resource layer will call a corresponding entry in the bus layer.

5.3.2 Adding Detail To The Munich Chip

There are three features of the Munich chip that are important to include in the layered model. One feature is the Munich chip can simultaneously transmit and receive on all of its 32 channels. The second feature is, the Munich chip can transmit and receive data on the same channel at the same time. Finally the third feature is the Munich chip checks its own descriptors to see if there is anything to transmit.

The first two features are solved by increasing the multiplicity of the WAN_chip task and its underlying processors. So for example if there were 30 channels active, the WAN_chip task would receive a multiplicity of 60. The reason for the multiplicity is that any data arriving at the Munich chip for reception or transmission does not experience any queueing delay. The third feature of the Munich chip is accounted for by adding a separate task called Munich Poll, that has its own separate underlying “processor”. This is separated from the WAN_chip task because it has no direct effect on the processing of data. It is only used to represent the effect of the Munich chip constantly requesting the bus to check if it has anything to transmit. The number of Munich poll tasks is equal to the number of active Munich chips in the system. The polling done by the Munich chip had a considerable effect on the bus and on overall performance when not transmitting, so including it was necessary. Figure 19 shows the model with the additions of detail to the Munich chip.

FIGURE 19. With added detail of the Munich chip

5.3.3 Polling Delay And Polling Overhead In Port Worker

Layered modelling does not support a polling discipline at the time of this writing, which means some mechanism in the model has to account for the effect of a polling delay. This is handled by adding a delay server called Poll Delay. Any particular software operation that will experience a polling delay, will first call the Poll Delay server, which then will call the CPU.

In a polling system, there is also polling overhead, which is the CPU cost of checking whether or not there is something to service. This cost is separated from the cost of servicing a request at a certain queue. For every polling system, there is an average proportion of

CPU time wasted on polling overhead. To represent this in the model, a task called Poll Overhead is added to drive the entries which represent the costs of various polling overheads. Each copy of the Poll Overhead task represents the poll overhead incurred by one active channel. The CPU task will also have new entries that have the polling overhead cost for each entry. The Poll Overhead task calls these overhead entries, to add the effect of wasted polls. The model with the accounted for software polling can be found in figure 20.

FIGURE 20. Model with software polling.

5.3.4 Concurrency In Scenarios And Scaling Of The Model

There are two more issues that need to be discussed to complete the model. The first is identifying concurrency in scenarios. So far all the calls to various entries in the model are synchronous, meaning that a request from a task must wait until its request is complete before making another one. From the description of the RLAN in chapter 4, the reception and transmission of packets and frames by the LAN chip or the Munich chip is done through DMA access. Therefore the reception of packet and frames is done concurrently with respect to the processor. This has to be addressed in the model, by changing the property of the calls going from the operation tasks to the LAN or Munich chip tasks, from synchronous to asynchronous.

The final issue to be discussed is how to drive the model to obtain the desired performance measures. Since each scenario represents the handling of a packet or frame, the frequency at which these scenarios are triggered at will be in terms of packets or frames per second. The driver layer of the model (not part of the software) will have a task corresponding to every source of events; this means one task for each task in the operational layer.

To represent a certain number of active channels in the model, the multiplicity of the WAN driver tasks is set to that same number. Each driver task will have a think time which is one divided by a given arrival rate of frames or packets per second. Since the sources of work do not wait to send the next packet or frame, each driver task will make an asynchronous call to its corresponding operational task. Note the LAN driver will always have a multiplicity of 1, because there is only 1 LAN. The operation tasks are set to infinite serv-

ers, because there can be any number of packets or frames in the system at once. Each frame or packet is “handled” by an operation task which sequences it through the system. Figure 21 shows the model with the discussed changes.

FIGURE 21. Final structure of the RLAN model

5.3.5 Accounting For Munich Throttling

The Munich chip throttle as described in section 4.2 is a mechanism to decrease the bus occupancy of the Munich chip. The throttle achieves this by not allowing the Munich chip to access the bus in 30 μ s intervals, every 30 μ s (a duty cycle of 50%). Every time the

Munich chip makes a request for the bus, the time spent on the bus is main bus arbitration time plus memory access time. When the Munich chip cannot access the bus, requests to memory are queued internally. The Munich chip can queue up to a maximum of 16 requests.

An assumption in the model is made that Munich throttling has no significant effect when the Munich chip is receiving and transmitting frames. When frames are being received or transmitted on the WAN side, it is stated in section 6.6 that the Munich chip will request the bus every 4 bytes it receives, and 8 bytes it transmits. Therefore the Munich chip will make a request every 0.5 ms receiving and 1 ms transmitting, because the throughput of a DS0 is 8Kbytes/sec. With a relatively large gap between requests compared to 30 μ s, the Munich throttle would have negligible effect. Therefore when the Munich chip makes a request for frame reception or transmission, the time spent on the bus will be main bus arbitration time plus memory access time.

Munich throttling however can have a significant impact when the Munich chips are performing a lot of polling. For example lets assume that 32 channels are active in the Munich chip and there are no frames to transmit on any of these channels. In 30 μ s the Munich chip can queue 8 polling requests, which can be serviced with one bus access. This will reduce arbitration time per request by $1/8^{\text{th}}$. Since there are other Munich chips contending for the bus, there is a chance that more than 8 requests can queue up. It was decided to take the best case, that for all polling operations the arbitration time was reduced to $1/16^{\text{th}}$ per request. This assumes every time a bus access occurs, there are 16

polling requests queued. Below is a summary of the new service times for the WAN_bus task to account for the Munich throttle.

$$\begin{aligned} \text{Munich_poll_op} &= \text{Memory_access_poll} + \text{Arbitration_time}/16 \\ \text{Munich_mem_ops} &= \text{Memory_access_mem_ops} + \text{Arbitration_time} \quad (\text{when RX/TXing frames}) \end{aligned}$$

5.4 Calculation To Account For Munich Chip Polling

The Munich chip polling effect is handled in the model by introducing a Munich Poll task which is described in section 5.3.2. The missing part of the representation is finding an appropriate think time for each Munich Poll task, to account for the Munich Poll effect. This think time represents an average period between polling requests made by the Munich chip. General formulas for the behaviour of the Munich chip polling effect can be derived from the description of its behaviour found in section 4.2.

The polling period per channel is influenced by the arrival rate of frames to be transmitted by the Munich chip. It is known from section 4.2, that the Munich chip polls all of its channels in 125 μ s. Therefore the transmission link utilization $P(i)$ can be derived to determine how many polling requests the Munich chip will make per 125 μ s cycle for channel i . One can look at $\frac{1}{P(i)}$ as the number of 125 μ s cycles that must occur before the next polling request for channel i .

$$P(i) = 1 - \frac{1}{f} V_i \lambda_i$$

f Maximum transmission speed of a DS0 channel, being 8000 bytes/sec.

V_i Average size of the frame to be transmitted on channel i .

λ_i Frames per second to be transmitted on channel i .

Using $P(i)$ and knowing the cycle time of the Munich chip, the average period between Munich chip request polls can be determined with the following formula.

$$Z_{poll}(i) = \frac{1}{P(i)} C_{munich}$$

$Z_{poll}(i)$ Derived average cycle time when channel i gets polled to see if there is anything to transmit.

C_{munich} Munich chip cycle time to check all of its ports to see if there is anything to transmit.

For each chip, the individual $Z_{poll}(i)$ terms can be averaged out, and that will be the think time for one Munich Poll task.

Another calculation can be performed to determine the fraction of utilization the Munich chip is taking from the bus.

$$U_{poll} = \sum_{i=1}^N \frac{A}{Z_{poll}(i)}$$

U_{poll} Percentage of time the Munich chip occupies the bus with polling.

A Time to check if there is something to transmit.

N Number of active channels. This number cannot be greater than 32, because the Munich chip can only handle 32 channels.

If more than one Munich chip is in the system, the formula below can calculate the total bus utilization the Munich poll effect.

$$U_{polltotal} = \sum_{j=1}^M U_{poll(j)}$$

$U_{polltotal}$ Percentage of time all Munich chips occupy the bus with polling.

$U_{poll(j)}$ Percentage of time the j^{th} Munich chip occupies the bus with polling.

M Number of Munich chips in the system.

The Munich polling representation is now complete. Each Munich Poll task is the average of all the individually calculated $Z_{poll(i)}$ for each channel i for one Munich chip (therefore i cannot exceed 32 channels). The bus utilization calculations presented are not necessary for any model parameters. These calculations can be used for model debugging purposes, by cross checking the calculated bus utilization caused by the Munich poll effect with predicted value by the model.

