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.

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.

Bell & Howell Information and Learning
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
800-521-0600

UMI[®]

Resource Functions for Model Based Performance Analysis of Distributed Software Systems

by

Stefan Bayarov

A thesis submitted to the Faculty of Graduate Studies

in partial fulfilment of

the requirements for the degree of

Master of Engineering

Ottawa-Carleton Institute for Electrical Engineering

Department of Systems and Computer Engineering

Faculty of Engineering

Carleton University

Ottawa, Ontario, Canada, K1S 5B6

June 08, 1999

© Copyright 1999, Stefan Bayarov

National Library
of Canada

Acquisitions and
Bibliographic Services

395 Wellington Street
Ottawa ON K1A 0N4
Canada

Bibliothèque nationale
du Canada

Acquisitions et
services bibliographiques

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.

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 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.

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.

0-612-48447-5

Canada

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

Resource Functions in the Model Based Performance Analysis of Distributed Software Systems

Submitted by Stefan Bayarov
in partial fulfilment of the requirements for
the degree of Master of Engineering

Chair, Department of Systems and Computer Engineering

Thesis Supervisor

Carleton University

June 08, 1999

Abstract

This work suggests ways to reduce the efforts of building software performance models by making their building a part of the design process. For the purpose a design tool, *ObjecTime Developer*, is used in the performance model building. *Developer* is a rapid design environment, which creates collaborative models based on communicating state machines and generates code for target platforms. This work introduces a test harness for automatic collection of resource demand data on the target platforms, and a tool for manipulation of the collected demand data. The work describes a measurement technique, based on application suite of models, which provides easy way for collecting resource demand data from the target platforms, and updating these data for new platforms. The demand measurements cover commonly used functions for systems constructed with *Developer*. The measurements are then fed into the model to support more accurate predictions made from the design environment. This work implements a new approach by grouping resource functions into resource environments; thus, making the practical use of resource functions much easier to implement and comprehensive to use in the process of performance modeling. A procedure for integration with the PAMB tool is implemented, and an example of automated model build and performance evaluation is included.

Table of Contents

1. INTRODUCTION	11
1.1. BACKGROUND	11
1.2. GOALS OF THE THESIS	13
1.3. OUTLINE OF THE THESIS.....	14
1.4. CONTRIBUTIONS OF THIS THESIS	14
2. PERFORMANCE ENGINEERING	15
2.1. APPROACHES TO PERFORMANCE ENGINEERING	15
2.2. PERFORMANCE ANALYSIS THROUGH PERFORMANCE MODELLING.....	16
2.2.1. <i>Approaches to Performance Model Development</i>	17
2.2.2. <i>Layered Queuing</i>	17
2.3. DEFINITIONS FOR RESOURCE DEMANDS	20
2.4. WORK ON SOFTWARE DEMAND CHARACTERIZATION	22
2.4.1. <i>Adequate Accuracy of Resource Functions</i>	23
2.4.2. <i>Complexity and Resource Functions</i>	24
2.5. THE ROOM METHODOLOGY	26
2.6. TEST HARNESS AND RESOURCE DEMAND DATA COLLECTION	28
2.7. ANGIO-TRACES	29
2.7.1. <i>Granularity of measurements</i>	31
2.7.2. <i>Parameter passing and angio-traces</i>	31
3. OVERALL DESIGN OF PAMB	33
3.1. PERFORMANCE TUNING WITH PAMB AND THE "WIDEBAND APPROACH"	34
3.2. BUILDING THE PERFORMANCE MODEL	36
3.3. RETRIEVING RESOURCE FUNCTIONS	39
4. RESOURCE FUNCTION ADDITIONS TO PAMB.	42
4.1. PRINCIPLES OF EXPERIMENTAL DESIGN	42

4.2. RESOURCE DATA AND ANGIO-TRACE GENERATION.....	43
4.3. PAMB PROCESS WORKFLOW.....	45
4.4. RESOURCE DEMANDS	46
4.5. RETRIEVING RESOURCE FUNCTIONS	50
4.6. CLASS HIERARCHY OF PAMB	51
4.7. RESOURCE FUNCTION MANAGEMENT MODULE.....	53
4.7.1. <i>Class Hierarchy of the Resource Function Module</i>	53
4.7.2. <i>Specifications of the Resource Function Module</i>	53
4.7.3. <i>Structure of the Resource Data Tables</i>	54
5. RESOURCE DEMAND DATA GATHERING AND MANAGEMENT	57
5.1. APPROACH TO MEASUREMENT OF DEMAND DATA.....	57
5.2. TEST HARNESS	60
5.3. CONSTRUCTING RESOURCE FUNCTIONS	65
5.3.1. <i>Resource Functions and User Code</i>	65
5.3.2. <i>Dimensions of Resource Functions</i>	66
5.4. RESOURCE DEMAND DATA COLLECTION FOR PAMB	69
5.4.1. <i>Components and Resource Demand Data</i>	71
5.4.2. <i>Synthetic Application Suite</i>	71
5.4.3. <i>Synthetic Application Parameters</i>	72
5.4.4. <i>Model Summary</i>	72
5.4.5. <i>Results of measurement of demand data</i>	73
5.5. RESOURCE DEMAND DATA MANAGEMENT.....	74
5.5.1. <i>Resource entries list-boxes</i>	75
5.5.2. <i>The Resource Environment window</i>	75
5.5.3. <i>The Resource Function window</i>	77
5.5.4. <i>The OK and Cancel buttons</i>	78
5.5.5. <i>The File menu</i>	78
5.5.6. <i>The Edit menu</i>	79

5.5.7. <i>The Options menu</i>	79
5.5.8. <i>The Help menu</i>	80
6. RESOURCE FUNCTION RETRIEVAL MECHANISM AND PAMB	81
6.1. INTRODUCTION	81
6.2. FORM AND FORMAT OF A RESOURCE FUNCTION	82
6.3. BUILDING A RESOURCE FUNCTION	82
6.4. SELECTING AND USING RESOURCE FUNCTIONS	83
6.4.1. <i>Selecting Resource Environment(s)</i>	84
6.4.2. <i>Selecting Resource Functions</i>	84
7. TUTORIAL EXAMPLE	85
7.1. OVERVIEW	85
7.2. THE CASE COMPONENTS	85
7.2.1. <i>The Server</i>	86
7.2.2. <i>The Client</i>	86
7.2.3. <i>The Scenario</i>	87
7.3. THE STEPS	88
7.4. THE EXPERIMENT RESULTS	91
7.4.1. <i>Single CPU system</i>	91
7.4.2. <i>Two CPU system</i>	93
7.5. IMPACT OF EXECUTION TIME ESTIMATES ON THE PERFORMANCE MODEL	94
8. CASE STUDY - THE GCS APPLICATION	96
8.1. OVERVIEW	96
8.2. THE SOFTWARE DESIGN	98
8.3. THE SCENARIOS	104
8.3.1. <i>Subscribing a client</i>	104
8.3.2. <i>Sending a request for a document</i>	105
8.3.3. <i>Saving a document</i>	105
8.4. IMPACT OF EXECUTION TIME ESTIMATES ON THE PERFORMANCE MODEL	105

8.4.1. <i>Client subscription scenario</i>	105
8.4.2. <i>Request to obtain a document</i>	107
8.4.3. <i>Request to save a document</i>	109
8.4.4. <i>Request to obtain a document, distributed system</i>	111
8.4.5. <i>The three scenarios combined</i>	113
8.5. IMPACT OF DESIGN CHANGES	114
8.6. REAL TARGET EXPERIMENTS.	115
8.6.1. <i>Model creation</i>	116
8.6.2. <i>Measurements taken</i>	116
9. CONCLUSIONS	117
9.1. SUCCESS AND CONTRIBUTIONS	117
9.2. LIMITATIONS	119

List of Figures

FIGURE 1. ALLOCATION OF SOFTWARE TASKS TO PROCESSORS.....	17
FIGURE 2. SAMPLE LQN MODEL.....	18
FIGURE 3. EXAMPLE OF A RESOURCE FUNCTION	20
FIGURE 4. STRUCTURE OF THE OBJECTIME TOOLSET	26
FIGURE 5. SOFTWARE LAYERS IN THE TARGET ENVIRONMENT	27
FIGURE 6. THE TEST HARNESS COMPONENT.....	28
FIGURE 7. THREE PHASES IN GATHERING AND USING RUN-TIME DATA	29
FIGURE 8. SAMPLE ANGIO-TRACE OUTPUT.....	31
FIGURE 9. HIGH-LEVEL PERFORMANCE ANALYSIS WITH PAMB	33
FIGURE 10. PERFORMANCE TUNING WITH THE WIDEBAND APPROACH.....	34
FIGURE 11. TRACEABILITY WITH PAMB AND THE WIDEBAND APPROACH.....	35
FIGURE 12. BUILDING PERFORMANCE MODEL WITH PAMB	36
FIGURE 13. EVALUATING PERFORMANCE MODEL'S PREDICTIONS.....	37
FIGURE 14. PAMB TOOL FRONT PANEL.....	38
FIGURE 15. RESOURCE TABLE SELECTION INITIALISATION.....	40
FIGURE 16. SELECTION OF RESOURCE FUNCTION TABLE.....	40
FIGURE 17. RESOURCE FUNCTION TABLE INITIALISATION.....	41
FIGURE 18. SELECTION OF A RESOURCE FUNCTION	41
FIGURE 19. VISUALIZING A 2^K FACTORIAL DESIGN	43
FIGURE 20. PAMB WORKFLOW PROCESS.....	45
FIGURE 21. SELECTING RESOURCE DEMAND DATABASE	46
FIGURE 22. MAPPING RESOURCE DEMAND DATA.....	47
FIGURE 23. SUMMARY OF MAPPED DEMAND DATA.....	48
FIGURE 24. RESOURCE ENVIRONMENT DATA WINDOW.....	49
FIGURE 25. OVERALL PAMB CLASS HIERARCHY	52

FIGURE 26. OVERALL REFCAM CLASS HIERARCHY	53
FIGURE 27. TEST HARNESS FRONT END	61
FIGURE 28. MODEL DATA SCREEN	62
FIGURE 29. PLATFORMS WINDOW	63
FIGURE 30. PROCESS OF RESOURCE DEMAND DATA COLLECTION	70
FIGURE 31. FRONT PANEL OF REFCAM.....	74
FIGURE 32. THE RESOURCE ENVIRONMENT WINDOW	76
FIGURE 33. THE RESOURCE FUNCTION WINDOW	77
FIGURE 34. SELECTION OF RESOURCE ENVIRONMENTS.....	83
FIGURE 35. THE CLIENT/SERVER STRUCTURE	85
FIGURE 36. THE SERVER BEHAVIOUR DIAGRAM	86
FIGURE 37. THE CLIENT BEHAVIOUR DIAGRAM	87
FIGURE 38. ANGIO-TRACE FILE FOR SINGLE SCENARIO	88
FIGURE 39. TOTAL WORKLOAD SPECIFICATION	89
FIGURE 40. LQN LOAD SPECIFICATION.....	90
FIGURE 41. THE EXPERIMENT RESULTS FOR NON-DISTRIBUTED SYSTEM.....	92
FIGURE 42. THE EXPERIMENT RESULTS FOR DISTRIBUTED SYSTEM	93
FIGURE 43. THE EXPERIMENT RESULTS FOR FAST-CPU SYSTEM.....	94
FIGURE 44. HIGH-LEVEL UCM VIEW OF THE GCS SYSTEM.....	96
FIGURE 45. GCS SYSTEM - STRUCTURE DIAGRAM.....	97
FIGURE 46. SERVER MANAGER ACTOR - DIAGRAMS.....	99
FIGURE 47. SERVER ACTOR - BEHAVIOUR DIAGRAM	100
FIGURE 48. GCS SYSTEM - DATABASE CLASS HIERARCHY	101
FIGURE 49. GCS SYSTEM - OVERALL CLASS HIERARCHY	102
FIGURE 50. RESULTS FOR GCS SYSTEM - CLIENT SUBSCRIPTION	105
FIGURE 51. RESULTS FOR GCS SYSTEM - REQUEST FOR DOCUMENT.....	107
FIGURE 52. RESULTS FOR GCS SYSTEM - SAVE DOCUMENT	109
FIGURE 53. RESULTS FOR DISTRIBUTED SYSTEM - REQUEST FOR DOCUMENT	111

FIGURE 54. GCS SYSTEM - NEW OVERALL CLASS HIERARCHY	114
FIGURE 55. GCS SYSTEM - REQUEST TO SAVE A DOCUMENT	115

1. Introduction

Performance is an essential attribute of real-time software, and often needs a long and expensive tuning process at the time of product integration and testing. This tuning process often occurs too late in the development cycle. Early evaluation of performance would be of great value to many projects. But early evaluation inevitably points to some kind of model, since the software itself does not yet exist.

Methods for designing new computer systems have changed rapidly in the last decade. Market pressures has been forcing companies to replace any *ad hoc* process in this area with a targeted and highly systematic process that focuses new designs on specific workloads. At the core of this process are models of the software system's performance and its workloads. Developing and verifying these models is the domain now called *performance analysis*.

In addition, object orientation (OO) has become a popular paradigm for the development of real-time systems. The adoption of this paradigm has been sped up by the vast increase in the performance, storage capacity and overall capabilities of today's hardware, the increasing complexity of real-time systems, and the increasing need for faster and faster turnaround of products to market. The design/development speed and maintenance advantage of OO real-time systems has caused many corporations to switch from more traditional software development paradigms. The OO paradigm has clear advantages over system developed through other methods.

To make things even more complicated, OO real-time systems often distribute the processing of an application across a number of platforms to improve performance and reliability, and to reduce costs. There is a reliability advantage with OO real-time systems because of the paradigm itself, which is very much closer to human thinking and understanding, a fact of importance when developing complex, real-time application software. Reliability has also been enhanced by an increase in expertise in this area over the last couple of decades. There is a cost advantage with OO real-time systems because of the wide range of existing tools and developed methodologies, available on the market. One such tool is ObjecTime's *Developer*, a CASE tool for design and development of OO real-time software. This tool works with ObjecTime's own methodology Real-time Object-Oriented Modelling (*ROOM*) [Selic94], and is compliant with the Unified Modelling Language (*UML*) [UML97] as well.

1.1. Background

We can consider a distributed software system as a set of sequential processes that alternately compute locally and interact among themselves through communication channels. From this viewpoint, there are a number of reasons why understanding and evaluating distributed software systems and predicting their performance is more difficult than it would be for sequential programs. The following main reasons are:

- **Size of the system** - distributed systems tend to be large. Instead of dealing with one sequential process, developers now have to deal with many processes simultaneously.
- **Multiple threads of control** - as a rule, distributed systems possess many threads of control. Therefore, the concept of *program state* needs to be extended to include all the local process states and the states of the communication channels.
- **Inherent non-determinism** - this is a very specific characteristic for distributed, asynchronous systems. It means that two executions of the system may produce different, but nevertheless valid orderings of events. Even when a software system is correct it still remains difficult to predict which of the possible executions will lead to correct results.
- **Non-homogeneous communication delays** - those kinds of delays are non-homogeneous and non-deterministic, due to the usage of store and forward mechanisms and channel contentions. Predicting communication times is thus impossible.
- **Error latency** - there is a lag between occurrence of an error and its discovery. Due to the significant communication delays and autonomous operations, this lag may be much larger in distributed systems than in sequential ones.

Because of this, software designers of real-time and distributed systems often have only the vaguest idea of the performance impact of certain key design decisions. Under such circumstances, designing for performance may be a question of trial and error where the intuition and experience of the design team play a key role. Although this design process differs from company to company, there are some common elements:

- ◆ an application analysis team, often working with customers, usually determines market requirements and reduces them to key workloads;
- ◆ lead architects work with interdisciplinary experts in examining practical concerns to bound the space of potential designs;
- ◆ the performance team, working with the lead architect(s), writes a model that measures performance in terms of throughput or total turnaround time;
- ◆ performance architects reduce the workloads to representative benchmarks and test cases;
- ◆ using a compiler and those benchmarks, performance architects test the model, usually employing trace-driven simulation;
- ◆ on the basis of these results, the design team chooses (and later refines) a specific system architecture;
- ◆ the team implements the architecture by translating the high-level design to lower-level

details and tuning.

The ObjecTime Developer is such a tool, which helps in the systematic design process. The PAMB tool ([Hriscuk95a], [Hriscuk95b]) has been developed to work with the ObjecTime Developer. The designs developed with Developer run on over 60 different platforms, and each has its resource data. However, the PAMB tool extracts model structure but not resource data. There is need, therefore, to develop a proper mechanism to introduce target resource data in the generated by PAMB performance models.

1.2. Goals of the Thesis

The primary goal of this thesis is to provide an automated environment for capturing resource demand data for many platforms and feeding them into performance model. The environment was created for demands made by the ObjecTime's *microRTS* [Selic94] software primitives. Such an environment is the basis for improving OO real-time system performance modeling through the use of resource functions. This work also aims to create an environment to resolve parameter dependencies in resource functions and to store those functions for a fast and easy access. In connection with this, the thesis intends to develop an interface to insert resource functions in the performance models. A policy for resource demand data collection / manipulation is proposed. Using a set of real-time targets, resource demand data for those targets is collected. The corresponding resource functions for those targets are stored and used in the performance modeling of different architectures and workloads. Furthermore, the possible relative performance improvement with each policy is discussed. The following questions regarding resource functions and OO real-time performance modeling motivate this thesis:

- ◆ What is the effect of usage of resource functions in object-oriented real-time system performance models?
- ◆ How do resource functions fit in performance models?
- ◆ Is the use of resource functions more beneficial than other methods?
- ◆ When using resource functions, what are the effects of system architecture and workload characteristics on the performance modeling?
- ◆ What characteristics of resource functions are most useful in performance models for high system performance?

The goal is to demonstrate that performance questions *can* be addressed together with concurrency architecture, even at a very early stage, and that they do not require a lot of training or time to use them. Part of the goal of this thesis, therefore, is the development and validation of a real design, that is to produce a prototype and verify it against a real target platform.

1.3. Outline of the Thesis

This thesis is divided into ten chapters. An introduction to the problem domain of this work, with goals and motivations, is presented in Chapter 1. Some of the main issues of performance engineering and general background of the domain are listed and discussed in Chapter 2. Chapter 3 discusses the PAMB tool and its overall design, as inherited at the beginning of working on this project. The new concepts of resource function management, the overall perspective of the solution proposed by the author are given in Chapter 4. The problems in gathering resource demand data, description of the newly developed automated tool for collecting such data for the purpose of building resource functions, as well as presentation of the ReFCAM tool for managing the collected demand data are given in Chapter 5. Chapter 6 provides a description of the resource function mechanism, as outlined in this work and integrated by the author in the PAMB tool. Chapter 7 presents a very simple client/server ObjecTime model as a tutorial example to describe the concepts and ideas of this work. In chapter 8, the author presents a group communication subscription server system (GCS) as case study in this work. Chapter 9 summarizes the conclusions and the contributions made by this thesis and recommendations for directions for future research. Appendix A contains excerpt of the case study system's output. Lastly, the reference section presents a list of the used references in this work.

1.4. Contributions of this Thesis

This thesis addresses the problem of how to manage and manipulate resource functions with $1..n$ independent parameters. In particular, the contributions of this thesis are:

- A new resource function management utility for the purpose of storage and manipulation of resource demand data;
- A test harness to automate the collection of resource demand data and store the data for later use;
- The creation of a set of ObjecTime executable models, which run on the target platform(s) and provide the means to measure the resource demands of the ObjecTime's microRTS primitives on this target, then to output those demands for storage and retrieval;
- A new approach to handling and construction of resource functions. Previously, there were multi-dimensional resource functions and different ways the literature offered to construct them. This work introduces a way of reducing dimensions of resource functions by flattening them into single-parameter resource environments, and a way of searching for those resource environments, thus speeding up several-fold search and retrieval of resource functions in the process of performance model building.
- Sample C++ big model, running on target, to demonstrate the validity of the proposed concepts and illustrate the use of the PAMB tool.

2. Performance Engineering

This chapter gives an overall review of performance engineering in general and the need for performance engineering.

Performance refers to the response time or throughput as seen by the end users of the software system.

Software performance engineering is used to enhance performance or address performance concerns throughout the lifetime of a computer system. *Software Performance Engineering (SPE)* can be defined [Smith90] as a set of methods for constructing software systems in such a way that they meet some defined performance objectives. The process starts early in the software development lifecycle and uses quantitative methods to identify a satisfactory design by eliminating those with unacceptable performance. Having this step in the early design stages avoids (hopefully) developers investing significant time in the implementation of the software system. The process of SPE includes performance data collection, quantitative analysis techniques, prediction strategies, performance data presentation and tracking, model verification and validation, performance design principles, and so forth.

The performance of computer software in general is influenced by many factors at various levels of granularity. For example, increasing the size of a memory cache on a processor or enabling a certain type of compiler optimisation can result in significant performance changes. This sensitivity to detail is exacerbated by the fact that many critical performance-related factors are unknown at the time the design is being finalised. This imposes the need for SPE even more.

2.1. Approaches to Performance Engineering

An SPE performance test may use a combination of experience, performance measurements, or performance models. But, the utility of these approaches depends on the architecture of the computer system being analysed. The most difficult type of computer system to analyse has several distributed applications that execute on a heterogeneous, geographically distributed computer system.

A first cut performance test may use experience, intuition or trend extrapolation [Smith90]. These approaches can be useful if something similar to the proposed system already exists and an expert is available. By expert here the author means a person with rich knowledge of such a system. This method, however, is difficult to apply to large systems or heterogeneous environments: there are too many details to consider. This approach is best suited to the analysis of a centralised software system.

Another performance test approach is to measure performance metrics during an experiment or during normal use of the system [Smith90]. The measurements are considered to be trustworthy since they

are taken from the final system. There are two major problems with measurement: 1.) Measurement is not feasible in the design and development stages of the system because the system is not measurable if it is not operational. 2.) Measurement of most systems is a complex activity, which involves considerable human and machine costs. How do we set up the measurement experiments? Do we use a "live" workload, not knowing whether that workload will be representative of the typical operating conditions? or do we use a controlled workload [Jain91], e.g., a *benchmark* or *synthetic workload*, which we assume represents typical usage?

A measurement provides good data about the configuration that is measured, but it does not say much about other configurations. Generality is hard to obtain. There may be dozens of related configurations that are impractical to measure but also important. The measurement approach may be impractical due to system size, expense, or lack of availability. In particular, metrics of a distributed application are difficult to measure because the control and data-flow are spread amongst a number (potentially high) number of computing nodes and software components in different locations.

The last approach to performance testing consists of constructing a model, which emphasises the key performance aspects of the analysed system. This model later can be evaluated to provide performance predictions about the system. A performance model characterises the system by combining structural information with behaviour descriptions of the key workload elements (in terms of resource usage). Performance predictions are developed using this information and these predictions can be analysed for problems. Performance models are flexible, providing performance predictions for several system configurations or different workloads. The problem with this approach is the cost to build such performance models.

2.2. Performance Analysis through Performance Modelling

Providing early performance estimates in the design of new software helps the designers to make appropriate decisions, and to revise them where necessary in the early stages of development. The two main approaches to combining software design and performance engineering are *empirical* and *predictive*. The fully empirical approach is to implement the software system in some form, measure its performance, and then modify the software as necessary. The predictive approach, as described by Smith [Smith90], starts by deriving a model of software execution patterns from the design. As a next step, a second model is constructed to provide for performance predictions, after which the predictions are used to guide the modification of the design.

The predictive approach uses separate *performance models*, which are constructed for each software system. Such a model characterizes the system by combining structural information with behavior descriptions of the key workload elements (in terms of resource usage). Using this information, performance predictions are developed and analyzed for potential performance

problems: thus, the approach provides the means for faster evaluation and experiment.

2.2.1. Approaches to Performance Model Development

Software performance engineering and performance model development are closely linked. Usually an SPE methodology includes at least one model development technique because performance models are heavily used during performance analysis. Models can be categorised [Jain91] by their solution technique which may be *simulation* or *analytical*. But, this division is misleading because the performance predictions of an analytical model can be produced by simulation. Another classification is to consider a performance model to be *executable* or *abstract*. An *executable performance model* includes algorithms, which closely mimic the actual implementation's execution and this implementation is emulated to produce performance predictions. An *abstract performance model* concentrates on the major features of an implementation, which are sufficient for a mathematical analysis to predict performance. The performance model techniques discussed here are:

- The simplistic executable model development, which constructs an executable model that mimics an application's execution behaviour;
- The evolutionary executable model development, which occurs when an early executable design is used as an early performance model and the executable design evolves into an implementation;
- The investigative abstract model development, which constructs an abstract model from the analyst's investigation;
- The hybrid abstract model development, which uses executable information to supply parameters for an abstract model;
- The transformational abstract model development, which constructs an abstract model using a series of formal transformations.

Performance models of software designs can give early warnings of problems like excessive delays, but are seldom used mainly because of the considerable effort needed to construct them.

2.2.2. Layered Queuing

Large, distributed software systems may involve several layers of servers. A client may send a request to a server, which in turn may require the services of a lower level server in order to satisfy that request. The middle level server blocks until the lower level server completes its work; and the client blocks, waiting for a reply from the middle level server. This type of interaction, where a process makes a request for service and blocks until it receives reply, is known as a *rendezvous*. Rendezvous are

prevalent throughout client/server systems but conventional queuing models (originally designed to model hardware convention) cannot properly estimate waiting times due to software server contention. The Layered Queuing Network, or LQN ([Woodside95] and [Rolia95]) was developed as means of properly modelling client/server systems with rendezvous interactions.

In a layered queuing model, both the software and hardware resources of a system are modelled as **tasks**. A task is an object, which has single thread of control (no internal concurrency) and which can initiate or accept service requests. Tasks can send requests to each other. A task, which only sends requests but never receives any, is called a *pure client*; a task, which both sends and receives requests, is referred to as an *active server*, while a task, which only receives requests is called a *pure server* (see Figure 1). Tasks can offer more than one service. The performance characteristics of a service, which task offers, may also vary depending on the client making the request.

Figure 1 - LQN - Allocation of software tasks to processors

An **entry** is a subdivision of a task corresponding to a particular service. Messages are addressed to entries, and each entry has its own execution parameters. Task entries are used to distinguish between the various services offered and between the various performance characteristics of these services. The entries retrieve requests from a common FIFO queue. So, a task with entries can be reduced to a single server with multiple classes of customers. A task can also be turned into a multi-server by setting its multiplicity to greater than 1.

The execution of each task entry is divided into **phases**, i.e. each entry has a specific number of phases of execution before it completes. When a task sends a request to the entry of another task, the calling task is blocked until the receiving task retrieves the message off the queue and the appropriate entry completes its first phase of execution. Any subsequent phases of execution continue in parallel with the calling task. A call can be made completely asynchronous by having no execution time in the

first phase of the entry. Using first phase allows the model to represent rendezvous interactions in client/server systems.

Lastly, each task in a model is assigned to a processor. A processor *task* is automatically generated for each processor used. The execution time for each entry of each task in each phase is divided into slices, or quanta. An entry in the processor task is created to represent the quantum of execution time assigned to each phase of execution of each entry running on that processor. Therefore, the execution time of an entry in a particular phase is therefore translated into a number of calls to the corresponding quantum entry in the local processor task. This way hardware contention can be modelled.

Formal Parameters

The parameters labelled on the graph of tasks and their interactions (such as Figure 1) are:

- T_i , on reference tasks (the rectangles). T_i is the mean delay time per cycle.
- s_{ip} , on server task i , in the form $[s_{i1}, s_{i2}]$. s_{ip} is the mean total execution time in phase p , for $p = 1, 2$.
- y_{ip} on the arc from task i to j , in the form (y_{ij2}) if i is a reference task or (y_{ij1}, y_{ij2}) if i is a server task. y_{ip} is the mean number of requests by rendezvous from task i to task j , during phase p . For a reference task there is only y_{ij2} , since it only has a second phase.

The processor service times and processor request rates are not shown; they are derived from the execution time of the tasks, and the number of requests they make. In the mathematical analysis of these networks it is assumed that the intervals of execution time between requests are random, independent and exponentially distributed; and that the requests per phase are random, and geometrically distributed. This is meant as an abstraction for data-dependant execution and request sequences.

Figure 1 shows a system with two client tasks and two levels of servers. There is no strict layering of services in this example. Application 1 consists of tasks $T1$ and $T3$, where $T3$ is server to $T1$. Application 2 has a single task $T2$. Both applications use server tasks $T4$ and $T5$ (server A and B), and server A sometimes itself uses server B. Each task is represented by a parallelogram, which may be divided internally into separate entries. Inside each entry in square brackets is a list of average phase execution times, and attached to each arc in round brackets is a list of mean numbers of messages sent, in each phase of the sending task. Some of the attached processors are shown in ovals.

Figure 2 - A sample LQN model

The components of an LQN model (a sample LQN model is shown on Figure 2) are divided into the following categories for discussion purposes: the *scenario structure*, the *task behaviour*, and *environment configuration parameters*. The scenario structure of an LQN model considers a task as an opaque object that interacts with other tasks. The task behaviour component is a set of directed sub-graphs of activities, which describe the internal, probabilistic execution of a task. The LQN environment configuration parameters describe key aspects of the target execution environment. The configuration information is usually the first item to be varied when performing *what-if* performance tests.

2.3. Definitions for Resource Demands

Here we provide some definitions to clarify what is meant by resources, resource demands, software components and resource functions. It is a general introduction to the terminology used throughout this document.

Resource

A resource is any entity in the system, whether software or hardware, which offers a service. The most common resource referred to, and the one, which this thesis focuses on, is the CPU. Other resources are disks, memory, semaphores, file servers, or application-level servers.

Resource demand

Resource demand quantifies how much a resource is used to perform an operation. It could refer to the number of service requests made to a specific resource or to the amount of execution time the resource must spend servicing the request. Resource demand does not refer to the amount of time spent waiting to use a resource. This thesis concentrates on CPU time as the resource demand of interest. But the principles suggested here are generic enough, so they can be applied to other types of resource demands, such as the number of calls to a server or the number of I/O operations performed on a disk/memory chip, and so forth.