5.5 Calculation To Account For Polling By Software

Layered modelling does not currently represent a polling discipline. The queueing delays in a system which use a polling discipline are different from M/M/1 queueing delays, and have to be approximated. An approximation has been proposed in section 5.3.3, where a Poll Delay task supplements delay a frame or packet will experience in the

system due to polling. To determine the correct delay value, it is important to know the average cycle time of a polling loop. The expression to determine average polling cycle time was obtained from [23], and can represent many infinite memory polling disciplines. What is shown below is in essence the same equation, but adapted to modelling the Port Worker task.

$$E[C] = \frac{d}{(1-\rho)} \text{ where } d = \sum_{i=1}^N \sum_k d_{i,k} , \rho = \sum_{i=1}^N \sum_k \rho_{i,k} , \rho_{i,k} = \lambda_{i,k} b_k, \text{ and}$$

$$k \in \{ \text{HDLC_service_TX}, \text{HDLC_service_RX}, \text{LAN_service_RU}, \text{LAN_service_CU}, \\ \text{RX_framer}, \text{TX_framer} \}$$

$E[C]$ Average time to complete one cycle of the polling loop.

$d_{i,k}$ Switchover time to go from queue i, k to queue $i, k - 1$. This is polling overhead of queue k .

$\rho_{i,k}$ Offered load of queue i, k .

b_k CPU cost of servicing a job waiting in queue associated with k .

$\lambda_{i,k}$ Arrival rate of jobs going into queue i, k .

To determine the real polling cycle time of the Port Worker task, the effect of the DMA has to be factored in. This is achieved by the following formulas:

$$E_{adjusted}[C] = E[C] \frac{1}{1 - (U_{LAN} + U_{Munich} + U_{poll})}$$

where

$$U_{LAN} = \lambda_{LAN} V_{LAN} B_{LAN} + \lambda_{Munich} V_{Munich} B_{LAN}$$

$$U_{Munich} = \lambda_{Munich} V_{Munich} B_{Munich} + \lambda_{LAN} V_{LAN} B_{Munich}$$

U_{LAN} Utilization fraction used by the LAN chip on the main bus copying data to/from memory.

U_{Munich} Utilization fraction used by the Munich chip on the main bus copying data to/from memory.

U_{poll} Utilization fraction used by the Munich chip on the main bus for polling TX descriptors.

B_{LAN} Service time for the LAN chip to perform a 32 bit read/write operation.

B_{Munich} Service time for the Munich chip to perform a 32 bit read/write operation.

V_{LAN} Average size of the packet incoming on the LAN side.

V_{Munich} Average size of frame incoming on the WAN side.

λ_{LAN} Average throughput of packets incoming on the LAN side.

λ_{Munich} Average throughput of frames incoming on the WAN side.

Using the derived poll cycle time for the Port Worker task, it is possible to make a reasonable estimate on various polling delays experienced by frames or packets going through the RLAN. The Port Worker task implements a set of queues with one server vis-

iting each queue in a circular fashion. A high level representation of the Port Worker task in section 4.3, shows the order in which the functions that poll the queues get called.

If an assumption is made that the arrival of packets or frames fits a Poisson distribution, then the mean polling delay seen by a packet/frame at RX_framer, or LAN_service_RU is

$$E_{poll}[W] = \frac{1}{2}E_{adjusted}[C]$$

The approximation also assumes the poll cycle time is nearly deterministic.

When a frame goes from HDLC_service_RX to LAN_service_CU (LAN to WAN), or a packet from LAN_service_RU to HDLC_service_TX (WAN to LAN), the queueing delay experienced will be the same as the approximation above.

The reason for this is, as the number of active channels increase, the average polling cycle time is largely dominated by the for loop. Therefore on average a frame or packet will see a delay half of the polling cycle, because it is assumed that probability of a frame or packet being deposited in any of the WAN queues is uniform.

Knowing the polling order of the Port Worker along with the approximations above, it is possible to come up with reasonable end to end delays for a packet or frame going through the RLAN. Figure 22 summarizes the processing and polling orders derived from figure 9.

FIGURE 22. Polling orders for frames and packets in Port Worker

Major functions relevant to processing frames from WAN to LAN.

Order functions have to be called to process frame

rx_framer
HDLC_service_rx
LAN_service_cu

Order in which Port Worker call functions

HDLC_service_rx
rx_framer
LAN_service_cu

Major functions relevant to processing packets from LAN to WAN.

Order functions have to be called to process frame

LAN_service_ru
HDLC_service_tx
tx_framer

Order in which Port Worker calls functions

LAN_service_ru
HDLC_service_tx
tx_framer

Using figure 22, table 1 shows where each polling delay approximation is applied within the Port Worker loop.

TABLE 1. Polling Delays assigned to frame/packet queue transitions.

From	To	Polling Delay
frame enters RLAN	rx_framer	$0.5 E_{adjusted}[C]$
rx_framer	HDLC_service_rx	$E_{adjusted}[C]$
HDLC_service_rx	LAN_service_cu	$0.5 E_{adjusted}[C]$
packet enters RLAN	LAN_service_ru	$0.5 E_{adjusted}[C]$
LAN_service_ru	HDLC_service_tx	$0.5 E_{adjusted}[C]$
HDLC_service_tx	tx_framer	0

All the parameters derived in table 1 map into the Poll Delay task proposed in section 5.3.3. For example an entry LAN_service_cu_POLL in the Poll Delay task will have a service time (delay) of $E_{adjusted}[C]$. It should be noted that there is no approximation required for a frame going from rx_framer to HDLC_service_rx. This is because the HDLC func-

tion gets called first, and then the RX_framer. Therefore a frame processed by RX_framer will have to wait at least one polling cycle until it is serviced by HDLC_service_rx.

The total delay the packet will experienced in a queue serviced by the Port Worker is

$$E_{\text{total_delay}}[W] = \text{polling_delay_approx.} + E_{\text{entry}}[W]$$

where $E_{\text{entry}}[W]$ is the average time to service a request at an entry. This entry will be one of the software entries in the Port Worker task, and the value is obtained from the LQN solver. The poll_delay_approx. variable is the polling delay approximation, which is found in table 1.

Another effect of polling that has to be put into consideration is the cost of empty polls. From [23] it is known that:

$$\varphi_{i,k} = \lambda_{i,k} E_{\text{adjusted}}[C]$$

where $\varphi_{i,k}$ is the mean number of messages served per visit to queue i. For $\varphi_{i,k} < 1$, taking

$\frac{1}{\varphi_{i,k}}$ gives the number of empty polls for queue i before a packet or frame will arrive. This

value translates into the number of visits the Poll Overhead task makes to its various poll overhead entries in the CPU task. Also each Poll Overhead task will have a think time which is equal to $E_{\text{adjusted}}[C]$.

5.6 Simplifications of the model

A complete model of the RLAN has been presented in figure 21, but there are several simplifications that can be made, without the model losing any of its important properties.

The main reason for doing this is for faster simulation or solving of the model, and lack of finer grained input parameters.

- The WAN chip tasks can disappear, because there is never any contention for them. The Munich chip can transmit and receive at the same time, and should not pose any performance constraints on the system.
- Sending and receiving of acknowledgments by the system can also be removed. No measurement work done in [9] or [17] has set out and looked for values pertaining to reception and transmission of HDLC acknowledgments. To account for this effect in the system, the incoming and outgoing traffic rates can be increased slightly to include the presence of acknowledgment packets in the traffic. However models presented in this thesis do not provide any compensation for the presence of HDLC acknowledgments.

The simplified final layered model of the RLAN can be found in figure 23. It contains all the simplifications above, and approximations discussed in previous sections, which were:

- Pre-emptive priority is used in the model to better represent the bus arbitration in the RLAN. To accomplish this, three tasks were introduced at the bus layer. Each task represents operations one of the devices (CPU, LAN chip, Munich Chip) performs on memory. Memory is now a processor representing the memory and the bus, that schedules the three tasks, using pre-emptive priority. The LAN chip and Munich chip have equal priority, and the CPU has a lower one because it can be pre-empted any time.

- The Munich polling effect is accounted for by reference tasks periodically making requests to memory. This task in the model is called Munich Poll.
- Polling in the software has been accounted for in two ways. One is the Poll Delay task which inserts an average polling delay a packet or frame will experience. The other way the polling has been accounted for is by reference tasks periodically making requests to memory. This task is called Polling Overhead.

FIGURE 23. Simplified model of the RLAN system

5.7 Summary

This chapter applied the first four steps in the methodology presented in chapter 3. At the very start, the scope of the analysis is defined enough to proceed with other steps in the methodology. The scenarios of interest were determined, traced through the system and documented using Use Case Map format. From these scenarios, a layered model was built by tracing the causality through the system, and binding responsibilities to the various resources. The final step applied was, dealing with several details found in the RLAN system that layered modelling could not express directly. These details were dealt with by using approximations in the model to represent them.

With the model structure complete, the tasks and entries dictate the parameters required for the model. For the parameters that are not known, measurements on the system will have to be made, which leads to a natural progression to step 5. After that, the model can be used to answer the performance questions set out by the performance analysis study.

Chapter 6

Measurement Gathering And Model Validation

This chapter will apply steps 5 and 6 in the methodology presented in chapter 3, which are measurement gathering and model validation. For the measurement gathering step, a description of how the measurements were obtained and processed will be discussed. Model validation will compare model results against actual measurements made with the RLAN under certain loads, and how they were obtained.

6.1 The CodeTEST Profiling Tool

The CodeTEST unit [1] is a sophisticated profiling tool for embedded systems, with four major features. These are to determine average time spent in a function or task, display traces of source code executed, do coverage analysis (how much code has been executed), and track memory allocation and de-allocation. This tool will be used for its tracing and average time spent in a function or task capabilities.

CodeTEST is a hardware profiling tool which piggybacks the pins on a processor chip. From an electrical perspective, this makes the CodeTEST probe passive. This makes the CodeTEST a processor specific instrument, because pins are rarely compatible from one type of CPU to the next, even if they are from the same manufacturer.

To perform measurements with CodeTEST, one has to first prepare (commonly termed as “instrument”) the software. Instrumentation involves inserting probe tags which are

integer IDs that relate something in the code. These IDs are put into a database which relates to data the user can understand. This instrumentation process is shown in figure 24.

FIGURE 24. Instrumentation process

The new step introduced in the compiling process of the software is the “amctag” phase shown in figure 24. This software is responsible for going through the source code and inserting tags at the beginning and end of each function. The amctag software also stores values for these tags, along with the function they belong to in the instrumentation database.

When the instrumented code is running, null operations will be executed that write the tag IDs into a reserved memory space. The CodeTEST unit then monitors bus activity and waits for memory accesses to occur in the reserved memory space. The CodeTEST unit then accumulates these tags, and periodically sends them to the CodeTEST front-end software. The front-end software with the help of the instrumentation database, outputs results a human user can understand. The entire operation just described is illustrated in figure 25.

FIGURE 25. Data gathering when code is running

6.1.1 Instrumenting in a RTOS

Since the RLAN software runs on a real-time kernel, the software has to be hand instrumented in certain places. The reason for this is, CodeTEST must know what functions are executed in what context, otherwise there is no way the CodeTEST can report correct measurements. For example, say there is a program with 2 tasks running on it. One of these tasks executes function A, and for some reason this task is switched out of context. The other task is switched in, and starts executing function A also. How is the CodeTEST supposed to determine how long it took to execute function A without knowing about the context switch? In order for CodeTEST to know what context a function executed in, the instrumented program must tell the unit when there is any chance of a context switch occurring in the program.

To achieve this, two special macros in C were made that tells the CodeTEST unit when a task may switch in or out of context. These are called `CODE_TEST_TASK_ENTER`, and `CODE_TEST_TASK_EXIT`. Below is an example of a task making a kernel call

`KN_receive_unit()`, which requests access to a critical section using a counting semaphore.

```
CODE_TEST_TASK_EXIT;
    KN_receive_unit(my_token, KN_WAIT_FOREVER);
CODE_TEST_TASK_ENTER;
```

If the critical section is occupied for some reason, the task which requested access to the critical section will block and be switched out of context. The CodeTEST unit is informed of this context switch because of the `CODE_TEST_TASK_EXIT` macro. Once the critical section is released, the task which requested access to the critical section will be switched into context by the kernel. After the task being switched into context, the next set of instructions executed will be the `CODE_TEST_TASK_EXIT` macro. This informs the CodeTEST unit that the task is running again, and allows the unit to report proper execution times for functions executed by different tasks.

6.2 Goals Of Measurement Experiments Performed On The RLAN

The model in Chapter 5 (steps 1 - 4 in methodology) has identified the need to obtain measurements for six software functions, listed in table 2.

TABLE 2. List of functions Port Worker calls, and purpose

Function Name	Purpose Of Function
<code>RX_framer</code>	assemble a frame from Munich chip receive descriptors
<code>TX_framer</code>	disassemble a frame into a descriptor format the Munich chip understands
<code>HDLC_service_RX</code>	service the WAN (DS0) PPP link on the RX side
<code>HDLC_service_TX</code>	service the WAN (DS0) PPP link on the TX side
<code>LAN_service_CU</code>	put frame into LAN descriptor format and send a command to the LAN chip
<code>LAN_service_RU</code>	assemble a frame from descriptors, and send it to the correct DS0 channel

All measurements performed on the functions specified in table 2, are for frame or packet sizes of 1500 bytes. Measurements under various traffic patterns was not of interest. The reason for this is, the processing cost per frame for every function will be the same regardless of the incoming traffic pattern.

It should be noted that the CPU cost for the bridge/routing of a frame has been lumped into HDLC_service_RX and LAN_service_CU, because these two functions call it automatically.

6.3 Measurement Setup

The objective of the experiments was to determine the CPU cost of each of the six functions described in section 6.2. An assumption was made that for a given frame size, the CPU cost for executing a function will be the same for any number of active channels. This assumption is practically trivial, because the RLAN has no caching infrastructure which would make measurements load dependant. Most importantly the assumption simplifies the experiment setup required to obtain measurements. The hardware setup used to obtain measurements on the RLAN system can be found in figure 26.

FIGURE 26. Measurement setup in the lab

Where:

- PC1 was a 100Mhz Pentium clone running FTPserve under DOS.
- PC2 was a 33Mhz 486DX laptop computer running OnNet 2.1 Windows 3.1 FTP client.
- 5250 is a box that converts an ethernet connection to an ISDN connection. The 5250 was configured in leased line mode (sends hello packets).
- RLAN consisted of System Controller Card with a i960CF processor, a BRI card that can handle up to 30 connections, and Munich throttling was disabled. RAC 3.3.0 release software was used with the required modifications done in [15] to get the RLAN to work with the CodeTEST. The version of the BRI code was 3.0.0.
- The entire setup was isolated from the backbone, except for communication required between the Sun Workstation and the CodeTEST. This does not affect the experiment in any way.

As shown in figure 26, there are only two PCs, which means that only one connection was active during all these measurements. The link was setup as a PPP singlelink connection, with the RLAN in bridging mode. Other experiments have been performed with various different configurations like multilink and compression, the results from those experiments can be found in [17].

6.4 Approach Taken To Make Measurements

Since the RLAN uses a polling discipline to service all its queues, some CPU time will be wasted checking queues that have no available work. The RLAN with one active channel is mostly idle, and it would be wrong to take the average CPU cost per visit as a value for doing useful work. The reason for this is that polling overhead of empty polls will be factored into this average, lowering the actual CPU cost of doing useful work. This is illustrated in figure 27. Therefore there are two values of CPU cost which have to be obtained from a function that is being called in a polling system. One of the CPU costs is polling overhead, and the other is the cost of doing useful work.

FIGURE 27. Histogram of a function being called in an experiment

Another problem encountered was that the CodeTEST would return how much time was spent in a function. For example, if function A calls function B, CodeTEST will tell you how long the CPU spent in A and B separately. It was desired to have the aggregate execution time of A and B for function A. To make matters worse, these functions call other functions and can generate a complex call structure spanning thousands of lines of code. The question is how does one capture the idle work and useful work, without knowing the call structure of each function?

To avoid understanding the various call structures, traces were generated every time a function was called. Every time a function is called, a trace begins and ends when the function of interest enters and exits respectively. The trace follows the whole call sequence and indicates how long the CPU spent executing various functions. Since the RLAN is continuously running, the CodeTEST will record many traces of a specified function until the CodeTEST's on board memory is full. The concatenation of traces (can be in the order of thousands traces) is stored in a text file, which is run through an AWK script called **tracefilt**. The script **tracefilt** parses the text file, to determine where the trace starts and finishes for a function of interest. **Tracefilt** will then aggregate CPU cost found between the start and end points of the trace. These numbers are written into a text file, which a program in C called **histo** runs through and generates a histogram of the data. The process just described is illustrated in figure 28.

FIGURE 28. Process trace data goes through to generate an execution profile

The histogram output from the **histo** program is the actual execution profile, which in the case of this system looks much like figure 27. Plotting the output from the **histo** program, execution times observed for a function would be on the X-axis, and the number of times the function executed for a particular time span will be on the Y-axis. The advantage of using these execution profiles is, generally nothing is missed, provided your trace time is sufficiently long enough. The reason for this is, no measurement is averaged, which makes it possible to see peculiar behaviour of the software. In the case of the CodeTEST, the trace was active for about one to two seconds, which gave an opportunity of roughly 20 frames to go through the system for measurements when the RLAN was doing useful work.

By comparing execution profiles with data and no data going through the system, it is possible to separate polling overhead from useful work. This is shown in figure 29.

FIGURE 29. An example of a execution profile

CPU cost of useful work that a function performs has to be derived from the execution profile. Comparing profiles when data and no data is going through the system gives polling overhead, and polling overhead plus useful work. To isolate cost of just useful work, one has to subtract polling overhead from the part that gives useful work plus polling overhead.

Unfortunately theory and practice do not always agree perfectly. When data was going through the RLAN, some of the profiles resembled others with no data going through. This behaviour was noticed only with functions to do with the WAN. It was later discovered that the 5250 sends “hello” packets when it has not received any data for some period of time. Therefore it turned out the idle experiments to determine polling overhead were not perfect. Observations of useful work were made in functions that should not have been doing work. For example when traffic goes from WAN to LAN, one would expect HDLC_service_TX never to do any useful work because the traffic direction is the other way. Looking at obtained profiles, useful work was being done for HDLC_service_TX, and if this is not the case, then this may indicate a problem in the software. The behaviour observed can be explained by flow control at the TCP/IP level, because TCP/IP will send ACKs which must travel through the RLAN.