Software component

A software component can be any software entity (function, method, process, or system), that performs a logical function. A software component is, therefore a resource, too.

Resource Function

Typically, the resource demand of a software component will depend on a number of parameters. For example, the CPU time required to send a message between two threads would depend obviously on the size of the message being sent. The purpose of a resource function, therefore, is to capture how the resource demand of a software component changes with varying values of the independent parameters. A mathematical function is used as a compact representation of a resource function.

Figure 3 presents an example of a resource function for the *send* software component in the ObjecTime's *microRTS* (see section 2.5) for one of the target platforms. The dots on the diagram represent the measurements obtained as a result of some conducted experiments. The curve represents the mathematical function used as a compact representation of the resource function.

Figure 3 - Example of a resource function

This curve is fitted to the measured points to best represent the relation between the measured parameters.

The main point then is not how the resource function is represented, but how accurately it estimates the resource demand for a given set of parameter values. Section 2.4.1 discusses in more detail the accuracy of resource functions.

2.4. Work on Software Demand Characterization

Although the term *resource function* is not in general use, the concept of characterising the resource demand of a software component at various parameter levels can be found in a number of published articles. One of the earliest references is [Booth80], where a *cost metric* is used to describe the number of CPU cycles consumed by an operation in an abstract data type as the input to that operation changes. A study by Xu and Hwang [Xu96] models the communication overhead of a message passing on an IBM SP2, under different combinations of message size and number of nodes involved in a group communication. Vetland [Vetland93] refer to resource functions as *complexity functions* and use them to capture CPU, disk, and network demand for the components in a variety of distributed applications. Vetland and Woodside [Vetland96] capture the resource function for XDR marshalling overhead in remote procedure calls, varying the length of opaque data. A practical use of these resource functions is found in Menasce's CLISSPE performance modelling scripting language [Menasce97], which expects as input, linear models relating the CPU demand to the number of data bytes involved in RPC and I/O calls.

Booth and Wiecek [Booth80] attempt to expand the traditional abstract data type (ADT) to include performance information. The resulting *performance abstract data type* contained *cost metrics* for each operation of the ADT, which described how the CPU demand of an operation was related to the inputs for that operation. Constructing these functions require in-depth analysis of the control and data flow within an operation. Each operation is divided into primitive sub-operations, and the average number of times a primitive sub-operation is called is determined by examining the size of the data structures involved and the probability of transition from one primitive sub-operation to the next. The authors suggested that a library of performance ADT-s could then be developed and used to evaluate the performance characteristics of alternative system designs - where each system is built on top of the existing ADT-s. Although the notion of a library of resource functions is useful, the proposed construction technique for the cost metrics requires a prohibitive amount of work, not to mention the availability of source code.

Vetland [Vetland93] suggests a more practical approach to building resource functions, which involves measuring the component's resource demand at various levels of its independent parameters and fitting the measurements with regression. Source code is used only as a means of determining the mathematical form of the regression fit (e.g. linear, polynomial, log, etc.). Vetland writes:

"... the form of each complexity function should be determined by code walk-through or possibly by inspection of the design documents for the software components. Thus, white-box measurement experiments are considered necessary." (p. 204)

When source code is not available, the modeller must resort to black-box testing, in which case Vetland suggests a manual process of "iterated assumption and testing against the data" (p. 203) - i.e. taking measurements, guessing at the proper form of the function, modelling the data with that form. In case the fit is not good enough change the form and possibly collect more data - a process, which Vetland concedes is "very difficult and time-consuming to undertake". What is needed then is an automated process, which can iteratively progress towards an accurate enough form for the resource function.

Current research, therefore, has not provided a practical means for constructing resource functions. Little attention has been given to what type of experimental design should be used during blackbox testing, when the shape of the underlying, *true* function is unknown. In addition, there remains the problem of how to model the data once it is collected. Menasce's modelling language, for instance, will only accept linear resource functions, regardless of how well those models fit the actual data.

2.4.1. Adequate Accuracy of Resource Functions

The accuracy of a resource function is defined as the estimated percentage difference between the measured demand data for some value(s) of the independent parameter(s) and the calculated by this resource function demand for the same value(s) of the independent parameter(s).

The accuracy of the performance model then will depend on the execution time estimates provided for the phases of each entry (see Figure 1), where resource functions are used to provide those estimates.

In the process of measuring and collecting demand data for the ObjecTime's *microRTS*, one issue came up: How many levels of each parameter are needed - that is how many points should the resource demands be taken at?

If a resource function is non-linear or discontinuous, more points will be required to describe it than if it had been a single, straight line. On the other hand, it is very easy to get into measurement overkill. If we consider the resource function on Figure 3 for a *send* operation, a full factorial experiment would involve, for example, varying the message size from 64 B to say 6400 B in steps of 64 [B]. The collection of so many points can be costly and is probably not necessary for providing accurate estimates of demand.

Here an issue can be the fact that a resource function is simply not predictable, i.e. it can be non-deterministic function. There could be a number of reasons for such behaviour and not all of them necessarily known - like caching, etc. In such case collecting more and more points does not help

determine the stochastic behaviour of the resource. No additional measurements would improve the accuracy of such a resource function.

One not very practical way then to build a resource function for a specific performance model, requires that the modeller engage in an iterative process. The resource functions would have to be constructed initially to conform to an arbitrary, reasonable accuracy. The performance model then would be created, using the so built resource functions. Based on the model behaviour, the percentage error of the model's responses then can be calculated. The modeller then must decide if the original resource functions are "good enough" or if the accuracy requirements need to be tightened. If greater accuracy is required then the resource functions must be upgraded to satisfy the new requirements.

However, as a rule of thumb we can assume that the performance of a system depends on one or two key parameters, like system workload, CPU cycles, etc., i.e., the bottleneck(s) in a system control its performance. Then a more practical way would be to perform measurements at the bottlenecks of the system, which would give us some picture about the performance of the whole system.

The other issue in providing adequate accuracy of a resource function is - having collected the demand data (the experiment measurements), how to model them?

Most of the time, a simple linear regression does not provide an adequate enough fit. For the purposes of this work the author started by using truncated power splines [Press88] for interpolation of the resource demand results into resource functions.

In the case of pre-packaged resource functions, the resource function builder cannot know in advance the accuracy requirements of every user of that function. Therefore, an arbitrary, reasonable limit on the function's accuracy would have to be imposed. Expert opinion could be relied on to provide some guidelines.

From the above described two issues one conclusion can be made: The modelled resource function, $f_m(X)$ (where X is the vector of independent parameters), gives only an approximate value for the corresponding software component's resource demand. The reasonable accuracy of this value contributes to a reasonable overall accuracy of the performance model.

For software performance models, for example, overall model errors of 20% may be tolerable. In such a case, a resource function accuracy of 10% and 20% would be a "reasonable" value.

2.4.2. Complexity and Resource Functions

In this work we are interested in the effort in resource consumption for executing an activity (i.e., a software operation). This might be a single atomic operation (on level OS-kernel) or an algorithm, or

might contain one or more algorithms. Resource consumption means CPU, file accesses, or any other service needed by this operation.

Parameters of the activity are values that affect its resource consumption and can be any number q independent parameters. For example, for an activity, which sends a message, the size of the message could be a parameter. If the activity has no parameters, it is always the same in resource consumption, or it varies in ways that we cannot predict.

If the activity is an algorithm, and has parameters, then its resource demands may be described by the complexity of the algorithm. This is called a **complexity function** for the algorithm, and is the subject of a huge body of theory. Usually the theoretical results are asymptotic only... they do not give a prediction for useful finite values of the parameters, but they show how the resources will vary with the parameter, as it becomes very large.

Useful values have to be investigated by experiment rather than theory, and it is the empirical values that take the name resource functions. If there are parameters, then the resource function is a function that describes how the resource demands vary with the parameters. It is not itself an algorithm whose complexity we want. When considering resource functions, there are some general mathematical problems to consider:

- How do we measure the complexity of a resource function, which in turn will help with the computability of this resource function?
- Is the algorithm A_1 better than the algorithm A_2 (for computing the same resource function) in the sense that it uses fewer resources, such as time or memory space?
- How do we build a more or less universal algorithm to calculate resource functions, which can have a wide variety of independent parameters?
- What is the impact of the number of those parameters on the computability of a resource function?

Complexity considerations are of crucial importance in the area of resource functions. However, the author found that most of the researched literature does not cover the resource functions considered in this work (see paragraph 4). For example, the works of [Ogihara96] and [Borwein87] cover *sort* and *search* algorithms, which however cannot be related to the considered here algorithms, implemented by the ObjecTime's microRTS (see next section 2.5).

The domain of complexity and computability of resource functions is a vast and complex one. Despite this fact, in the literature the work on complexity of resource functions is very scarce. This is

understandable, since resource functions in their nature are experimentally obtained and any theoretical generalization about them is difficult, if at all possible. In order to reduce the complexity of this domain, the author has taken some steps to simplify it for the purposes of this work. Details about this can be seen in chapter 5.3 of this work.

2.5. The ROOM methodology

The ROOM (Real-Time Object-Oriented Modeling) methodology and modeling language [Selic94] was devised to model real-time systems, which can be grouped by the following characteristics:

- *Timeliness* - a requirement to perform its function "on time", whatever that happens to mean in a particular context;
- *Dynamic internal structure* - the system components, which deal with particular aspects of the environment, must be dynamically reconfigurable to match the dynamic of external environments.
- *Reactivity* - responding to different events whose order and time of occurrence are not always predictable;
- *Concurrency* - at any given moment multiple simultaneous activities can be taking place in the system;
- *Distribution* - multiple computing sites cooperatively achieve some common function;

The ROOM language is a graphical modeling language that permits the building of fully executable models. This is achieved through the ObjecTime *Developer* tool, which is used to create collaborative models based on communicating hierarchical state machines.

Figure 4 - Structure of the ObjecTime Toolset

The models created by the Developer tool represent both the structure and behaviour of the system they model. The ROOM methodology distinguishes between two main types of classes - passive (*data*) classes and active (*actor*) classes. The actors as active elements and their inter-relations determine the overall behaviour of the modeled system. The internal operation of an actor over time defines its *behaviour*. The high-level behaviour of an actor is represented by an extended state machine i.e., at any point in time an actor has a state that determines how it will

react when it receives stimuli (a message). An example of actor behaviour diagram can be seen in chapter 7, on Figure 36. The ROOM language allows as well the modeling of the components in a system and the interactions between them, i.e. the **structure** of a system. In ROOM every actor has a structure diagram, in addition to its behaviour diagram. An example of actor structure diagram can be seen on Figure 35, in chapter 7 of this work.

The ObjecTime Developer toolset implements the ROOM language. Figure 4 shows an overall structure of the ObjecTime toolset.

A part of the toolset is the ObjecTime ROOM virtual machine. The ROOM virtual machine is a hypothetical hardware device capable of directly executing ROOM model specifications. The *ObjecTime's* run-time system provide an implementation of the ROOM virtual machine, and are responsible for providing the necessary mechanisms that support the ROOM paradigm as well as the services needed by ROOM models. In the process of developing ways of collecting and storing the resource demand data, one point became very clear – the need for easy storing/retrieving and managing the collected resource data.

The ObjecTime toolset includes a target run-time system (i.e., TargetRTS) [Selic94], which is linked with the application code to provide a stand-alone executable that may be run on either a workstation, or on a target environment with an underlying real-time OS (see Figure 5).

Figure 5 - Software Layers in the Target Environment

The ObjecTime's *microRTS* kernel runs on a high number of platforms (currently about 60). The amount and diversity of data was such that it was very easy to loose any grasp of what data and where these data are available/located.

Thus came up the idea about some test manager utility, which would manage such amount of data. This enforced the creation of the *Resource Function Data Management Utility (ReFCAM)* tool, which was to help with the overall management and storage of the collected resource demand information. For more information about the *ReFCAM* tool, its architecture and overall design please see chapter 5 – “*Resource Demand Data Gathering and Management*”.

2.6. Test Harness and resource demand data collection

As mentioned earlier, currently *ObjecTime's microRTS* kernel runs on a relatively high number of target platforms (about 60 at the time of writing this paper). The original approach to collecting resource data for each platform was to use native performance measurement tools for each target platform. This approach would have had the advantage of being more precise about the collected data. However, it became very clear that this approach has one big drawback - the tests have to be instrumented for each platform accordingly. Such an approach would have required a lot of time and human workdays to implement.

Given the high number of platforms, alternative approach was taken by the author. A set of *ObjecTime* models was designed and developed to run on each platform. Each model would perform a series of measurements on the resource demand of *microRTS* software primitives and report them in a common format. The obtained in such a way resource demand data will be stored for later use by *ReFCAM* and *PAMB* tools.

Figure 6 - The test harness component

Figure 6 gives an overall picture of how the test harness fits with the rest of the components involved here. The test harness provides a graphic interface (1) to the analyst/designer, and manages the synthetic application suite (2) - it loads it onto the required platforms (3); for each synthetic application run on each platform corresponding log-files with resource demand measurements are generated.

Chapter 6.2, "*Resource Data Collection*" explains in more details the overall structure and functionality of the test harness and the set of *ObjecTime* synthetic applications.

In order to obtain the resource demand for all the *microRTS* software components it was necessary to have them "perform" alive. Another words, there was need for an ObjecTime model, which would exercise all of the *microRTS* software components and provide the means to measure their resource demand data and output these performance measurements in readable format for later use. This raised the question of gathering, storing and using run-time information. The author had to look into some general principles in this area and come up with a solution to this problem.

Figure 7 - Three phases in gathering and using run-time data

The whole process of gathering and using run-time information can be split up in three relatively independent phases (see Figure 7):

- Generation of run-time information. This requires observing the system and gathering the required data;
- Storing the data and transporting the data to location where it is used;
- Interpretation of the collected data and its usage for the purpose it was gathered.

This way of looking at the process of gathering and using runtime information gives the advantage of considering many of the encountered problems separately.

The problems and the way those were resolved in this work are described in chapter 5, *Resource Demand Data Gathering and Management*.

2.7. Angio-Traces

The concept of angio-traces [Hrischuk99] and their implementation in ObjecTime's product is vital part of this work. Angio-traces are produced by prototype(s) of the system under design. Typical end-to-end operations are executed on the prototype and are recorded in a tracing format, referred here as angio-trace and which maintains casual relationships. The angio event fields are for performance modeling purposes and designed to be portable across environments. Each angio event records:

- The dye that was active when the event was recorded;
- An identifier for the prototyping language statement that was executed;

- A task identifier for the concurrent object that executed the statement;
- Event type information for constructing the performance model;
- Event parameters for resource demand calculations or trace management.

The dye is formed from three parts: a trace name and an occurrence number. The first two parts are specified by the analyst and define the scenario. The prototype at runtime automatically assigns the occurrence number to distinguish between multiple dyes that are executing concurrently with the same trace name. The generation of traces has two important characteristics:

1. The dye crosses object boundaries during interactions and persists across context switches during execution.
2. A naming convention is used for trace names to allow post-execution causal analysis. Concurrent threads of execution that cooperate on the same scenario are related by name.