All traces were started after 10% into a 300Kbyte FTP transfer, to give TCP/IP some time to stabilize the throughput of the transfer. For idle measurements, traces began after 30 seconds of inactivity on the channel. All traces were done in deep mode [1], and all traces stopped when the CodeTEST’s tag buffer was full.

6.5 Measurement Results For Software Functions

A summary of the relevant measurements obtained within the scope of the analysis can be found in table 3. The scope of these measurements was to consider data travelling from LAN to WAN, WAN to LAN, compression disabled, singlelink PPP, and 1500 byte packets. Other measurements with multilink and compression were also performed, which can be found in [17]. The results in table 3 can also be found in [17].

TABLE 3. Functions and service times selected for entries in the model for 1500 byte packets

software functions	poll overhead	useful work done
TX_framer	0.000006	0.000040
HDLC_Service_TX	0.000016	0.00057
LAN_Service_RU	0.000007	0.00016
HDLC_service_RX	0.00003	0.00039
LAN_Service_CU	0.0000047	0.000041
RX_framer	0.000005	0.000057

It should be noted that the CPU cost of executing the bridging/routing functions in the RLAN was aggregated into either HDLC_Service_RX or LAN_service_RU. Which of the two functions the cost of bridging/routing is aggregated into is dependant upon the direction a packet or frame is going in.

6.6 Measurement Results For Hardware Operations

Parameters for certain hardware operations had to be obtained, as outlined by Chapter 5, and can be found in table 4.

TABLE 4. Obtained Hardware Measurements

Hardware Operation	Execution Time On Main Memory
Munich Chip Read/Write	240ns per 32bit word
Munich Chip Poll	720ns per channel
LAN Chip Read/Write	240ns per 32bit word
Arbitration Time	70ns per bus access for Munich and LAN chips

No experiments were done to obtain the values found in table 4. They were obtained by talking to engineers who have experience dealing with the various hardware in the RLAN. All the operations above were based on how many clock cycles it took to complete. Since the clock cycle was known, and the engineers knew it took certain amount of clock cycles to perform an operation, an execution time for an operation can be derived. Data sheets and technical manuals on the LAN and Munich chips proved to be incorrect at times. This is of no real surprise, because technical manuals for chips have a tradition of never being completely accurate.

Recall that the Munich chip and LAN chip have an on board FIFO memory. When the chips are receiving data, in the model it was assumed that the average time one of the chips made a request to the bus was when the FIFO buffer was half full. For transmission, it was assumed that every time the chip would have access to the bus, it would transfer enough bytes to fill the transmit FIFO buffer. In the case of the Munich chip, the transmit FIFO buffer to be filled would be for a particular channel, and the LAN has only one transmit buffer. The Munich chip has an 8 byte FIFO buffer for reception and transmission, and the LAN chip had a FIFO buffer of 32 bytes for reception and transmission.

6.7 Comparing Model Predictions With Real Observations

With the required values for all the parameters obtained, the model is completed. To gain confidence of the model's correctness, some validation should be made. Model validation is step 6 of the performance methodology. Unfortunately a true validation cannot be done, because no delay measurements on the RLAN were made. Comparing delays from model predictions with real measurements is the best validation, because delay predictions are the most difficult to obtain. This is because delays are usually a function of more than two parameters. Hence if any one of these parameters is wrong, this can be reflected in an incorrect delay prediction. Therefore comparison of delay results gives better coverage across the model.

Predicted utilization results from the model will be compared with actual measurements made in [9]. Comparison of utilizations is not as good as delays, because utilization is a function of throughput and service demand. That means utilization is really only a function of one parameter which is throughput, because service demand is known by measurement. Figure 30 shows the essence of the experiment setup used in [9] to obtain the measurements for validation. More detail on this experiment setup can be found in [9].

FIGURE 30. Diagram of experiment setup for model validation

The experiment setup for operation under load, consists of two RLANs back to back with their DS0 channels connected to each other. This is done because it is a fast and manageable way of scaling the system to a large number of active channels. The packet generator is always in one of the two positions shown in figure 30.

The following four tables shows comparisons of utilizations between the model and measurements from [9], for various operations occurring on the bus.

TABLE 5. 30 Channels, 500 byte packets, LAN to WAN, 14.2 fps per WAN link

Port Worker Function Name	Model Function Utilization Of Bus	Measured Function Utilization Of Bus [9]
Munich_TX_memory ops	0.027216	N/A
LANchip_RX_memory ops	0.026433	N/A
Total DMA utilization	0.025	0.024
TX_framer	0.063015	0.065
HDLC_Service_TX	0.32403 *	0.222
LAN_Service_RU	0.06491	0.044
HDLC_service_RX	0.37242	0.241
RX_framer	0.048106	0.050

* estimate of useful work for HDLC_service_tx obtained from [9].

TABLE 6. 30 Channels, 500 byte packets, WAN to LAN, 12.8 fps per WAN link

Port Worker Function Name	Model Function Utilization Of Bus	Measured Function Utilization Of Bus [9]
Munich_RX_memory ops	0.024674	N/A
Munich_Polling	0.19243	N/A
LANchip_TX_memory ops	0.024328	N/A
Total DMA utilization	0.238	0.228
RX_framer	0.069627	0.069
HDLC_Service_RX	0.43314	0.283
LAN_Service_CU	0.02497	0.017
TX_frame_bus	0.044	0.044
HDLC_Service_TX	0.13658	0.022

TABLE 7. 60 Channels, 1500 byte packets, LAN to WAN, 5.03 fps per WAN link

Port Worker Function Name	Model Function Utilization Of Bus	Measured Function Utilization Of Bus [9]
Munich_TX_memory ops	0.05413	N/A
LANchip_RX_memory ops	0.051775	N/A
Total DMA utilization	0.105	0.04
TX_framer	0.075	0.074
HDLC_Service_TX	0.36138	0.262
LAN_Service_RU	0.04970	0.0303
HDLC_service_RX	0.3279	0.241
RX_framer	0.052	0.044

TABLE 8. 60 Channels, 1500 byte packets, WAN to LAN, 5.04 fps per WAN link

Port Worker Function Name	Model Function Utilization Of Bus	Measured Function Utilization Of Bus [9]
Munich_RX_memory ops	0.048931	N/A
Munich_Polling	0.3784	N/A
LANchip_TX_memory ops	0.0471	N/A
Total DMA utilization	0.473	N/A
RX_framer	0.088	0.089
HDLC_Service_RX	0.2434	0.182
LAN_Service_CU	0.017353	0.0127
TX_frame_bus	0.0247	0.028
HDLC_Service_TX	0.09533	0.0148

6.8 Discussion Of Utilization Results

In tables 5 - 8, many of the model values correspond closely to the measured values, but there are also some substantial differences. The major differences appear in functions HDLC_Service_RX, HDLC_Service_TX, LAN_Service_CU, and LAN_Service_RU. All parameters for the model are obtained from [17], with the exception of one (which is marked in table 5). One of the reasons these discrepancies exist is, the measurement technique used was different from [9]. Looking at the instrumented source code from [9], it is found that significant portions of code were missed in the instrumentation. In particular the polling overhead associated with each of the four functions mentioned above. Table 9 compares polling overhead measurements from [17] and presented in section 6.5, with measurements presented in [9].

TABLE 9. Comparing polling overhead measurements

Function Name	Measurements from section 6.5	Measurements from [9]
HDLC_service_TX	1.6e-5	8.46e-6
HDLC_service_RX	3e-5	1.5e-6
LAN_service_CU	4.7e-6	3.6e-6
LAN_service_RU	7.0e-6	5.4e-6

For HDLC_service_RX, it was possible from [17] to go back and look at raw traces to find CPU costs of functions that HDLC_service_RX called, and were missed by [9]. The results of these findings are found in table 10.

TABLE 10. CPU costs missed by [9] for HDLC_service_RX polling overhead

Function missed	CPU cost from [17]
sort_out_ncptimeout	7.4e-6
sort_out_timeout	7.2e-6
total CPU cost missed	14.6e-6

If total CPU cost missed for HDLC_service_RX in table 10 is added with the measurement reported in [9], the result will come very close to what is presented in section 6.5. Unfortunately this type of double checking could only be done with HDLC_service_RX, because portions of code that was not instrumented in the other three functions, did not make any function calls. Since CodeTEST reports CPU cost per function, no measurements were available for the code fragments.

The average ratio of finding work to do in a port versus no work was 1:27, with the exception of LAN to WAN 1500 byte packet case where the ratio was on average 1:40. Ratios of these magnitudes contribute greatly to the discrepancies found in tables 5 - 8 if the polling overhead was not properly measured. When measurements from [9] were put

into the model, all predicted utilizations matched with the measured utilizations to within 3%. In [9], RX and TX framer were instrumented such that it captured all the CPU cost. For all four cases the match between model and measured utilizations matched to some degree, except in table 8 which cannot be explained.