The angio-tracing mechanism currently is implemented in the *ObjecTime Developer* tool as part of its *Simulation RTS* mechanism - see Figure 8 for a short sample output of an angio-trace from *ObjecTime* model run. It is a scaled-down version of the original concept and does not provide a full range angio-tracing of all primitives in the *ObjecTime*'s run-time kernel.

The current *ObjecTime* implementation of angio-tracing belongs to the first generation angio-tracing. This is the reason why the current implementation cannot provide many features, available only in the next generation angio-tracing. Features of the third generation angio-tracing include:

- It is highly portable because it makes very few assumptions about the environment. It can be implemented in the micro-RTS, the RTS simulator, or other target environments (e.g., host RTOS library);
- It is not dependent on wall-clock time so it is network transparent. This allows it to work on an uni-processor target system, a multi-processor target system, or with several *ObjecTime* designs that are connected over a network (e.g. Internet). This could provide the ability to have a target system and an RTS simulation interact;
- It has formal properties that can be the basis for unique, new features to differentiate your product. Currently the Message Sequence Chart (MSC-s) diagrams are generated from the angio traces. Other capabilities, which exist (in prototype form) are: the automatic generation of performance models and 3D visualization of software execution;
- The formal properties allow optimization of the tracing format to minimize memory consumption and CPU overhead. This is crucial for an effective angio-trace implementation;

```

thread srvrldye.1 {
    2 Client:client1(2-1) state Ready;
    2 Client:client1(2-1) receive %done;
    2 Client:client1(2-1) send %releaseClient delay 1;
    2 Client:client1(2-1) spawn srvrldye.1.1;
    2 Client:client1(2-1) transition done action 'send interthread RTInteger';
    2 Client:client1(2-1) state Done;
};

thread srvrldye.1.1 {
    3 Server:server(3-1) state Ready;
    3 Server:server(3-1) receive %releaseClient;
};

```

Figure 8 – Sample angio-trace output

Currently there is on going design effort to bring to ObjecTime *Developer* the latest from the angio-trace theory and implement angio-traces as part of the *microRTS* kernel. This radical change of the implementation of angio-tracing in the ObjecTime's product raised some questions and concerns, which are presented here.

2.7.1. Granularity of measurements

One of the questions to arise during this work was about the granularity of the collected measurements on the target RTS: To what *depth* of granularity of the resource demand data to go? The software primitives provided by the ObjecTime's *microRTS* are built using more granular primitives (native OS API functions) from the underlying real-time OS. However, it will be very difficult to provide mechanism to angio-trace those API calls; as well as such very high-level granularity will increase the complexity and size of the graphs, built during the performance modelling. Thus, it was decided to perform angio-tracing and resource function data collection on the level of the *microRTS* kernel primitives.

Here it has to be mentioned that a concern existed even with the decision to perform resource demand data collection on the level of the *microRTS* kernel. However, some preliminary tests with very large software models (thousands of nodes) have been done with success.

2.7.2. Parameter passing and angio-traces

During the process of refining the process of using resource functions, one specific problem was encountered. How to reflect into the angio-trace file the parameters passed to a class method? Another

words, if we do *send(message)*, how to ensure that the size of the message will be passed to the angio-trace mechanism and reflected in the angio file, so it can be used by the PAMB tool? The currently implemented version of angio-traces in ObjecTime Developer does not support parameter passing.

Again, resolving this problem is considered being outside of the scope of this work and will not be discussed in many details in here. However, for the purposes of this work some solution (even temporarily) was needed. One solution, which came in the process of designing, was to use a workaround provided by the tool itself.

For a quick prototyping ObjecTime Developer implements the so-called RPL (*Rapid Prototyping Language*) [Selic94]. The language uses a limited number of statements, among which the so-called *busy* statement (see Figure 7, the highlighted line) is part of the RPL environment. By using this statement one can output *free-hand* text in the output angio-trace stream, i.e. this is one way to put in the angio-trace output additional information about microRTS primitives and their resource demand. Another words, the *busy* statement is like the *printf* statement used for debugging purposes.

In the process of working on this thesis there was a number of problems encountered. During the work the author solved some of those problems; some are marked as something outside the scope of, but important to the core of this work. This chapter tries to list the problems encountered up to this moment (May, '99), as well as explain their nature and offer some ideas for resolving them.

3. Overall Design of PAMB

The *Performance Analysis with Model Building* (PAMB) tool [Woodside98] is a tool intended to investigate a performance engineering subsystem for a design tool like ObjecTime's *Developer*. It implements Curtis Hirschuck's model building technique [Hirschuck97] and includes some existing components, based on concepts developed earlier in the department [Rolia95] and [Woodside95]. The general architecture of PAMB can be described as a collection of individual programs interacting within a common paradigm.

The core process of the approach is the low-cost, automated construction of an LQN model [Woodside95] from an ObjecTime design and a set of scenarios. Here by *scenario* the author means description of the end-to-end execution of the design in response to a specified set of inputs. Scenarios are the common unit of work that the system must perform in the ObjecTime design, the deployed system, and in a performance model.

This section gives an overall description of the PAMB tool, as designed and implemented by Curtis Hirschuck [Hirschuck97]. The high-level view of how the software developer would use the PAMB tool is shown on Figure 8 below.

Figure 9 - High-level Performance Analysis Methodology with PAMB

At the beginning, the analyst enters the ObjecTime design and validates its operation. Then the design is executed (in simulation mode) to produce end-to-end description (another words, *traces*) of its behaviour for specific application scenario(s). Traces are a low-cost source of behaviour information because they can be automatically generated from a variety of sources. A special tracing format is used and it is called *angio-trace* (introduced in [Hirschuk99]). As described in [Smith90], only those scenarios judged to be most significant are selected, such as heavily used requests or requests of special importance. The PAMB then constructs its own performance model from the scenario execution description(s). The PAMB evaluates its internal model to produce performance predictions for a *performance experiment* specified by the analyst. The performance predictions are average values for processing response times and resource utilisation. The software analyst analyses the predictions to check whether the design satisfies the requirements or needs to be adjusted.

3.1. Performance Tuning with PAMB and the "Wideband Approach"

The tool implements a technique for performance modelling, called *wideband approach* [Woodside98], which is a low-cost approach to performance analysis using models. The goal of the wideband approach is to detect performance problems as early as possible and, in the event expectations are not met, to provide useful information for tuning. Central to the wideband approach is that the tuning process uses abstract performance models to hint where the effort is most needed, and it can begin with the first model. The approach does not eliminate the need for a performance expert; it rather provides a model with which software developers and performance experts can communicate efficiently, focusing on the problems of performance analysis.

The general tuning process follows steps 4 through 8, as depicted on Figure 9 below.

Figure 10 - Tuning with the Wideband Approach

First specific scenarios are chosen and angio-traces for them are generated (steps 1 and 2). The process of picking specific scenarios for execution can be simplified, if few simple rules are followed:

- ◆ Pick scenarios, which occur frequently;
- ◆ Pick scenarios, which are expected to have a high resource usage;
- ◆ Use as source the system specification, requirements, design documents, any test documents, experience.

Once the model is generated (step 3) from the generated trace(s) and after that evaluated (step 4), the performance analysis is complete if the performance predictions meet the performance objectives (step 5). As we can see, the wideband approach not only accommodates information from a multitude of sources but it can be applied throughout the lifecycle of the design. It can begin early in the

specification and requirement stage for feasibility tests, continue with periodic feasibility reassessment or tuning, and then used for post-deployment performance considerations. The following Figure 10 depicts the lifecycle during the design of a software system, using the PAMB tool for performance modelling.

Figure 11 - Traceability with PAMB and the Wideband Approach

There are a number of advantages in this approach, like:

- The tool operates in the software engineers domain, i.e.
 - ◆ hide performance model building, evaluation, and interpretation;
 - ◆ assumes minimum knowledge about performance theory;
 - ◆ use of familiar metrics: utilization, waiting time, throughput;
- Through automatically building performance models
 - ◆ minimizing the requirements for performance expertise;
 - ◆ includes actor architecture and scenario behaviour;
- Map performance predictions to software models
 - ◆ utilization to devices, actors, threads;
 - ◆ waiting and service time to behaviour, i.e. use-case or MSC;
 - ◆ use of LQN method, which is scalable and very suitable for modelling large systems

3.2. Building the performance model

In this section we present in more detail how the PAMB tool works internally and in regard to the input, provided by the software developer. If we refer to Figure 8 below, building a model with PAMB with proceed as follows:

- (1) - generate the angio-traces (as one or more angio-trace files) by executing the ObjecTime design in simulation mode;
- (2) - abstract the trace(s) into a structured graph, which provides a higher level abstraction for analysis;

Figure 12 - Building Performance Model with PAMB

- (3) - detect the occurrence(s) of actor interactions and entries by matching sub-graph templates with the structured graph. This would produce a partial workload model for the given scenario, called a *sub-model*;

- (4) - specify the LQN model by including for each scenario its corresponding sub-model;
- (5) - finish the model by assigning parameter values, which configure the model for its target environment;

The last step introduces parameters for resource demands and the target system's environment. Resource demands are introduced for each actor's activities using a set of resource function environments supplied by the Resource Function Collection and Management (ReFCAM) utility, which is described later in this work.

The PAMB tool can be used not only for building a performance model. It has an interface to vary a model's parameters, evaluate the model, and present the resulting performance predictions in user-friendlier format. The steps in this process are shown on Figure 12 below.

Figure 13 - Evaluating Performance Model's performance predictions

- (6) - the developer assigns the remaining parameter variables a value (or a list of values) to evaluate;
- (7) and (8) - for each variable assignment a model is generated and evaluated by analytic or simulation means;

- (9) - once the suite of models are evaluated, the performance predictions are gathered, filtered, and sorted by scenario, task, and device; then they are presented to the software developer as a set of graphs, or as sequence diagrams [UML98] with performance annotations;

The use of angio-traces as the input behaviour description allows the wideband approach to accommodate many situations, such as multiple ObjecTime designs that interact, a multi-site development effort, or assessing the performance of designs where the source code is not available (e.g., due to intellectual property rights). Additional benefit is the fact that an angio-trace can be extracted from many sources, such as annotated specifications (e.g. Use-Cases [Jacobson] or sequence charts).

The PAMB user interface

Using the ReFCAM tool, the generated resource demand data will be converted and stored in a database, which data can be used afterwards by PAMB in the process of building of performance model(s). The PAMB front-end graphical user interface (GUI) is shown on Figure 13 below.

Figure 14 a) and b) – PAMB tool front panel

The figure depicts the front end with the *Options* menu pulled down, as well. This is done to show how the connection with the other components in the workflow process (ReFCAM and the test harness) is done.

What this means is that the GUI simply provides some step-by-step process algorithm for building the performance model for a software system – from resource demand data collection, its storage and manipulation in a database, to its use in the *PAMB* tool for building of the system's performance model.

The front end (Figure 14a.) is basically a layer, which offers the means to invoke separately developed components. Each component would perform a single step in the whole workflow process – collect data, store the data in a database, etc. Thus, each previous step in the process would prepare the ground for the next one.

When this work started, the *PAMB* tool was a set of scripts, executed in specific order to achieve the final goal - building of a performance model. The use of the tool was difficult and labour intensive. During a series of discussions, the author and the main creator of *PAMB*, Curtis Hrischuk came to the idea for *workflow* in *PAMB*. This is how the front end of the *PAMB* tool came to existence. The further implementation of this front end depends on the further development of the *PAMB* tool. The currently existing GUI for the *PAMB* tool should be developed further to make it more practical for use, or it will remain just an exercise. Currently there is initiative to have a presentation of the *PAMB* tool to the ObjecTime staff and for this purpose further development of the *PAMB* GUI might be needed. The author added the menu-bar at the top of the front window (Figure 14b.), which gives the means to access the ReFCAM and the test harness from inside the *PAMB* tool.

3.3. Retrieving Resource Functions

One of the concepts in the original design of *PAMB* was the abstraction of a *selection policy object*. A selection policy object selects one object from a set of objects, according to a given algorithm that is provided with other attribute information. A selection policy object will select the most appropriate item for a given set of information and resolve any conflict and/ or ambiguities.

Second concept in the original *PAMB* design was the abstraction of providing a resource function table for different target environments. A *resource function table* identifies all of the applicable resource functions for a given environment, where a target configuration is an instance of an environment. A resource function table has a column of attributes, which map into a resource function.

The original design of *PAMB* implemented a four-step process for retrieving a resource demand object for an activity during the building of a performance model. These four steps are illustrated in Figure 15 through Figure 18, where an ellipse is a *Perl* object and a directed arc between two *Perl* objects is a method invocation.

Figure 15 - Step 1: Resource table selection initialisation

Figure 16 - Step 2: Selection of a Resource Function table

During the first step (see Figure 15) the user identifies selection policies as a command line option (1), or through an environment variable, or as an item in the model configuration file. Then the model manager (2) sends the identified policies to the resource table manager, which instantiates the table selection policy (3).

During the second step depicted on Figure 16, the model manager requests (1) environment attributes, which are assembled (2) into a *Perl* hash table and sent to the model manager. These environment attributes are forwarded to the resource table manager (3), which identifies candidate resource tables for this environment and sends them (4) to the table selection policy. The table selection policy requests more information (5) from the user to resolve the selection conflict, if needed. The user provides (6) the requested information, which the table selection policy uses to select (7) a resource table from the candidates and informs the resource table manager which table was selected. The model manager receives a reference to the selected table (8) and now can communicate with it.

On the next two figures (Figure 17 and 18) we depict the next two steps of this process. During step 3, depicted on Figure 17, the user identifies (1) selection policies for the resource functions as a command line option, an environment variable, or as an item in the model configuration file. Then the model manager sends (2) the identified policies to the resource function table. The resource functions table instantiates the proper selection policy for resource functions.

Figure 17 - Step 3: Resource function table initialisation

Figure 18 - Step 4: Selection of a resource function

During the last step the model manager requests (1) resource function attributes and parameters, which the activity returns (2) as two *Perl* hash tables. The resource function attributes and parameters are forwarded (3) to the resource function table, which identifies (4) several candidate resource functions and sends them to the resource function selection policy. The resource function selection policy requests (5) more information from the user to resolve the selection conflict (if needed), which the user provides in (6). The resource function selection policy uses the provided information to select (7) a resource function from the candidates and informs the resource function table which function was selected. The resource function table sends (8) the parameters to the resource function, which responds (9) with a resource demand object. The resource demand object then is sent (10) to the model manager.

The major drawback in this approach was the assumption and usage of resource functions with a high number of independent parameters. The nature of the resource demand data is such that it often depends on a high number of variables, which in turn define the complexity of the corresponding resource functions. In practice this means that each resource function could have between 1 to n independent parameters, which could vary in a range of values. This leads to creation of very complex resource functions, and using them would have complicated their practical usage.

As it can be seen in the above algorithm description the original PAMB tool allowed for some ambiguities, which led to (potential) conflicts in the resource function selection. In turn, this required additional input/feedback from the designer to resolve the conflict(s).

The changes to this approach, undertaken in this work, are explained in the next chapter *Additions to PAMB*.

4. Resource Function Additions to PAMB.

This chapter describes the design of the retrieval interface and how we overcome the problems described in chapter 3 of this work.