It must be emphasized that one cannot expect very accurate correspondence between model predictions using measurements from [17] and section 6.5, against actual measured values of [9]. One of the primary reasons is the measurements found in [9] were obtained from version 3.4.3 of the software, yet the model uses measurements obtained from version 3.3.0. The other reason, a much more weaker reason is that the measurements in [9] were obtained when the software was under load. This can bring better on board processor caching into the picture, which can yield to some lower results than what the model predicts. This reason is weak because some results were predicted fairly accurately all things considered.

Even though the model uses measurements from an older version of the RLAN software, no adjustments will be made to compensate for any discrepancy found when comparing measurements in [9].

6.9 Summary

This chapter applied steps 5 and 6 in the performance methodology presented in chapter 3. Now that the structure of the model has been determined, the parameters needed are identified. The necessary measurements were done on the software components, using the Codetest profiling tool. With the completed model, the predictions were compared to

actual performance measurements to gain confidence in the model's correctness. The values being compared were the percent bus utilization of various software and hardware operations. Some discrepancies that were found are explained by differences between the calibration experiments and the validation experiments; analysis suggests the calibration data is of better quality for those points. At this stage, nothing can be said about the reasonableness of the delay results from the model, without further investigation.

With the model completed and checked, the next and final step is to use the model in different ways to answer performance questions in chapter 7.

Chapter 7

Exploiting Model For Capacity Planning And Architectural Re-design

Chapters 5 and 6 have derived a model for the RLAN system. Using this model to answer performance questions is the last step in the performance methodology presented in chapter 3. This chapter will demonstrate three uses for the model, which are system capacity, non-Poisson arrival rates of frames/packets, and architectural re-design.

7.1 Goals Of The Performance Study

The goals of the performance study is to understand the performance of the RLAN system. This is done by determining the capacity of the RLAN with infinite memory, under Poisson and bursty arrival of frames and packets. There is no quality of service criteria to relate to the delay seen at the LAN or WAN queue. Therefore a rule of thumb will be used to determine when the system is at its maximum recommended operating point. Since the RLAN is an open system, the rule of thumb has to be developed for typical open system behavior which is shown in figure 31.

FIGURE 31. Defining operating range for an open system

The region on the delay curve which defines the maximum recommended operating point is slightly before the curve starts to drastically increase exponentially. For the RLAN, this point occurs when the bottleneck device is utilized to 70% to 80%. If the bottleneck device is a polling device, then the maximum operating point is when the device is doing 70% to 80% useful work.

Anything to the left of the maximum operating point is called the operating range of the system. Points to the right of the maximum operating point are practically not achievable. This is because delays near saturation are extremely sensitive to system workload. Since only portions of the system are modeled, any extra work performed by the system can easily double or triple the delay.

After the capacity experiments are completed, a hypothetical hardware re-design is proposed in an attempt to improve the capacity of the RLAN.

7.2 Predicting Capacity Of The Expressway RLAN With Poisson Arrivals

This capacity study will investigate how many channels can be supported given a constant arrival or exit rate of frames across all active WAN links. Figure 32 shows the average waiting time at the WAN queue, as the number of channels increase. For each new channel that is added on the WAN side, the LAN side throughput increases by 5.3 frames per second (fps) to service the new connection. Each frame is 1500 bytes, data travels from LAN to WAN, or WAN to LAN, and all connections are uncompressed PPP singlelink. The WAN queue is monitored because if the delay becomes larger than 1 second, the packet will be discarded.

FIGURE 32. Average waiting time experienced in the WAN queue

Another capacity experiment can be performed on data going the reverse direction (WAN to LAN). Figure 33 shows the average waiting time experience by a frame in the LAN queue. Just like with the WAN queue, if the delay in the LAN queue exceeds 1 second, the frame will be discarded.

FIGURE 33. Average waiting time experienced at the LAN queue

For both the LAN to WAN and WAN to LAN cases, the bottleneck was the main bus. This result is expected, because every device in the system requires the use of the main bus/memory combination.

Using the delay curves found in figures 32 and 33, capacity of the system can be determined. For the range of active channels the model was run, on average the system will not drop any packets or frames. Looking at the delay curve found in figure 32, the maximum operational range for 1500 byte packets moving LAN to WAN is roughly 130 channels.

For the reverse direction, two design possibilities are explored. One is with Munich chip throttling turned on, and the other with it turned off. For the throttling off, the capacity of the system is about 70 channels, and with throttling about 85 channels. Munich throttling gains an extra capacity of roughly 15 extra channels. This capacity is gained by

enabling the Munich chip to queue up polling requests. Unfortunately all polling requests by the Munich chip have to be served within 125 μ s. This fact translates into small savings for Munich chip bus occupancy, because only arbitration time has been reduced as discussed in section 5.3.5. Arbitration time accounts for 10% of an entire polling operation for one channel. Therefore throttling the Munich chip will at best give 10% less occupancy on the bus by the Munich chip.

Overall the WAN to LAN capacity results predict roughly 45 to 60 channels less capacity than the LAN to WAN case given same load. This can be attributed to the Munich polling effect, because in the WAN to LAN case, the RLAN is not transmitting anything on any DS0 link. For the LAN to WAN case the Munich polling effect has no significant effect on performance because the RLAN is always transmitting frames on all DS0 links.

Another use of the delay curves is to determine the average memory requirements for the queue. Knowing the average delay at a given number of channels, and service time of HDLC_service_TX or Service_LAN_CU, it is possible to determine the average queue length. Once the average queue length is known, it is possible to determine the average memory requirements based on the average frame size. The formula below can determine the average memory required, which is based on everything just discussed.

$$AverageMemoryRequired = \frac{AverageDelay}{ServiceTime} AverageFrameSize$$

7.3 Polling Discipline

The second performance study is for bursty traffic with variable packet or frame sizes. Some extensions to the model were made as a part of the study:

- Bursty traffic sources were included in section 7.5.
- Further sub division of interprocess messages was introduced, according to the size of data packets to be processed. A server with this feature is called a “selective server” found in section 7.4. This could have been done with a large number of entries, but that approach would have been difficult to manage.
- A full implementation of a polling server that also includes variable packets sizes is described in this section.

These extensions were used to get the results found in section 7.7 and do a comparison study in section 7.8.

The approximations necessary for an analytic solution of a polling discipline have not been developed, thus one has to resort to simulation in order to predict delays accurately. This brings the introduction of a new task type in ParaSRVN, which implements an exhaustive polling discipline.

There were two implementation problems encountered when adding the new polling task into ParaSRVN. One of the problems was the need to assign different service times to entries for phase 1 execution. It was shown in [9] that different sizes of packets, had different CPU costs. This is different from the analysis done thus far which always assumed a fixed packet size. The other problem was to properly represent the behavior of the Port

Worker task, entries have to be replicated. The reason for this was, each entry would represent the software polling an active DS0 link. The polling task would then poll the entries in the same manner as the Port Worker task does in the RLAN. A further constraint to entry replication was, it has to be done in a memory efficient manner, with minimal modification to the SRVN input grammar. Generating LQN files with manually replicated entries, and appropriate call paths was difficult, and not memory efficient.

With the given limits, to solve the entry replication problem, the polling task has the capability of replicating and managing queues for each entry. When a request is made to an entry, the request will be routed into one of the replicated queues, by the polling server with no cost incurred. The polling task polls and services all the replicated queues for all the entries that belong to the polling task. To solve the problem of different packet sizes having different execution times, entries in the polling task are allowed to execute different classes of requests. Every request class has its own service time defined by the user, but no entries in the polling task cannot have more than one phase of execution.

The polling discipline implemented in ParaSRVN is an exhaustive polling discipline. With the use of asynchronous or forwarding calls, the polling discipline can support a chain of entries/queues that are polled. This is a powerful feature in the analysis of the RLAN, because one can perfectly describe the polling order of the Port Worker task without compression active in the model. Therefore the simulation will yield to very realistic delays a frame experiences going through the RLAN, because of a proper representation of polling in the layered model.

Figure 34 shows how the layered model parameters are re-defined to exhaustive polling discipline parameters.

FIGURE 34. Re-definition of LQN parameters for the polling discipline

Priority: Number of queues per entry			
	phase 1	phase 2	phase 3
Think Time	N/A	N/A	N/A
Execution Demand	Poll Overhead	Useful work phase 2	Useful work phase 3
COV	same	same	same

• • •

Poll Overhead - phase 1:

Polling overhead is defined as the CPU cost of checking an entry, or replicated entry to see if there is any processing to do.

Useful work - phase 2 - Maximum number of phases:

Useful work is defined as the CPU cost of processing a single message in the queue waiting to be served. The poll server will incur the CPU cost of processing the request, depending upon the phase of execution specified in the message. This cannot be less than phase 2, or greater than the maximum allowable phases of the simulator.

Number Of Queues Per Entry:

Number of queues replicated for a given entry.