The original PAMB tool would assign resource values to the LQN model nodes by using a pool of resource functions, from which the tool would collect resource demand data during the process of building performance model (see section 3.3). In some cases those functions would have a (potentially high) number of independent parameters, i.e. the complexity of those functions becomes an obstacle to their easy and correct storage and usage. There were few drawbacks in this approach:

- the amount of resource functions to scan through in order to find the proper one, i.e. the question of time spend searching arises here;
- the complexity of the resource functions, i.e. the problem of multi-dimensional resource demand data and their representation;

One of the goals of this work is to make the PAMB tool easier to use and perform some research in the ways of easy storage and manipulation of resource functions. One of the first problems to resolve in regard to this goal was to decide about to what depth of demand data collection this work should engage. From Figure 5 we can see that a model designed with ObjecTime Developer runs on top of the ObjecTime *microRTS*, which in its turn runs on top of the real-time OS of the specific target. One approach would have been to go down to the RTOS level and try to measure and collect demand data on that level. However, experience with ObjecTime customers show that most often their models consist of about 70% to 80% of calls to functions of the *microRTS*. This lead to a decision that this work conceptually concentrates on the resource demands of the functions implemented by the ObjecTime *microRTS*.

4.1. Principles of Experimental Design

When measuring resource consumption values, the choice of parameter values for the measurements is called the *experimental design*, and each combination of parameter settings is called a *design point*. The resource consumption values form a *response surface*. There is a large field of research devoted to experimental design and response surface technologies [Mayer89]. This section will touch only on some main points in this area. A response surface captures how the value of a dependent variable (the response or the result) varies as the values of some independent variables (the parameters) change. The main idea behind these methodologies is that an experiment is designed to explore and model some limited area of the response surface, which the modeller is interested in. For instance, a simple response surface, which can be modelled as a first order linear hyper-plane requires fewer points to be

collected than a more complicated second order model. The underlying assumption is that the modeller knows at least approximately, what the response surface looks like in the area of interest. If the modeller does not know then it is assumed that s/he will engage in an iterative process [Box87], i.e. collecting points, modelling them, assessing the closeness of fit. If the fit is unsatisfactory, s/he will be changing the functional form of the model and collecting any additional points to satisfy that form.

When the modeller is uncertain of the form of the response surface, [Box87] suggests beginning with a first order fit and a 2^K factorial design, where K is the number of independent parameters. In such a design, two levels of each parameter are considered and a design point exists for each combination of parameter values. Figure 19 depicts a design, where three parameters are involved and each parameter can take one of two values (e.g. -1 or 1).

Figure 19 - Visualizing a 2^K factorial design with three parameters

Box's proposal is to start with a simple 2^K factorial design, model the data using the least squares, with a first order polynomial, and if the fit is not adequate, move to a more complicated model, such as a second order polynomial.

From this discussion, we can construct two principles to help us in the construction of resource functions:

- the process must be iterative, beginning with a simple model and progressing, if necessary, towards more complex model;
- more complex models will require more points.

4.2. Resource Data and Angio-Trace Generation

The ObjecTime Developer tool provides the capabilities to generate angio-traces directly from a software design. This is achieved by developing the design using ObjecTime's *RPL* (Rapid Prototyping Language), and then running the design prototype in simulation mode from inside the toolset.

The current ObjecTime toolset allows for the generation of angio-trace output only for communication primitives, like *send*, *reply*, etc. Currently the toolset implements the first generation angio-tracing [Hriscuk99], which has a number of limitations. Among the most relevant to this work is the limitation for generating the parameters to the communication primitives into the angio-trace output stream. Those parameters are used later on by the ReFCAM tool to determine the proper signature of the software primitive and return its resource function.

In addition to the communication primitives, the ObjecTime *microRTS* implements a large number of software primitives, used in the developing of software designs through the ObjecTime toolset. This has been one of the reasons that an additional language construct has been added to the ObjecTime's RPL - the *BUSY* statement. For angio-traces, one can consider this statement as equivalent of the *printf* statement in the standard languages. The statement takes as input any string and outputs it in the angio-trace stream - an example can be seen on Figure 8.

The *BUSY* statement has been used in this work for the purposes of generating angio-traces for the rest of the implemented by the ObjecTime *microRTS* software primitives, used by the software designs in chapters 7 and 8 (see Figure 36 for an example). The general syntax of *BUSY*, used to generate an angio-trace comprehensible for the ReFCAM tool is shown below:

```
BUSY 'method-name object-type <composite-object-type | none> object-size'
```

method-name :: new | delete | copy | assign

object-type :: RTCharacter | RTInteger | RTBoolean | etc.

composite-object-type :: object-type

object-size :: digit<digit>

The *method-name* rule lists only four method names. For the purposes of this work only these methods, which are common between all *microRTS* classes have been selected to be managed by ReFCAM.

The following two examples show how the *new* method for an integer object will be output, as well as the *delete* method for an array of characters:

```
new RTInteger none 1
```



```
delete RTSequenceOf RTCharacter 100
```

The ReFCAM tool then interprets the angio-trace output from the *BUSY* statement and the corresponding resource function is passed to the PAMB tool during the performance model building.

4.3. PAMB process workflow

As part of this thesis the author had to implement some changes to the general way the PAMB tool worked. This involved designing and implementing some GUI for the tool, as well as changing some of the background processing done by the tool. At the time of writing this paper the tool was in the form of a number of scripts, which were used as command-line executables. In order to simplify the use and present the whole process in a more visual form, a simple front end GUI was developed - with the idea to introduce some *workflow* concepts in the whole process. The author worked together with Curtis Hrischuk to design and implement the GUI of PAMB.

The idea behind the workflow concept is to provide easy to use work environment for the software developer. This way he will be able (in an organized way) from inside the PAMB tool to invoke the test harness, compile, load and load the synthetic application(s) on specified platform, after which invoke ReFCAM from inside PAMB, too. All those actions should be performed in an ordered and structured manner, where the GUI of the tool will have to help/impose usage in such manner.

The general workflow process of collecting, manipulating and using resource data is depicted on Figure 20 below. The workflow process will go as follows:

Figure 20 - The PAMB workflow process

- 1 - invoke the test harness to compile, load, run synthetic application(s) and collect output results (i.e., resource demand data) on a specific platform (dashed line means it can be optionally invoked from inside PAMB);
- 2 - the output results from running the synthetic application (s) get stored in log files;
- 3 - invoking the *ReFCAM* to import the obtained results in a database (dashed line means it can be optionally invoked from inside PAMB, or invoked separately);
- 4 - the result log-files are read in by the *ReFCAM*, converted and stored in database format;

5 - the results get stored in a database with resource demand data;

6 - the PAMB tool reads in the stored database information in the process of building the performance models.

It can be seen from the figure above that the PAMB is the central command centre, which offers the end-to-end performance modelling process. This means that the designer can decide for which target platform a performance model will be built, then compile, load, and run the synthetic application (s) on this target, then collect the produced resource demand data and import them into a resource demand database. After that a performance model can be build using the freshly collected resource demand data.

4.4. Resource demands

In the PAMB tool workflow process steps 4 and 5 (see Figure 14) allow the user to select a resource demand database, after which to pick for use the proper resource functions from this database. The next two sections describe in detail this process.

Selecting resource demand database

On Figure 21 below is depicted the *Select demand data* window, which would pop up as a result of selecting step 4 in the PAMB workflow process.

Figure 21 - Selecting a resource demand database

This window gives the user the option to browse and choose a resource demand database by pressing the *Choose* button. After choosing and loading a resource database, a summary view of the contents of the chosen database will be displayed in the list-box. The list-box will show all the resource environments, available in the chosen database.

The list-box on this window will show the main database entries, which a resource environment consists of, like the platform, the CPU, the name of compiler used, etc. Here the user is presented with the option to view the contents of the resource demand database (to decide whether this is the right one, for example) before proceeding to the next step of building the performance model.

A mouse double-click on a list-box entry will bring up a window (see Figure 23) with all the database information on this entry.

Mapping resource demand data

The window depicted on Figure 22 below would pop up as a result of selecting step 5 in the PAMB workflow process (see Figure 14). This is the *Define Environment* window, which presents the user with the options to define the environment to be used for specific processor in the performance model building. Here the user is presented with the option to map an environment to a processor.

Step 5 [Icons]

Define Environment

	Processor	OS	CPU	Compiler	Rate	Multi-server	Results
Add	default	TORNADO1	IS60	cygnus-2.7	1.0	1.0	
Change							
Delete							
Reset							

	Actor	Processor	Priority	Multi-server	Results
Add					
Change					
Delete					
Reset					
Load					

Save Filename: **Choose**

Previous **Next** **Dismiss** **Help**

Figure 22 - Mapping resource demand data

The top part of the window depicts the information of the used resources (i.e., processors) by each individual task/actor. In the top part of this window the user has the option to define all the processor(s), which exist in the system. For each processor(s) one can define the type of OS running on each one of them, the CPU type, the compiler used to compile the task to run on this specific CPU, etc.

The bottom part presents the relationship between those tasks/actors and the processors. In the bottom the user can define the relationship between CPU and tasks, the priority each task runs on the CPU, etc.

In case the user presses the *Change* button, the screen on Figure 23 below will pop up. Here the user has the option to change the mapping between a processor and the environment, which this processor will work in. The bottom text widget on this window (labelled *Processor*) shows the processor, which is to be mapped. The list-box above shows the available resource environments in the currently selected database (the one, selected in step 4 in the modelling process). By selecting (left mouse-click) on a list-box entry and pressing the *Map To* button, the processor will be associated with the chosen environment. The resource functions for this environment will be used further in the performance model building.

Figure 23 - Summary of mapped resource demand data

Again, as in step (4) by double clicking on a list-box entry a pop-up window (see Figure 23 above) will be shown. This window will present all the data available about the chosen resource environment. For example, the data will include the date of generation of the resource functions for this specific environment, the OS to run on, the CPU characteristics, compiler used and parameters to this compiler, etc. The entries in this window are explained in more detail below:

- *Date Generated* - this text field contains the generation date of the resource demand data for this specific environment, i.e. the date when the test harness loaded and ran the ObjecTime synthetic application(s) and collected the resource demand data;
- *Run-time Platform* - the name of the RTOS running on the CPU, for which data was collected;
- *CPU* - the make/type of the processor;
- *CPU Clock-rate* - the clock rate of the CPU, for which resource demand data was gathered;
- *Compiler* - the name of the compiler, used to compile the ObjecTime synthetic application(s);
- *Compiler Version* - the version of the used compiler;
- *Compiler OS* - the name of the RTOS running on the CPU, for which data was collected;

The screenshot shows a window titled "View DB" with a list of resource environment data fields and their values. The fields are: Date Generated (25-10-1997), Run-time Platform (Tornado), CPU (m68040), CPU Clock Rate (empty), Compiler (cygnus), Compiler Version (2.6), Compiler OS (SUN5), Compiler Flags (empty), Heap [kB] (150), uRTS used (Release 5.20.B11), Units of Measure (1e-06), Selection Policy (ResFuncSelPolicy), and Comment (Generated for testing pur). A "Dismiss" button is at the bottom.

Date Generated:	25-10-1997
Run-time Platform:	Tornado
CPU:	m68040
CPU Clock Rate:	
Compiler:	cygnus
Compiler Version:	2.6
Compiler OS:	SUN5
Compiler Flags:	
Heap [kB]:	150
uRTS used:	Release 5.20.B11
Units of Measure:	1e-06
Selection Policy:	ResFuncSelPolicy
Comment:	Generated for testing pur
Dismiss	

Figure 24 - Resource environment data window

- *Compiler Flags* - the command-line flags, passed to the used compiler;
- *Heap* - the size [kb] of the heap, which the synthetic application used during its measurement run;

- *uRTS used* - the release ID of the ObjecTime's microRTS, used in the resource demand data collection;
- *Units of Measure* - defines the scale used for the presentation/storage of the resource data.
- *Selection Policy* - specifies the type of selection policy used in the process of performance model building
- *Comment* - free-text comment, entered to describe some additional conditions/information;

This window offers read-only capabilities; i.e. one can not change the resource environment properties from this window.

4.5. Retrieving Resource Functions

The original design of PAMB included the implementation of the four-step process for resource function retrieval (see section 3.3). This was a somewhat complex algorithm, which did not provide a satisfactory way for retrieving of resource functions during the model building process.

There are a number of criteria that must be considered in choosing and building a resource function, which characterises any resource demand. These include the following:

- *Generality* - to be employable in different systems and/or platforms.
- *Computational effort* - this is in direct connection with complexity and computability of resource functions, as mentioned in section 2.4 of this work.
- *Storage requirements* - each resource function requires some amount of resource data, which is used in the process of building the resource function and calculating incoming resource demands. The issue of enough resource data is raised here.
- *Numerical stability* - to produce deterministic results for a wide range of parameters.
- *Implementation effort* - the need to consider design/development effort necessary for the practical implementation of a resource function.

To achieve the above goals, the author implemented a class hierarchy in the Perl programming language, which provides the means for storing and calculating resource demands. The measured demand data then is grouped into *environments*. An environment here combines measurements, made for specific values of the main factors, defining the performance demands of the models, using the ObjecTime target RTS. Those are, to mention few, the CPU on which a model will run, the CPU clock

rate, the compiler used to compile the design, the compiler options, the type and version of the RTOS running on this CPU, etc.

When designing models in ObjecTime Developer, about 70% to 80% of the code uses functions, implemented by the ObjecTime target RTS. Thus, it was decided to measure demands only for the main functions implemented by the target RTS. Currently, all the functions in the ObjecTime target RTS have zero or only one independent parameter (like *send*, which resource demand varies with the changes of the size of the message being sent). Thus, the author implemented the resource functions by simply storing the measured demand data points as Perl arrays.

The next step was to decide how to obtain the correct resource demand upon receiving a request for some unknown in advance value of the independent parameter. It is well shown in the literature (e.g. [Prenter75]) that any set of $(N+1)$ points in two dimensions can be interpolated by Lagrangian polynomial of degree N . With such a function as N increases, its behaviour in between the points can vary drastically from that of the true function. In practice, instead of fitting the points globally with one polynomial, a number of simpler, smaller-ordered functions can be used to fit points in a certain interval. The points at the boundaries of each interval, where the functions are joined, are referred to as *knots* and the resulting approximation is known as a *spline*.

To simplify the matter, for the purposes of this work the author started by using truncated power splines for interpolation of the measured demand data. The power splines use as basis function the truncated power basis function [Friedman89]. There are some other possibilities for implementing resource functions. For example, [Vetland97] and [Courtois97] suggest some practical methods for statistical fitting of resource functions using multivariate adaptive regression splines (MARS). In this work, the author did not consider those methods. However, the current implementation allows for easy and quick embedding of those methods.

4.6. Class hierarchy of PAMB

The current general class hierarchy (in UML notation) of the *PAMB* tool is shown below on Figure 25 below. The diagram does not show all the classes (and details) in the *PAMB* class hierarchy, only those directly involved in the resource function selection and usage.

Figure 25 – Overall PAMB class hierarchy

The main topics mentioned in this chapter are about the process of picking the right resource function(s) for the right environment. Another words, the process of resource function selection has two phases:

- given data about the resource environment we currently attempt to build performance model for, how to pick the closest match;
- given data about the software primitive we currently search performance resource function for, how to pick the closest match;

The first is taken care of by the GUI provided by the PAMB. Through the GUI a user can pick the right resource database, open and view its contents and if satisfied with it, map the proper resource environment(s) to the resources in the system.

4.7. Resource Function Management Module

The PAMB class hierarchy contains a sub- hierarchy of classes, which are shared between the ReFCAM and PAMB tools. Those classes contain the logic for the resource function module in the PAMB tool.

4.7.1. Class Hierarchy of the Resource Function Module

The following Figure 26 shows (in UML notation) the class hierarchy for handling of resource functions.

Figure 26 – Overall ReFCAM class hierarchy

The class hierarchy of the ReFCAM presents the main *Perl* classes, involved in the process of resource function handling. It presents how the PAMB and the ReFCAM tool access resource demand data for the purposes of resource function building.

4.7.2. Specifications of the Resource Function Module

This section presents the specifications for the ReFCAM module, as implemented by the PAMB and the ReFCAM tool. The following *Perl* classes constitute the ReFCAM module:

ResEnvMgr

This is the main class, which inherits from the other resource data classes.

ResFuncMgr

This is the class, which contains resource data about a specific resource environment.

ResFuncSelPolicy

This class provides methods for selecting resource function according to some criteria.

ResFuncEnv

This is the top class, which presents the main functionality.

ResFunc

This class is used to store information for a specific resource function, as well as methods for manipulation of the data. It contains data like for what resource this resource function is, the name of the software primitive it characterises, etc.

Spine

This class is used to implement methods for spline approximation. It is used by the ResFunc class when a request comes in for resource demand.

4.7.3. Structure of the Resource Data Tables

The above described classes of the ReFCAM tool implement a series of tables used to store the demand data and the relationships between these data. The following sections give a brief description of the most important tables, which are the resource environment table and the resource function table.

The resource environment table identifies the context in which resource functions have been measured/collected. The fields of this table are as follows:

- **PLATFORM** - the platform, on which the demand data measurements were taken. The platform is a combination of CPU-type, RTOS running on that CPU, and the compiler name/version used to compile the software designs.
- **GENERATED** - the date when experiment took place and the demand data generated.
- **CPU_TYPE** - the CPU type the demand data was taken for.
- **CPU_SPEED** - the CPU's internal clock speed.

- **COMPILER_VENDOR** - the name of the vendor of the compiler used to compile the software designs to run on the specific CPU/RTOS.
- **COMPILER_VERSION** - the version of the compiler used to compile the designs.
- **COMPILER_PLATFORM** - a number of real-targets have cross-compilers, which run on different platforms. This field specifies the platform name in case a cross-compiler is being used.
- **RTS_RELEASE** - the ObjecTime's microRTS version, for which software primitives the demand data has been collected.
- **OPTIMIZATION** - source code optimisation level. Contains the compiler flags, used during compilation of the designs.
- **HEAP_SIZE** - the heap size, which the models in SAS (see section 5.4) used during the process of demand data measurements. The bigger heap size means more correct measurements.
- **MEASURE_UNITS** - units of measurement, which scale the demand data values. This field is used to transform the collected measurements for internal PAMB usage.
- **COMMENT** - free-hand description/comment.

The resource function table contains data, which is used by the resource functions to return the correct value requested. The fields of this table are as follows:

- **OBJECT, METHOD** - identifies the target RTS function.
- **RESOURCE** - the resource type this resource function characterises. Currently *CPU* is the only one supported.
- **MODEL** - a Perl array with data points.
- **SUBTYPE** - the sub-type of the object. For example, the microRTS type *RTSequenceOf* can have a sub-type of *char*, *RTInteger*, etc.
- **COUPLING** - in case of a communication primitive, describes whether inter- or intra-thread communication has taken place during measurements.

- **REPLICATED** - a boolean field, which is set if the communication target is replicated, i.e. if a communication primitive *"talks"* to more than one recipient.
- **CONTENTS** - in case the **OBJECT** value is *Actor*, it identifies if it was a simple actor (no sub-actors contained), or a complex one. Could be either *Simple* or *Complex*.

The ReFCAM tool uses the tables described here to properly identify in-coming requests from PAMB for demand data.

5. Resource Demand Data Gathering and Management

This chapter describes the general problems encountered during the process of automating the gathering and managing of resource demand data. This data had to be processed and stored in format usable by the Resource Function Management mechanism, which in turn would select and pass the required data to PAMB tool.

In general, we apply the term *resource functions* to describe data about resource demands such as the CPU consumption, bus traffic, file system I/O-s and so forth, which is generated by the operation of a program. They are functions because they may depend on the size of messages or other operation parameters. In this work the author was concerned mostly with CPU consumption resource demands.

A performance model needs data on these resource demands at some level of detail, to provide parameters for the model. A user would like to have a library of values for overheads and for frequently used components, interface so that values are inserted automatically as much as possible. It needs research on the model architecture to create an interface capable of doing this correctly.

From Figure 6 in chapter 2, the two main components here are:

- the test harness, which provides the means to compile the ObjecTime synthetic applications, run those applications on specified platform, and collect the output data (i.e., platform measurements);
- the synthetic applications (SAS), used to be loaded and run on a specific platform and perform measurements of the ObjecTime's *microRTS* primitives

The following sections present a detailed description of those two components.

5.1. Approach to Measurement of Demand Data

In the preceding chapters the attention was on solution of models and on system characteristics, expressed through resource functions. In doing so, we have assumed that numerical parameters are given to use or are readily available. In this chapter we turn our attention to measuring, gathering, and storing resource demand data.

There are three main sources of measured data: existing accounting software, hardware monitors attached to the system, and software monitors, i.e., software added to the system specifically for measurement. Measurements are obtained principally by hardware or software monitor supplemented by accounting software and hardware specifications. Depending on particular situations and parameters, the resource demand data may be obtained directly or through intermediate values, which

one can obtain directly. This latter approach is very similar to methods used in simulation to obtain measures.

Accounting Software and Hardware Specifications

Most computer systems nowadays include software to determine user's resource usage (like CPU time per user/task, turnaround time for batch jobs and active time for batch jobs, etc.) so that they may be billed for their activities.

A principal problem when using accounting software for resource demand data collection is that accounting data often excludes resource usage by the operating system not directly invoked by user programs.

Hardware Monitors

Since accounting data and hardware specifications are usually not sufficient for supplying resource demand data, it is usually necessary to use a hardware or software monitor to supplement these other data. A hardware monitor is simply a collection of digital circuitry, which is attached to the hardware of the computer system under test. Existing hardware monitors range from devices consisting of few circuits [Snelick90], [Flanagan92] to the most sophisticated computer systems including disks and other peripherals [Kroeger92], [Hofmann88]. Hardware monitors are widely used to estimate system performance directly, without modelling. Even though there are hardware monitors available for a wide range of computer systems, one disadvantage for their usage is the lack of such for all available computer systems on the market today.

The main points of interest are the *probe points* in the system, where we attach the monitor *probes*. The probe points are places where we can measure the voltage levels corresponding to system states (such as CPU busy/idle, particular memory location being referenced, etc.). This measurement process will usually be *transparent* to the observed system, i.e. there will be no difference in system performance as result of a monitor attached. This is one of the principal advantages of hardware monitors over software monitors.

Another advantage of hardware monitors is that they can provide very accurate estimates for system parameters. However, they are inherently limited to measuring information we can interpret at the hardware level without knowledge of OS activities. This means we cannot easily obtain CPU service times by process, for example, and implies that we cannot obtain distribution estimates other than the mean for parameters such as CPU service times. For these reasons, hardware monitors are usually not sufficient for the purposes of resource demand data collection.

Software Monitors

A software monitor is a collection of pieces of code embedded in the OS to gather performance data. Depending on the architecture and OS, some of the code may be run at user level processes. There are two major approaches to the design of such a monitor: the *event* approach and the *sampling* approach. Any software monitor will perturb system performance somewhat, since it requires system resources to execute. The event approach has the advantages of maximum flexibility and generality, but it is likely to perturb system performance more than a sampling monitor does. In addition to providing more control over the overhead introduced by the monitor, in the sampling approach it may be more convenient to add a monitor to an existing operating system.

In the event approach, the designer/user of the monitor must define significant events, e.g., CPU is switched from one process to another, I/O request issued, etc. The RTOS modules, which effect such events must be modified so that when the events occur, the module records the type of event, the time it occurred, and any other important data about this event. The event records are processed after the measurement period to obtain the resource demand data. This approach allows the gathering of very detailed data.

In the sampling approach, monitor code is enabled periodically to determine whether the CPU is busy, what a particular device is doing, etc. Using this information one can directly estimate performance metrics such as CPU utilization, and thus indirectly estimate total CPU service time. The sampling approach is much less general and flexible than the event approach. For example, there are parameters such as variances of service times, which cannot be feasibly estimated with the sampling approach. The advantages are potentially much simpler implementation and the ability to directly reduce system perturbation by reducing sampling frequency.

Depending on the particular system and monitor, a software monitor may consume 20% or more [Sauer81] of the system resources, and particularly CPU and channel time. Thus software monitors usually produce very questionable results. However, by appropriate definition of events and implementation, this overhead may be kept down to about 5%. Such software monitor results are usually sufficiently accurate for a high number of performance engineering purposes.

Besides the inherent perturbation of system performance, there are two other significant problems with software monitors.

- the amount of data produced by the event trace may be overwhelming, particularly in terms of reducing the trace data after the measurement period;

- unlike hardware monitors, software monitors must be specifically designed for particular architectures and OS-s. Thus, there is no commercially available software monitors for many significant computer systems. Further, implementing a software monitor for an OS that already exists may require a significant amount of effort and expertise.

In the end it is worth mentioning that any of the above sources of resource demand data may also be very valuable in providing data for validation of the created models. For example, we can estimate the CPU utilization by the sum of CPU times divided by the length of the measurement period. If our model estimate of CPU utilization agrees well with our measurement estimate for the system without modifications, then we can place more trust in the model estimates for the modified system.

5.2. Test harness

This section presents work, which is in direct connection with the previous section. While working on this thesis and developing ways of collecting resource information for the specific platforms, for which performance predictions were to be produced, there were a number of obstacles to overcome. One such obstacle was the amount of resource demand data to be generated, collected and managed in the whole process of building performance model(s) for a target platform. This introduced the need to create the means to automate (fully or to some level) the process of producing and gathering such resource demand data. This is how the test harness (known as *autoTest*, see Figure 5) came to be designed and implemented.

The data gathering for the ReFCAM is automated by this *test harness*, which runs the *synthetic application suite* to measure, collect, and store the resource demand data about the ObjecTime *microRTS* primitives. The front end of the test harness is depicted on Figure 27 below. It presents a simple GUI front-end to the user, which main components are two list-boxes. The list-box on the left contains the information about the ObjecTime synthetic application(s) (see section 5.4), used in the process of gathering resource demand data. The set of synthetic application(s) (the *Synthetic Application Suite* or *SAS*) was developed for the process of resource demand data collection. The models in *SAS* provide measured demand data for the most commonly used functions of the ObjecTime *microRTS*.

The list-box on the right contains the data about the available target platforms, for which the synthetic application(s) will be compiled and loaded on to run and gather resource data about them. The auxiliary text-box on the bottom of the screen displays run-time information about the tasks, performed by the test harness.

The test harness currently is in its third major version and it has been used in industrial environment (in ObjecTime Ltd.) to automate the process of weekly regression testing of developed software products.

Figure 27 - Test harness front end

The test harness systematically runs the synthetic application suite on each target platform by automating the compilation, target loading, running, and output result collection. It is developed in *Expect/Tcl* scripts [Ousterhout] since this was considered as the most appropriate tool for this purpose. The harness provides an easy and seamless way to add a new synthetic application and/or target platform. It is developed using some object-oriented approaches and means being offered by the *Tcl* version 8.0. The following two sections briefly introduce some of the functionality of the test harness.

Synthetic applications List-box

The left-hand side of the main screen of the test harness presents a list-box, displaying the synthetic application(s) currently used. A double-click on any of the list-box entries will bring up the window depicted on Figure 28 below. This screen offers the option of entering and presenting all the data about a synthetic application. The fields on the window contain specific data about the synthetic application. The following gives brief overview of those fields:

Update Name - the name of the synthetic application;

Update File - full path to the ObjecTime's synthetic application;

Actor Name - the entry-point actor to be compiled;

Language - the implementation language used in the synthetic application;

Test parameters - command-line parameters to the synthetic application;

Figure 28 - Model Data Screen

Test timeout - the maximum time the test harness would wait for the synthetic application to finish execution, before deciding there is something wrong and kills its process;

Platforms List-box

The right-hand side of the main screen of the test harness presents a list-box, displaying the platforms currently used to run the synthetic application(s) on. A double-click on any of the list-box entries will bring up the window depicted on Figure 29 below. This screen offers the option of entering and presenting all the data about a target environment. The fields on the window contain specific data about the target.

The following gives brief overview of those fields:

Target - the name of the target environment

Library - name of pre-compiled ObjecTime library for this target

Compiler Host - name of OS environment to compile for this target

Compiler - name of compiler

Compiler Flags - options for the compiler (optimisation, debug info, etc.)

Target Platform: TORNADO101T.m68040-cygnus-2.7.2-960126.HPUX09 ☐ Generate

Target: TORNADO101T

Library: m68040-cygnus-2.7.2-960126

Compiler Host: HPUX09

Compiler: cc68k

Compiler Flags: "-O4 -finline -finline-functions -Wall -m68040"

Make Command: make

Executable Ext:

Pre-compile: OS_HOME=/disk8/os
WIND_REGISTRY=server0
WIND_BASE=\$OS_HOME/wrs/tornado-1.0.1

Post-compile:

Target Board: vme1

Target Type: tornado

Configuration: TORNADO101Tm68040cygnus272960126HP09

OK Close Apply

Figure 29 - Platforms Window

Target Board - network name of target board

Target Type - name of the specific target

Specifications

In order to store all the target and synthetic application data, the test harness uses a file. This file is in *Tcl* [Ousterhout94] format and describes straight *Tcl* data structures. The excerpt below shows a sample of such a file, which is read by the test harness.

```
##### Autotest data file -*-tcl*-  
  
##### new.atd  
  
##### Saved 11/04/98 17:52:55  
  
# BindingsTest  
::Update::Set BindingsTest {  
    actors {SystemTest}  
    alterExec {0}  
    .....  
    testParameters {}  
    textOut {1}  
    toolsetCompile {0}  
    updateFile {/home/tester/ObjecTime5.2/BindingsTest.update}  
}  
  
# TORNADO101T.i960-cygnus-2.7.2-960126.SUN5  
::Platform::Set TORNADO101T.i960-cygnus-2.7.2-960126.SUN5 {  
    compiler {cc960}  
    compilerFlags {-O4 -finline -finline-functions -Wall}  
    compilerHost {SUN5}  
    configName {TORNADO101Ti960cygnus272960126SUN5}  
    downlFact {}  
    execFact {2}  
    extension {}  
    library {i960-cygnus-2.7.2-960126}  
    makeCmd {make}  
    postCompile {}  
    preCompile {OS_HOME=/disk8/os  
        WIND_REGISTRY=server0  
        WIND_BASE=$OS_HOME/wrs/tornado-1.0.1
```

```

.....

export VXWORKS_HOME VX_BSP_BASE VX_HSP_BASE VX_VW_BASE

export GCC_EXEC_PREFIX

PATH=$WIND_BASE/host/$WIND_HOST_TYPE/bin:$PATH

export PATH}

target {TORNADO101T}

targetBoard {i960}

targetType {tomado}

}

```

This file is read in during harness' start-up time and used for initialization of platform and synthetic application data.