There are two remaining implementation issues left to be discussed about the exhaustive polling discipline. The first one is the polling order of all the defined replicated entries. The second issue is, what happens when a incoming message has a queue id greater than the number of replicated entries. The solution to the second issue is solved by the following fanning in calculation performed once this condition has been detected.

$$DestinationQueueInEntryA = \frac{QueueIdInMessage}{MaxNumberOfReplicatedEntriesInTask} \times NumberOfReplicatedEntries$$

This calculation takes the queue id as a percentage of the maximum number of replicated entries in the task. The percentage then determines which replicated queue the message will go into. If the calculated queue id is still greater than the number of replicated entries, the simulation will stop. This indicates that there is an error in the model, because there is no queue for that message in ANY of the entries in the polling task. When messages are being fanned into replicated entries, their original queue id is preserved. This gives the capability of the messages to fan out again after being served in a chain of polling queues.

The other issue mentioned earlier is what the polling order of the polling task is, and how is polling order affected by an uneven amount of queues replicated within one task. The order in which entries are polled is determined by the order in which they are defined in the LQN file. Order in which the queue sets for each entry is polled is defined in figure 35.

FIGURE 35. Polling order implemented in the exhaustive polling discipline

Polling is done by checking all the entries with a certain queue id, and a poll cycle is defined when all the entries with all their queue ids have been checked. The polling algorithm will skip any entry that does not have the current queue id being polled in other entries. This happens when an entry does not have as many replicated queues as other entries within the polling task. In figure 35, when the queue id is 4, entry C is skipped because all its entries have been already polled.

7.4 Selective Server

This is a server that executes any request class in the same manner as the polling server did. There are two differences between the selective server and the polling server. One is the discipline used in the selective server is FIFO. The other difference is the selective

server cannot replicate queues for each entry, as there is no need for this. As in the poll server, each entry in the selective server cannot execute in more than one phase.

7.5 Bursty traffic analysis

It has been discovered that packet arrival rates on computer networks usually do not follow a Poisson distribution [25]. Packet arrivals have been observed to have a more bursty behavior, hence making the arrival of a packet dependant on previous arrivals. This is quite different from Poisson arrivals, which assume the arrival of a packet is independent of previous arrivals. Bursty traffic comes from the nature of client/server applications that are typically used on networks. Applications like FTP, World Wide Web, and Push Technologies promote enough data transfer to a destination that cannot fit into one packet. Therefore for every request made to a server by a client, a train of packets will be received that contains requested information. Packet arrivals in this situation will be dependant on previous arrivals, because packets are known to arrive in bursts.

A bursty traffic profile has been designed by Gandalf, to mimic the bursty properties of packet arrivals found on computer networks. Using this profile on the RLAN model will give a better indication to how well the system will perform with more realistic incoming traffic patterns. Below is a illustration of how the bursty traffic profile is defined by its parameters.

FIGURE 36. Description of the Bursty traffic profile

A - Burst size (number of packets within a burst)

B - Quiet time between bursts

C - Packet size

D - Packet inter-arrival time between in a burst

Note: All four parameters are exponentially random variables

7.5.1 Definition Of Three Traffic Profiles

The RLAN model with the new extended features was subjected to three different bursty traffic profiles, which were the corporate, ISP, and worst case profiles. What defines these profiles is the amount of traffic going through the RLAN, and a ratio of transmission to reception of traffic on the ISDN side of the RLAN. These profiles were specified by Gandalf, based on data provided by their customers on their RLAN usage. A summary of the distinct profile characteristics can be found in table 11.

TABLE 11.Definition of traffic profiles

Profile Type	Receive : Transmit	Link Utilization Receive : Transmit (w.r.t. WAN)
Corporate	1 : 1	10% : 10%
ISP	1 : 9	1.1% : 10%
Worst Case	1 : 3	13.3% : 40%

Link utilization can be used to calculate the average throughput on each DS0 link, knowing that the maximum possible throughput is 8Kbytes/sec per DS0 (100% link utilization). Other parameters which are common to all three profiles can be found in table 12. Note the letter in parenthesis for the very left hand column refers to figure 36.

TABLE 12.Other bursty traffic parameters

Bursty Traffic Parameter	Receive	Transmit
Average Frame Size (C)	80 bytes	500 bytes
Max. Frame Size Truncation	1500 bytes	1500 bytes
Min. Frame Size Truncation	64 bytes	64 bytes
Average Frame Interarrival (D)	0.01 s	0.0625 s
Average Burst Size (A)	5 frames	5 frames
Burst Size Truncation	20 frames	20 frames
Quiet Time (B)	Requires Calculation	Requires Calculation

The only missing parameter is the quiet time between bursts, or the burst inter-arrival time. This parameter has to be calculated using the following formula.

$$QuietTime = \left(\frac{1}{LinkUtilization} - 1 \right) \times AvgBurstSize \times \left(\frac{AvgFrameSize}{WanSpeed} \right)$$

7.5.2 Bursty Reference Task

For bursty traffic, the approximations necessary for an analytic solution have not been developed, thus one has to resort to simulation in order to predict delays accurately. A bursty reference task was implemented in ParaSRVN to support the traffic profile defined in figure 36. It is the only client that is capable of utilizing the new features of the polling server. This was done by adding two pieces of information into a message that is sent when a client makes a request to a server in ParaSRVN. First piece of information is a queue id, which indicates the proper queue the request goes into from the set of queues that belongs to the entry. The second piece of information is the request class of the incoming request. Although new information has been added to the messages passed between client and server, this change is completely transparent to all components of layered modelling. The reason for this is, all the components in layered modelling do not look at the new information, but relay this information through a request to another server.

The bursty reference task manages any number of independent bursty streams by itself. Therefore there is no need to make duplicates of it with the multiplicity parameter, which saves significant amount of memory. Figure 37 shows the re-definition of input parameters for specifying the bursty reference task. Letters in parenthesis found in figure 37 refer to the parameters described in figure 36.

FIGURE 37. Re-definition of LQN parameters for the Bursty client

Priority: Number of independent bursty streams			
	phase 1	phase 2	phase 3
Think Time	Quiet Time (B)	Burst Size Truncation	upper_limit
Execution Demand	Avg. Burst Size (A)	Avg. Interarrival Time (D)	lower_limit
COV	bottom limit on request classes	# of different request classes	avg. frame size (C)

The other parameters not found in figure 36 are discussed below.

Burst Size Truncation:

The average burst size is obtained from an exponential distribution. From the distribution, it is possible to get unrealistically high values, and therefore provisions have been made to set an upper bound on the random variable. Any random value beyond the specified upper bound will just become the upper bound.

Bottom Limit On Request Classes And Number Of Different Request Classes:

These two parameters define the upper and lower bounds on which requests classes will be generated by the bursty reference task. Bottom limit parameter is the desired lowest request class number that will be generated. The number of different request classes parameter is a offset from the bottom limit in how many classes will be generated. One restriction that arises from these two parameters is the sum of these two parameters cannot exceed the number of maximum phases in the simulator, which is now set at 15. Also the bottom limit cannot be less than 2, because of implementation issues. In layered modeling, phase 0 does not exist, and phase 1 is defined as polling overhead for the exhaustive polling discipline.

Average Frame Size:

This is the average frame size in terms of bytes, and must lie between the upper and lower limit parameters described below. As mentioned earlier, each frame size has a different service time, which has its own request class. Therefore to get the average frame size parameter from the bursty traffic profile into the model, one has to map that parameter to an average request class number.

Upper And Lower Limit:

These are the upper and lower limit for frame size in bytes possible for a given experiment. If the randomly selected frame size exceeds these limits, it will be truncated to the closest of the two limits.

Number Of Independent Bursty Flows:

This parameter specifies the number of independent bursty streams the bursty reference task can generate. Each of these streams must have some kind of mapping to a replicated entry in the poll server. Every stream is identified using the queue id field, in the request message that a client sends to a server. That way different streams can be routed to different queues in the poll server.

A noteworthy point to mention is that the bursty client can only make asynchronous calls. The reason for this is, if the bursty traffic generator made synchronous calls, the behavior of packet arrivals would be governed by how the system is performing. This means the model would become a closed system which it is not.

7.6 Using New ParaSRVN Extensions To Model The RLAN

With the new extensions added into ParaSRVN, the burst analysis can now be done more accurately. The next section will describe how the new components are used to model the RLAN.

7.6.1 Defining The Port Worker Using The Poll Server

The order in which the Port Worker calls various functions in the actual code, is identically represented in the layered model. As the number of channels varies in the RLAN for a model run, so do the replications of queues for RX, TX framer, and HDLC RX, TX. When frames pass through the RLAN, they end up in another queue in the system waiting to be processed by the Port Worker. This can also be expressed in the polling discipline, by forwarding or asynchronous request to another entry in the Port Worker task as shown in figure 38.

FIGURE 38. Polling Task entries, note calls are asynchronous

With the 3 new ParaSRVN extensions already described earlier, figure 39 shows the new version of the model.

FIGURE 39. A modified model of the RLAN with the new extensions

In the model shown in figure 39, the WAN and LAN driver are now bursty references, the WAN and LAN chips are selective servers, and the Port Worker task uses the polling discipline. Looking at figure 39, one will notice that any large number of replications occur only in the operation tasks, as opposed to previous models presented. This demonstrates that the model will use less memory because allocating a high number of tasks in ParaSRVN can get quite memory consuming. Other than that, the essence of the model is unchanged.