Applicability of the harness

The test harness is fully developed and currently in use at ObjecTime Ltd. Currently it has gone into its third version and has been fully deployed in the regression test cycles during product releases in the company's release cycles.

5.3. Constructing Resource Functions

To construct a resource function, an experiment must be designed and measurements must be taken. The number and location of measurements in addition to how the measurements are modelled will determine the accuracy of the resource function. In this section we examine some principles of experimental design and some recent research in resource function construction.

5.3.1. Resource Functions and User Code

This section covers how using the described in this work approach to automated collecting of performance data can be used to include gathering of resource demand data about user-written code.

The resource demand data collected by the performance models (see section 5.5) is strictly related to the software primitives provided by the ObjecTime *microRTS* (see Figure 5) on a specific target platform. While this data covers a good range of performance data, which an ObjecTime application would require to build its performance model, it does not provide 100% coverage. In a real-world application built with ObjecTime Developer, software designers add their own code, define their own data types, etc. This code very often utilises very many user defined data types and most of the time has the biggest impact on the performance of the created application.

With the creation of the set of performance models in combination with the ReFCAM tool, it becomes much easier to measure the resource demand of any user code and store it for later use by PAMB in the performance modelling. There are two options here:

- The users alter the developed set of performance models to include their own data types, and add their own code, as well.
- Users develop their own performance model, which should comply with the basic requirements to be manipulated by the test harness.

Following any of the above two approaches provides a relatively easy way for gathering resource demand data, and the data storage for later use by the PAMB tool.

5.3.2. Dimensions of Resource Functions

Resource functions are widely used to summarise the dependence of resource demands on the parameters of the action. A resource function is the empirical equivalent of a complexity function for an algorithm, and is valid for a range of values of the parameters of interest. Those parameters are known as independent parameters of the resource function and can be in the range of $(1..N)$, where N is any number greater than 1.

In section 2.4 we presented in more detail some of the main concepts in the complexity theory of resource functions. From that section there are two main conclusions, which can be drawn:

- The nature and amount of resource demand data for specific resource is such that very often the corresponding resource function implements a very complex algorithm. For practical uses this is very often an obstacle, which needs to be simplified in order to use.
- The literature does not contain enough relevant information for the type of resource functions this work concentrates on

This is why this work took in another direction in regard to using resource functions in performance modelling. Instead of constructing, storing and using resource functions with more than two independent parameters, the functions were *flattened* to have (to depend on) no more than one independent parameter. This reduction of the dimensions of obtained resource functions was achieved through decomposing the resource functions into simpler sub-functions, i.e. if we have the resource function $f(x,y)$ with independent parameters x and y , we can present it:

$$f(x,y) \simeq f_1(x)f_2(y)$$

The idea here was to separate function parameters in two groups - parameters, which change often and parameters, which are relatively static. This approach was taken after considering the type of software primitives, which demands were modelled by the resource functions. The approach ensured much simpler way of handling the collected demand data.

The following example explains this approach in more detail. The following is definition of the class *RTSequenceOf*, as implemented by the ObjectTime's *microRTS* (see Figure 5). This class is the equivalent of array data type.

```
class RTSequenceOf : public RTDataObject {
protected:
    void          spaceClean( void );
public:
    int          size;
    RTDataObject ** array;
    explicit RTSequenceOf( int _size = 0 );
                RTSequenceOf( const RTSequenceOf & );
                RTSequenceOf( int _size, const RTDataObject ** _contents );
    virtual ~RTSequenceOf( void );
    RTS_INLINE RTDataObject & at( int ) const;
    RTDataObject & operator[]( int ) const;
    RTSequenceOf & operator=( const RTSequenceOf & );
    virtual int operator==( const RTDataObject & ) const;
    virtual const RObject_class * getClassData( void ) const;
    static const RObject_class classData;
};
```

This code and the whole *microRTS* are compiled for a number of platforms as part of different user-created models. When the user code defines a variable of type *RTSequenceOf*, the constructor method for this class is invoked, which makes demand on the system for CPU cycles. The same for the destructor, the assignment operator, etc. There are many factors, which define what this demand would be, and those factors are the parameters to the resource function, characterising the methods of this class. Following is a list of some of those factors, which can be represented as vector *X*, thus the resource function of this software primitive will be *f(X)*:

- The CPU type/make/clock rate - like PowerPC MV2603, 25 MHz

- The type/make/version of the RTOS the user model runs on top of - like VxWorks Tomado 1.0.1
- The make/version of the compiler used to compile the user model
- The command-line options to the compiler, defining the level of code optimisation it would perform during compilation, the level of debug information, etc.
- The version of the ObjecTime's microRTS
- The type of the elements, which would the RTSequenceOf object consist of
- The number of elements, which the new object will contain

As we can see, the vector *X* can be quite large. This is where decomposing of resource functions would help simplifying the corresponding resource functions. Considering the list of parameters above, we can separate them in two sets. In the first set we put the ones, which do not change very much and relatively static - for example, there are finite number of OS-s, each with a finite number of released versions. A new version of a specific OS does not get released very often. Same line of thought for CPU-s type, clock rate, etc. However, the opposite is true for the number of elements of an object of type *RTSequenceOf* - each user model can define variables of this type with a very wide range of sizes. The first set of parameters would be grouped to define an *environment* for a specific combination of their values.

The second set of parameters includes those that vary for different invocations within a single design, and is typically much smaller. For example, the *new()* method has only one parameter - the number of elements to be created (i.e., allocated). In the ObjecTime microRTS, most class methods have a single parameter, while some class methods have no parameters at all.

For example, the set of parameters: target board MV2603 (CPU - PowerPC), with clock rate 25 MHz, running Tomado 1.01, microRTS v5.2, compiled with compiler gnu 2.7.2, compiler option "-O" as command-line parameter, define an environment for which demand data was collected for all the class methods of the microRTS.

5.4. Resource demand data collection for PAMB

As mentioned in chapter 2.3, one of the areas this work concentrates on is the collection and manipulation of CPU demand data. Other resource demands (like I/O requirements, memory management, etc.) are not covered in this work. One major reason for this is that of practicality. All of the determining factors in the above examples are measurable and in a sense deterministic. Thus we can measure and include the resource demands for those factors in a model, at least conceptually. However, such an approach would usually be totally impractical. Why? If we measure and collect resource demands without directly observing the determining factors of those demands, then they will usually appear to be random, i.e. unpredictable. For example, if we are observing disk resource demand, we will determine it will consist of a positioning time and transfer time. The positioning time will be determined (principally) by the distance the arm with the reading head(s) must move and the mechanical speed. However, the initial position of the arm holding the heads and rotating platters is unpredictable without very detailed knowledge and in this sense, we can assume non-deterministic. In a virtual memory system the memory access times will depend heavily on the memory management system, which in turn may depend heavily on the behaviour of the entire multiprogramming set of programs. To achieve determinism in the representation of such factors would be very impractical since the determining factors are so many and in a random relation. Though assuming such resource demands are not predictable, that does not mean we cannot characterize them. There are ways for that, and the most detailed characterisation is that of probability distribution [Sauer81]. However, this is out of the scope of this work, which for simplicity considers only CPU resource demands as more easily determined.

As mentioned in chapter 1 of this document, a set of synthetic applications (called here the *Synthetic Application Suite* or *SAS*) was developed and used in the process of resource demand data collection. The models in SAS do not provide 100% coverage of the existing *microRTS* functionality. Some of the ObjecTime synthetic applications will be changed to implement more test cases. For example, additional test cases will be added to the *PerfComm* model to gather additional resource demand data about some of the communication primitives. Those primitives are currently in design phase and the additional test cases will be determined and implemented at a later stage of this project.

There are three synthetic applications developed as part of this work, which provide gathering of resource demand data for the three main groups of the *microRTS* software primitives:

- the *PerfFrame* model - used to collect data about the *microRTS* primitives, which provide actor incarnation and destruction functionality. The *import*, *importAt* and *destroy* primitives are timed for this test. Variations for these tests include thread setup and the size of actor imported/deported.

For import: In a *for* loop, 1000 actors are imported without index specification. If an actor cannot be imported, we break out and calculate the latency based on the number of successful importations. If there is no problem, we calculate the latency based on 1000 successful importations.

For importAt: In a *for* loop, 1000 actors are imported with index specification. If an actor cannot be imported, we break out and calculate the latency based on the number of successful importations. If there is no problem, we calculate the latency based on 1000 successful importations.

For deport: the *deport* method causes an actor incarnation to be removed from one or more imported aspects. We time the amount of time it takes to deport all actors in a specified actor reference. Since there were 1000 actors imported to be deport, we divide the elapsed time to deport all the actors on that reference by 1000 to calculate the latency.

- the *PerfComm* model - used to collect data about the *microRTS* primitives, which provide communication between actors and between actors and processes;
- the *PerfData* model - used to collect data about the *microRTS* primitives, which provide data class methods. The tests and measurements performed by this synthetic application are concerned with the main methods for each of the data types, like *new*, *delete*, and *copy* - those are considered the ones with highest performance demands.

The synthetic applications output the measurements in a standard ASCII file format, which is read by the ReFCAM tool and transformed and stored into database for use by the PAMB too.

Figure 30 - Process of resource demand data collection

On Figure 30 is depicted the process of resource demand data measurement and collection.

5.4.1. Components and Resource Demand Data

When we look into the whole concept of the *autotest* harness, we cannot overlook the fact that in essence what this harness does is a component-based measurement of performance. Another words, it offers the means for collection of resource demand data for software components. Here by *components* the author has in mind the broader meaning of this term, i.e. that components in general are discrete pieces of code. They can be as small as an API call to the underlying real-time OS or as large as an entire application.

The proliferation of components reflects the surging popularity of component-based development. In component-based development, programmers build new applications by snapping together existing components and objects that work together to perform the function of the application. To ensure that components work with other components within the application, developers build components in compliance with a component object interoperability model, such as CORBA, COM/DCOM, etc. However, the emergence of the Internet and three-tier/multi-tier WEB-based computing has greatly complicated the component-based development picture. Now components work not only on the client-side but provide extensive server-side functionality. Throw in this picture the multi-platform nature of the server-side of an enterprise and there is recipe for trouble, where one of the troubles facing developers is performance.

5.4.2. Synthetic Application Suite

As mentioned previously, one of the goals of this work was to automate the process of gathering demand data for multiple targets. As part of this work a suite of three ObjecTime models was developed, referred here to as the *Synthetic Application Suite*, or *SAS*. Those models exercise the full range of software primitives, as implemented by the ObjecTime virtual machine (Figure 5) and measure the CPU demand of those primitives during run-time. This section presents a brief description of the *SAS*. Currently there are three initial models to measure respectively messaging, frame, and data manipulation operations. Subsequent refinements of the models may include state machine walker tests, external layer tests, modifications of data/messaging/frame tests to accommodate collection of additional resource demand data, etc.

The set of models is used to collect resource demand data for fair estimation of the performance of an ObjecTime model with respect to the following factors:

- *Data tests* - To evaluate the performance of the basic data primitives for ObjecTime data types.
- *Functional tests* - To obtain data, with respect to different RTS services and their usage with different types of data

- *RTS sensitivity* - To evaluate the extent RTS options affect the performance of the functional tests.
- *Compiler sensitivity* - To determine the extent compiler choice affects the executable speed of the model.

5.4.3. Synthetic Application Parameters

The default execution for the synthetic applications is to execute each of the tests once in a given order. At runtime, the user will be able to specify (for each of the performance models) the following parameters for execution:

- **-m** – Memory Activity - If specified, the model will take action to allocate and free memory in order to determine if it has an effect on the tests;
- **-t=<TestName> [TestName2] [TestName3]** -- Tests - Execute only the tests listed. If no value specified, the option is ignored. Values for *<TestName>* will be available in the Performance technical note;
- **-r** – Reverse - If specified, the model will execute the tests in reverse order from the default;
- **-n=<number>** – Number of trials - Will execute the suite *<number>* times and average the results for a more accurate result. Default is 3;
- **-s=<number>** - Specifies the heap size (in KB) the model is to use. Default is 150. This can be also used to control execution time;
- **-v=<value>** -- Value - If *-n* is not specified, this option will be ignored. Otherwise, *<value>* can be one of *minimum*, *maximum*, *average* or *median*. Thus, for *-t=<number>* of executions, the minimum, maximum, average or median will be the result outputted. The default value is *minimum*;

5.4.4. Model Summary

The set of synthetic applications (SAS) contains a total of over 100 actors, 30 data classes and 3 protocols. Each full run produces about 171 data points for that platform and load.

There are three types of tests:

- **Data** - For ROOM and standard (built-in-the-language-of-implementation) data types, we calculate speed of the primitives: new, delete, and copy (if appropriate);

- **Messaging** - We test the communications services. Variations include different thread setups (intra- vs. inter-thread), replications (none vs. 10), synchronization (send vs. invoke) and data sent in message;
- **Frame service** - We test the basic frame service primitives, including *incarnate*, *incarnateAt*, *destroy*, *destroyAt*, as well as *import*, *importAt*, and *deport*. Variations include type of actor (simple vs. complex) and thread setup (intra- vs. inter-thread);

In multithreaded tests both threads have the same priority.

On single-threaded targets multithreaded tests are obviously redundant and their times should be the same as for the corresponding single-threaded tests.

5.4.5. Results of measurement of demand data

This section presents the format of the generated demand data from the SAS models. These models are compiled, loaded and run on the required target platform; during run-time the models perform and output measurements to standard output, which are captured by the test harness and logged into a file for later processing by the ReFCAM tool. Below is a sample of output of "raw" demand data, as produced by one of the synthetic applications:

```

Started test run 11/09/98 15:08:14

Update:      Performance
Actor:       PerformanceTests
Platform:    IRIX6S.r4400-gnu-2.8.1.IRIX6
Target Board: indy
Params:

ObjecTime Micro Run Time System
Release 5.20.C.00
Copyright (c) 1993-1998 ObjecTime Limited


PerformanceTests-v2.4- started:

Number of iterations: 3
Heap size: 153600
Type of average: minimum


Test Case #RTByteBlockSize7: started;
```

```

RTByteBlockSize7: iteration 1 of 3

RTByteBlockSize7: iteration 2 of 3

RTByteBlockSize7: iteration 3 of 3

RTByteBlockSize7:  new:  4.6571 delete:  4.51299

RTByteBlockSize7:  copy:  3.97835

Test Case #RTByteBlockSize7: PASS!

```

This data is read in by the ReFCAM tool afterwards, transformed into data Perl objects, and stored in a database for later use by the PAMB tool.

5.5. Resource Demand Data Management

In the process of developing ways of collecting and storing resource demand data for the different target platforms, which the ObjecTime *microRTS* kernel runs on, one point became very clear – the need for managing the collected resource data. Currently ObjecTime's *microRTS* kernel runs on more than 60 platforms. The amount and diversity of resource demand data is such that it was very easy to loose any grasp of what data and where these data are available/located.