7.6.2 Comment On Model Modularity

Instead of having one polling task with all the polling overhead and useful work embedded into it, one can expand the polling model to the one shown in figure 40. The polling task has been expanded to three tasks which are the polling task, polling overhead task, and the useful work task. The polling task uses a polling discipline, and the polling overhead task contains all the polling overheads defined in the polling task. The useful work task stores all the CPU costs of doing useful work for each entry in the polling task. When the poll task is polling, it will make one call to the poll overhead task to induce the cost of checking a queue for that entry. If there is something to be serviced in the queue, one call will be made to the useful work task to induce the cost of servicing one request.

The motivation of expanding the tasks as shown in figure 40 is to keep the model modular, and hence more flexible. For instance, moving from a model defined in chapter 5 to the one in figure 40 would require less changes if polling work was separated into two tasks. If one resorted to the one task approach, the model would require more changes to it. This is because all the parameters are embedded in one task, and in a non compatible definition format with the rest of the layered model components. Another advantage of the 3 task approach is, it is more of a clear way of illustrating how the work in the poll task is distributed, and is more consistent with the layered model semantics.

FIGURE 40. Another way of expressing polling in the model

7.7 Results Using Three Different Bursty Traffic Profiles

The following figures 41, 42, and 43 plot average end to end delay of a frame or packet travelling in either direction. Each average delay plot uses one of the three bursty traffic profiles defined in section 7.5.1. End to end delay is defined from the time a packet or frame appears in the descriptors, to the point when the packet or frame is ready to be transmitted out via DMA on the other side.

FIGURE 41. Corporate profile, with bursts, plotting end to end delay

FIGURE 42. ISP profile, with bursts, plotting end to end delay

FIGURE 43. Worst Case, with bursts, plotting end to end delay

An emerging pattern from all three profile runs is that traffic travelling from WAN to LAN experiences more delay than traffic going LAN to WAN. This has been consistently seen in all analysis done thus far. The reason for the delay differential in the three bursty traffic cases is different from analysis done in section 7.2. When a frame travels from WAN to LAN, it experiences a full polling cycle delay, because of the way the Port Worker task polls the queues. This polling order inefficiency is discussed section 5.5. If the poll cycle time is subtracted from the WAN to LAN delay, the result will be a delay curve which almost matches the LAN to WAN delay curve. In figure 44, one poll cycle time has been subtracted from the WAN to LAN curve to show how similar it is to the LAN to WAN curve.

FIGURE 44. WAN to LAN delay curve with one poll cycle time reduced

When the number of channels increased, the WAN to LAN delay is still slightly higher than the LAN to WAN delay. This is explained by the fact that frames come in from the WAN side about six times faster than outgoing frames within a burst. Therefore this will tend to create longer queues within a burst, which will increase end to end delay.

Looking at figures 41, 42, and 43, the ISP profile yields the highest capacity of the three experiment runs. This is expected because the ISP profile drives the least amount of traffic through the system. The worst case profile gives the worst capacity, but it turns out that the corporate profile does not do much better, giving only 15 channels more.

7.8 Approximations For Faster Solution Times With Bursty Traffic

Simulations for the three different profiles used considerable amount of resources and time to get results within reasonable confidence intervals. Serialization of work caused by the exhaustive polling discipline and bursty arrivals will increase simulation time by reaching desired confidence intervals slower.

One way to speed up simulation time is by putting in approximations in terms of layered model semantics. This will enable the model to be eventually solved analytically, which should speed up solution time dramatically. Approximations for polling have been described in chapter 5, but how good are they? There are no easy ways to approximate for the bursty traffic profile, so given the traffic profiles, how much of an effect does bursty traffic have on the end to end delays? If the effect is negligible or within some acceptable percentage difference from results for a Poisson source, then bursty traffic could be ignored.

Figures 45, 46, and 47 show simulation runs of three different models for the three different traffic profiles. The three models are:

- polling discipline and bursty traffic present
- polling discipline and no bursty traffic present (95% confidence interval within 0.5%)
- no polling discipline and no bursty traffic present (95% confidence interval as above)

FIGURE 45. Corporate profile, plotting delay results from 3 different models

FIGURE 46. ISP profile, plotting delay results from 3 different models

FIGURE 47. Worst Case, plotting delay results from 3 different models

Looking at all the results from the three different profiles and three different models, it would appear that bursty traffic only has a significant impact when the system is almost fully utilized. Since the absence of bursty traffic would not appreciably alter capacity prediction conclusions, bursty traffic within the scope of this analysis does not have to be used. Perhaps the most significant finding from these results is how accurate the polling approximations are compared to the model with the polling discipline and no bursts present. The approximations are very good when the system is below 85% utilization. As system utilization increases beyond 85%, the polling approximation begins to lose accuracy. Therefore this opens the possibility to analytically solving the system, provided the AMVA solution for pre-emptive priority scheduling improves.

One possible way of avoiding the priority problem is to use the shadow method found in [18]. This could unfortunately polarize the model into software only. Reason is once the service times for the various software functions are inflated to account for higher priority activity, there is no need to represent the hardware in the model. However this is only true for this study, because the delay higher priority requests see is not of interest, and therefore can be eliminated.

7.9 Hypothetical Hardware Re-design Of The RLAN

Another use of the model is to explore new hardware and software design possibilities. The hardware re-design will focus on increasing the capacity of the RLAN system. Previous analysis in sections 7.2 and 7.7 showed that the bottleneck was the main bus. Also the Munich polling effect had a significant effect on the performance when no frames were being transmitted.

To increase performance of the system, something has to be done with the bottleneck device. It was decided to add two processors into the system, and partition the main bus into three buses. This is an attempt at minimizing contention on the main bus, and hence increasing the capacity of the system. Each CPU will handle all the responsibilities of pushing data from one side of the RLAN to the other. For example the LAN to WAN processor will do all the processing to ensure a packet goes from the LAN to WAN side. The Munich and LAN chips have been separated on their own bus, and each CPU has been isolated on its own separate bus with code each CPU executes. This reduces the contention by the CPU for memory down to just shared data accessing. Also the LAN and Munich

chip will not hinder the execution of code by the CPU to the same extent as in the old design. Main memory is not partitioned, and is shared among all the devices like in the old design.

There is no more polling in the system, and the Munich and LAN chips will send interrupts to the appropriate processor that a frame or packet has been received and ready to be serviced. The compression card CPUs are also partitioned, where one card focuses on decompression (traffic going from WAN to LAN), while the other CPU focuses on compression (traffic from LAN to WAN). This is also done to eliminate any contention for dual ported memory between the two CPUs. A diagram of the new hardware and software redesign of the system can be found in figure 48. The compression card is not included because it is not in the scope of the analysis.

FIGURE 48. Hypothetical hardware/software re-design of the RLAN

The modified layered model of the new hardware/software redesign is shown in figure 49.

FIGURE 49. Layered model of a hypothetical hardware re-design

With the new model, polling has been removed and interrupt overhead has to be accounted for. It is assumed that total CPU cost of a single interrupt is 80 μ s. This value was obtained from a context switch measurement on the iRMK RTOS kernel, which turned out to be 40 μ s. Since every interrupt has two context switches, a value of 80 μ s is used. The interrupt overhead is added into the service time for entries *RX_FM* and *LAN_RU* in figure 49. It was also assumed that for every operation the CPU performs, it will spend half its execution time in main memory.

To see if there is any improvement in capacity over the old design, the same two experiments performed in section 7.2 were run on the re-design. Capacity predictions of the new model is shown in figure 50 for data travelling from WAN to LAN, and figure 51 for data travelling from LAN to WAN.

FIGURE 50. Average waiting time experienced at the LAN queue

FIGURE 51. Average waiting time experienced in the WAN queue

There are several interesting points worth mentioning in these experiment runs. First is the overall average delay in the LAN and WAN queues has significantly dropped in the operating range of the system. This is because the system does not perform any polling. The bottleneck device changes when the traffic switches direction. For the LAN to WAN case, the bottleneck is the LAN to WAN CPU, but for the WAN to LAN case, the main memory is the bottleneck. In contrast with the experiments done in section 7.2, the bottleneck device in both cases was always the main memory and main bus combination.

The capacity of the re-designed system has been improved for the LAN to WAN case by about 30 to 40 channels, yet the capacity for the WAN to LAN case has not improved at all. This would indicate that the proposed bus partitioning does not effectively reduce the Munich polling effect.

7.10 Summary

This chapter applied the final step in the performance methodology presented in chapter 3. It demonstrated how the model can be conveniently used to explore an architectural redesign, by making modifications to the existing model. Capacity was the key performance question in all the experiments, to see how the system would behave under Poisson or bursty stream of packet arrivals, and under a new architecture. From these results, it was determined that bursty traffic does not have much of an effect for the scope of this analysis, and that the polling approximations described in chapter 5 worked very well.

Chapter 8

Discussion and Conclusions

8.1 Summary Of Results

The problem stated in chapter 1 to be addressed by this thesis was, to present a generic methodology to apply layered modelling to embedded systems. This was achieved by presenting an outline of the methodology, and then applying in the following manner:

- Reverse engineered a bridge/router architecture, and documented some major functionality in Use Case Map format.
- Derived a model of a real bridge/router architecture, which includes a hardware and software architecture. The model also portrays software that polls to a certain extent.
- Obtained measurements using a hardware/software monitor.
- Justified the model against real measurements.
- Performed capacity extrapolations using the model.
- Did capacity analysis using bursty traffic.
- Showed how easily the model can be used to do performance predictions on a architecture re-design of the system.

8.2 Conclusions

A conclusion drawn from this thesis is, the performance methodology presented in chapter 3 worked for the case study presented. This is shown by the thesis itself, walking

through and demonstrating various steps in the methodology and how to apply them. The performance methodology was based on the experience of doing the case study, and the methodology was derived from the experience. Therefore the performance methodology covers all the necessary issues when doing performance analysis. The methodology is a more simplified approach than [21] because the analyst does not have to understand the code to a great extent to produce a Use Case Map as opposed to an execution graph. The Use Case Maps can also serve a purpose of being supplemental design documentation, because of their intuitive and high level nature. This makes it easier to communicate with other engineers in understanding how the system works, and justifying the layered model later on.

It remains to be seen how easily other people can follow the performance methodology presented in this thesis. The greatest strength of the methodology is the choice of design notation and performance tools. Basic Use Case Map notation is a very intuitive high level design notation that any software oriented person can comprehend without problems. When progressing to the layered model of the system, the scenarios described in the Use Case Map do not mysteriously vanish into the model. This means the model can be visually checked for correctness, and parameters from the Use Case Map can be traced to the prediction results of the model. In essence the layered model paradigm is self documenting, by bringing together parameters, workloads, and scenarios into a understandable format.

From the various analysis done on the models of the RLAN, several conclusions can be drawn. For the scope of the performance analysis in this thesis, bursty traffic does not

have a serious effect on end to end delay of frames or packets going through the RLAN. It is not safe to say that bursty traffic will never make a difference, because it may be possible to create a bursty traffic profile where the difference will be significant. For now, one will always have to confirm by simulation to know whether or not bursty traffic can be substituted by a Poisson arrival stream.

An even more significant finding is that the polling approximations worked well within the scope of the performance analysis. This is a significant finding because there is a lot of polling literature available that contains complex mathematics and not very practical solution complexity. Just knowing the poll cycle time, and assuming it to be uniformly distributed worked quite well. With the polling approximations, and knowing when to ignore bursty traffic opens the door to the possibility of fast analytic solving of models like the RLAN. Not only that, but the modifications to the layered model solver would be minimal if any. For the bursty traffic, all that is really required is some mathematical criteria to know when simulation is necessary, or when it can be replaced with a Poisson stream.

8.3 Future Research

Throughout the period of doing research for this thesis, several issues surfaced that would warrant future study.

- The need to formalize various polling disciplines in layered modelling is important to have, as many systems do poll. This is more true in embedded systems, or systems where there is high interrupt latency.

- Expand analytic techniques for non-exponentially based arrival rates, like different types of bursty references. Poisson arrival rates for some systems may not be a realistic input to a model, and this tends to be true with any system which is connected to a LAN network.
- Bursty references can cause interesting things to happen in a system. One of the questions which arose out of the case study was, how much memory is required in the RLAN to minimize delay? Maybe this answer could be solved with simple bounds analysis, but it may also be better to extend the layered model somehow to include memory limitations.
- When studying embedded systems, caching at the hardware level becomes more important when trying to optimize hardware and software at the same time. It would be definitely worthwhile to investigate how to include this into layered modelling. With this in place, a performance analyst can make intelligent suggestions on what type of caching and hardware architecture would be most suitable to improve performance.
- Implement and find analytic techniques for message loss at entry queues. This type of queue discipline can be useful for studying bridge/routers, or any system that has to drop requests because it is overloaded.

References

1. Applied Microsystems Corporation, *CodeTEST Users Guide*, Applied Microsystems Corporation, February 1996.
2. Applied Microsystems Corporation, *CodeTEST Installation Guide*, Applied Microsystems Corporation, February 1996.
3. Applied Microsystems Corporation, *CodeTEST Probe for the Intel 80960Cx*, Applied Microsystems Corporation, February 1996.
4. Alan Burns and Andy Wellings, *Real-Time Systems And Their Programming Languages*, Addison-Wesley Publishing Company, 1989
5. R.J.A. Buhr and R.S. Casselman, *A Use Case Map Approach To High Level Design Of Object Oriented Systems*. Prentice Hall 1996.
6. Greg Franks, Alex Hubbard, Shikharesh Majumdar, John Neilson, Dorina Petriu, Jerome Rolia, Murray Woodside, "A Toolset For Performance Engineering And Software Design Of Client-Server Systems", *Performance Evaluation*, no. 24, 1995, pp. 117 - 136.
7. Hassan Gomaa, *Software Design Methods for Concurrent and Real-Time Systems*, Addison-Wesley Publishing Company, 1993.
8. Fred Halsall, *Data Communications, Computer Networks and Open Systems*, Four Edition, Addison-Wesley Publishing Company, 1995.
9. Curtis Hrischuk, "RAC version 3.4.3 RSC90E Measurements and Analysis", Gandalf Technical Note, 1997.
10. Intel Corporation, *iRMK 960 REAL-TIME Kernel Reference Manual*, Intel Corporation, Copyright 1989.
11. P. A. Jacobson and E. D. Lazowska, "Analyzing Queueing Networks with Simultaneous Resource Possession", *ACM Systems Modeling and Performance Evaluation*, Volume 25, no. 2, February 1982.
12. Raj Jain, *The Art Of Computer Systems Performance Analysis, Techniques For Experimental Design, Measurement, Simulation, and Modeling*, John Wiley & Sons Inc, 1991.

13. D. Katz, "PPP OSI Network Layer Control Protocol (OSINLCP)". RFC-1377, November 1992.
14. Chiung-San Lee and Tai-Ming Parng, "A Subsystem-Oriented Performance Analysis Methodology for Shared-Bus Multiprocessors", *IEEE Trans. Parallel and Distributed Systems*, vol 7, no. 7, July 1996, pp. 755 - 767.
15. Peter Maly, "CodeTEST setup with the RLAN and Compression Card", Gandalf Technical Note, 1997.
16. Peter Maly, "Layered Model of the RLAN system", Gandalf Technical Note, 1997.
17. Peter Maly, "Measurements performed on RLAN", Gandalf Technical Note, 1997.
18. Daniel A. Menasce, Virgilio A. F. Almeida, and Larry W. Dowdy, *Capacity Planning and Performance Modeling From Mainframes To Client-Server Systems*, Prentice Hall PTR, Englewood Cliffs, New Jersey, 1994.
19. D. M. Neuse and J. C. Browne, "Graphical Tools for Software System Performance Engineering", *Proc. Computer Measurement Group Conference XIV*, Washington D.C., Dec. 1983, pp. 353 - 355.
20. Kenneth D. Shere, Rachelle A. Carlson, "A Methodology for Design, Test, and Evaluation of Real-Time Systems", *IEEE Computer*, February 1994, pp. 35 - 48.
21. Connie Smith, *Performance Engineering of Software System*, Addison-Wesley Publishing Company Inc., August 1990.
22. John D. Spragins et al, *Telecommunications Protocols and Design*, Addison-Wesley Publishing Company Inc., 1991.
23. Hideaki Takagi, "Queueing Analysis Of Polling Models: An Update", *Stochastic Analysis Of Computer And Communication Systems*. North-Holland, New York, 1990, pp. 267 - 318.
24. M. K. Vernon, E. D. Lazowska, and J. Zahorjan, "An Accurate and Efficient Performance Analysis Technique for Multiprocessor Snooping Cache-Consistency Protocols", *Proc. 15th Ann. Symp. Computer Architecture*, Honolulu, Hawaii, June 1988, pp. 308 - 315.

25. Walter Willinger, "Traffic Modeling For High-Speed Networks: Theory Versus Practice", *Stochastic Networks: The IMA Volumes in Mathematics and its Applications*, Vol. 71, Springer-Verlag, New York, 1995, pp. 395 - 409.
26. C. M. Woodside, *Performance-Oriented Patterns in Software Design (A multi-level service approach) Draft Chapters 3, 4*, Carleton University, September 1996.
27. C. M. Woodside, J.E. Neilson, D. C. Petriu, and S. Majumdar. "The Stochastic Rendezvous Network Model for Performance of Synchronous Client-Server-like Distributed Software", *IEEE Trans. Computers*, vol 44, no. 1, January 1995, pp. 20 - 34.

IMAGE EVALUATION TEST TARGET (QA-3)

APPLIED IMAGE, Inc
1653 East Main Street
Rochester, NY 14609 USA
Phone: 716/482-0300
Fax: 716/288-5989

© 1993, Applied Image, Inc.. All Rights Reserved