Figure 31 – Front panel of ReFCAM

This created the idea for a Resource Function Database Management Utility (*ReFCAM*), which is to help with the overall management of the collected resource information. This section presents this tool, its overall structure and functionality. On Figure 31 is shown the front end of the *ReFCAM* tool. The

developed GUI is simple and gives the basic concepts about what a resource function management utility should take in consideration.

The following sections describe in details the different options presented by the GUI and the ideas and concepts behind them.

5.5.1. Resource entries list-boxes

The front panel contains two list-boxes – the left-hand list-box presents a list of all currently loaded resource environments (by resource environment is meant the combination between real-time OS, compiler used, CPU characteristics of the used machine, etc.). Each entry is a brief description of such an environment. For a full description and manipulation (i.e., to edit/create new) of environment, a mouse double-click will bring up a separate window with information about the chosen resource environment. This window is not presented here, since at the time of writing this paper it is still under construction and not being finalised yet.

Below each list box there is a set of push buttons. Those are used to perform simple edit operations (like *Edit*, *Delete*, etc.) on the selected items in the corresponding list-box.

5.5.2. The Resource Environment window

A double-click on a *Resource Environment* entry in the *Resource Environments* list-box will bring up a pop-up window (see Figure 32). The pop-up window contains the full information about the clicked environment. By *environment* here the author means the independent parameters, used to define a resource function. Those parameters have been selected and grouped together to create the environment in which resource function is defined.

Through this window the user has the option to view and eventually edit the resource environment information. The information includes the values of all the independent parameters, which define this environment. Those are as follows:

- *Generated* - the date when a test run was performed and resource demand data collected;
- *Platform* - the platform this data corresponds to;
- *CPU type* - the make/type of CPU;
- *CPU speed* - the speed (in MHz) of the used CPU;
- *Compiler* - the name/vendor of the used compiler;
- *Version* - the version of the used compiler;

- *Flags* - any compiler flags, used during compilation;
- *Heap* - size of heap, used by the synthetic application(s);
- *uRTS* - the version of the ObjecTime's microRTS;

Resource Environment Details

Generated: 25-10-1997

Platform: Tornado Compiler: cygnus

CPU type: ppc603 Version: 2.6 under OS: SUN5

CPU speed [MHz]: Flags:

Heap [kB]: 150

Selection policy: ResFuncSelPolicy Measure units: 1e-06 uRTS: Release 5.20.B

Comment: Generated for PAMB testing purposes

Cancel OK

Figure 32 – The *Resource Environment* window

- *Selection policy* - the name of the selection policy used in the selection of resource functions;
- *Measure units* - a constant, used to multiply the resource demand with;
- *Comment* - a free-format text, adding some designer's comments. It is not considered as an independent parameter and does not define the environment.

The idea of having the choice to pick another selection policy is to give the flexibility to implement and use different policies for selecting resource functions. To do so, it will be sufficient to implement additional *Perl* class(s) and when the "Other..." menu option is selected, a pop-up browser window will be shown. In this window the currently available selection policies will be listed and available for selection and assigned to the specific resource environment.

The data can be viewed and edited, if needed, as well as saved back in the database for further usage.

5.5.3. The Resource Function window

A double-click on a *Resource Function* entry in the *Resource Functions* list-box will bring up a pop-up window with information about the chosen resource function. By resource function in this context the author means a resource function with maximum of two independent parameters, defined in the context of the specific resource environment.

Resource Function Details

Key:

Object: Method:

Sub-type:

Resource: ☐ Replicated

Demand [ms]:

Coupling:

Contents:

Size	Demand [us]
1023	113.374
1024	113.374
64	164.93
7	169.711
8	169.271

OK Cancel

Figure 33 – The *Resource Function* window

The picture on Figure 33 above presents this pop-up window, which contains the full information about the chosen (clicked on) environment. The presented on the window fields are as follows:

- *Key* - unique string, which describes the resource function. Generated by the tool.
- *Object* - name of the software component, as known in the *microRTS*.
- *Method* - the specific operation of software component, which this resource function characterises.

- *Sub-type* - some of the ROOM data types (like *RTSequenceOf*) contain data elements, which in turn have type. This field would specify the element type in such case.
- *Resource* - currently *CPU* resource type is the only one supported.
- *Demand* - the demand of this operation (in CPU time). The list-box below contains additional demands, if the resource function has second independent parameter.
- *Coupling* - this field is used in the case of demand data for communication primitives, and describes whether the communication is *interthread* (inside the same process) or *interthread* (i.e., inter-process).
- *Replicated* - this field is used in the case of demand data for communication primitives, when data is being sent to more than one (broadcast) recipient.
- *Contents* - this depicts additional data in regard to the actors in an ObjecTime model.

The data on this window can be edited with the exception of the *Key* field, which is system generated and read-only. Using the *Save* button then the edited data can be saved back in the database.

5.5.4. The OK and Cancel buttons

The *OK* and *Cancel* buttons in the middle of the screen are the main buttons to exit the ReFCAM tool. The *Cancel* button will close the tool and all changes to the currently loaded database will be discarded. The *OK* button will close the tool and commit all changes to the currently loaded database.

5.5.5. The File menu

The *File* menu option contains a number of entries, some of which are to be finalised yet. This section will list and describe the main ones and leave out the ones not considered important. At the moment of writing this chapter, the contents of the *File* menu has not been finalised yet.

Open resource database

Currently the ReFCAM uses simple ASCII files to store resource data. This was a decision, partially enforced by some limitations of the implementation language Perl to interact with more sophisticated database types. Another reason was to simplify the design of the ReFCAM in this first prove-of-concept implementation. The *File* menu option contains the entry *Open Resource DB*, used to open and load a database file with data about resource environments and their corresponding resource function data. This entry pops up a directory browser window, used to search for, point to, and load a database for manipulation in the RFBMU tool.

Merge with a resource database

The **File** menu option contains the entry *Merge with Resource DB*, used to find and load a database file with data about resource environments. This database file will be merged with the currently open database file and the result can be saved. This was implemented to make the manipulation and handling of database files easier.

Save a resource database

The **File** menu option contains the entry *Save Resource DB*, used to save back database information into its original source file. If no changes have been made to the loaded in the ReFCAM database, this action has no effect. Currently the used database format for storing resource demand data is flat ASCII files. The reason is mostly to simplify the design and avoid using unstable database drivers.

Save As resource database

The **File** menu option contains the entry *Save As*, used to save database information into a different database file. This is a standard menu option, which allows the user to save any changes made to a database file without destroying its original contents.

5.5.6. The *Edit* menu

This menu contains two main menu options: *Resource Environments* and *Resource Functions*. Each contains a sub-menu with entries, which duplicate the buttons under each of the list boxes on the main window.

5.5.7. The *Options* menu

This menu contains three main menu options: *Collect Data*, *Import Data*, and *Import Method*.

***Collect Data* menu option**

This option allows invoking the test harness for producing resource demand data. Using the test harness the user can compile the synthetic applications for specific target platform, load them on this platform and collect the resource demand data for this platform. The ReFCAM tool can use the collected data afterwards.

***Import Data* menu option**

This option allows to specify a log-file, result of test-run of a synthetic application on a specific target platform. This log-file contains the resource demand data, as captured by the ReFCAM tool during the run of the synthetic application. The specified log-file then gets converted into resource database format.

Import Method menu option

This option has been added to allow importing of data stored in log-format different than the one, developed for the ObjecTime synthetic applications. It allows the user to specify a path/name of a script file, which would process the result of test-run of a synthetic application, other than the ObjecTime ones. The specified log-file then gets converted into resource database format.

5.5.8. The Help menu

The ReFCAM application offers context-sensitive help. Currently this means that user can move the mouse pointer over any part of the main window and press *Help* / *F1* buttons. This will bring up a pop-up window with help about the chosen window item.

The *Help* menu offers currently the following five menu options:

- ◆ *On Context* - explains the application's context-sensitive help mechanism;
- ◆ *On Help* - explains the basic principle of using the help mechanism;
- ◆ *On Windows* - explains the application's GUI;
- ◆ *On Keys* - provides a list of all available key short-cuts in the application;
- ◆ *About* - pops up a dialog box with application's version number, etc.

The information provided by each of this menu options is stored in flat ASCII files and read in on demand.

6. Resource Function Retrieval Mechanism and PAMB

This chapter introduces some general concepts about resource functions and ways of retrieving them for use in the process of performance modelling. The interface between the resource manager and the PAMB is described here as well.

6.1. Introduction

An essential part of the component's data attributes is the software component's resource function. Resource functions, discussed by Vetland and Woodside, [Vetland96], model the relationship between a software component's resource demand and the parameters, which influence that demand. Most often, the resource demand refers to the CPU time consumed by a software component. However, here we can consider as well the access time for read/write of a disk, etc.

Resource functions fit as sub-models into the overall performance model by providing estimates for parameters like task entry execution time. The execution time of a task entry will often depend on a number of component-specific parameters. In a *copy* method, for instance, the CPU demand will depend on the size of the memory block to be copied.

Resource functions describe how the resource demand changes as the values of the independent parameters vary.

They are particularly useful therefore when examining a performance model's behaviour as the variables, which influence the resource demand of that model's sub-components, change. If each task entry is a component, the entry execution times can be easily estimated using the component's resource function.

There are four resource function research issues that have been identified.

1. What form and format should the resource function be stored in?
2. How can the details associated with resource functions be identified? For example, how are parameter names associated with parameter values?
3. How to determine a *good enough* fidelity is encountered when attempting to build a resource function? In this context, detail refers to the number of independent parameters considered, the various levels of parameters at which the measurements are made, and the number of measurements.

4. How can the user involvement in selecting and invoking a resource function be minimized?

Those issues are discussed here and some concrete and some to-be-implemented solutions are suggested.

6.2. Form and Format of a Resource Function

There are two design goals for a resource function:

- To allow a resource function to be as generally defined as possible, so the user has the option to construct a resource function as a local function call;
- To provide a standard interface to the output of the resource function

Both goals are well achieved by using `Perl` as the programming language for PAMB and the resource manager. `Perl` provides object-oriented and very powerful string-manipulating capabilities.

A mechanism is needed to associate attributes of a resource function with values, where an attribute is a tuple that consists of an attribute name and an attribute value. The attribute values are the parameters of the resource function. There can also be instances where extra attributes can be supplied as parameters to a resource function but those attributes are ignored. Ignoring attribute values should not be considered to be an error.

6.3. Building a Resource Function

Since a resource function is itself a model, the same problem of determining a *good enough* fidelity is encountered during building a resource function. Assuming there is an underlying true function of the component's resource demand, the accuracy of a resource function is defined as the percentage difference between its predictions and the values of the true function.

The question of good enough fidelity in a resource function then reduces to a question of:

- How many measurements of the resource demand should be made?
- At what levels of the independent parameters should they be made?
- How the collected data should be modelled so that the resulting function is within a specified accuracy?

The accuracy of a resource function directly impacts the parametric fidelity of the performance model in which it is used. As such, part of building of the performance model with a *good enough* fidelity involves constructing resource functions, which satisfy certain accuracy requirements. Therefore, a technique is

needed, which can build a resource function to fit within a specified accuracy, determined by the accuracy requirement of the performance model, in which it is used. The fidelity of the resource function will be considered *good enough* if its accuracy falls within that limit.

6.4. Selecting and Using Resource Functions

The selection and use of resource function must be automated, or be automatable. Otherwise, it will become too labour intensive for general use. Figure 26 shows the implemented class hierarchy, which helps in that it organises the resource data, so it can be easily accessible. In this work the author is using as basis some of the earlier concepts, built into the PAMB tool (see section 3.3). However, there were some deficiencies in the way resource functions were selected in the earlier versions of PAMB. An important part of this work is to simplify and improve the mechanisms for selecting and using resource functions in the PAMB tool.

Three concepts are the cornerstone of the new design for selecting resource functions. The first concept is the abstraction of a *selection policy object* (i.e., SPO). A SPO selects one object from a set of objects, according to a given algorithm that is provided with other attribute information.

1 - User picks the proper database with resource demand data

2 - In PAMB the resource function retrieval mechanism checks the existence/sanity of the chosen resource database

1 - User maps the resource environment(s) to resource(s)

2 - PAMB creates a Perl object for each resource environment in the database

Figure 34 a, b – Selection of resource environment(s)

The second concept is the abstraction of a *resource environment object* (i.e., REO). A REO *contains* (i.e., points to) all resource functions, which are relevant to the specific resource environment, described by this REO (see Figure 34).

The last concept is the step-by-step process of obtaining resource demand object for an activity object. Currently this process goes through two steps, which are illustrated on Figure 34.

The intended result here is to minimise/automate the user input in the process of choosing resource demand data to be used in the model building.

6.4.1. Selecting Resource Environment(s)

The concept of *resource environment* was introduced to help in the hierarchical organisation of the collected resource demand data.

This is the point, which Figure 26 highlights - to show that all decisions about picking the right resource environment will be made by the *ResEnvMgr* class. This class will implement methods to perform fuzzy interpolation for the decision making process. For instance, what if there are data for two environments with attributes $\{B, D, N\}$ and $\{C, D, M\}$, but the searched resource function(s) is for an environment with attributes $\{A, B, C\}$? Which environment, if any, can be picked? Is there need to interpolate the measured values from the environment (e.g., the processor speed of the measured values was 200 [MHz] but the speed of the target processor is 1 [GHz])?

For simplifying the implementation, the *ResEnvMgr* (see Figure 26) will pick an environment, which exactly matches the specified parameters.

6.4.2. Selecting Resource Functions

Looking at Figure 26, it becomes obvious that all decisions about picking the right resource function from specific (already chosen) resource environment will be made by the *ResFuncMgr* class.

For instance, what if for specific resource environment there exist in the database two measurements for values $(x=98, y=30.0)$ and $(x=879, y=1500)$, but there is a request for a y value for $x=512$? In such a case, the *ResFuncMgr* class will have to perform the necessary interpolations, utilizing power splines, and pick the closest match (if any).

The final deliverable of the resource function construction process is a mathematical model, which compactly describes the behavior of the result variable with respect to the parameters. Currently the author uses *cubic splines* [Press88] to handle the problem identified here.

7. Tutorial Example

In this chapter we present an example to illustrate using the end-to-end process of building a model with resource functions and to show the ideas presented in this work.

Here we will present the methodology of use, show the generated LQN model with the resource demand data, as constructed by the PAMB tool.

7.1. Overview

The tutorial example here shows the steps, which a software designer will perform in order to create a performance model of the system currently under design.

For the tutorial purposes of this chapter a very simple ObjecTime model was developed, representing a classical client/server system. It contains four actors. The overall structure of the model is shown on Figure 33 below.

7.2. The Case Components

Figure 35 shows the structure diagram of the client/server ObjecTime model, developed for the purposes of testing the concepts for performance modelling described in this work.

Figure 35 - The client/server structure diagram

There are four actors in the model - client, server, a wrapper actor, and a test harness actor, which contains the whole system (i.e., the *Wrapper* actor) and which is not shown here. The *wrapper* is the test actor that executes the test. The reason it is needed is because the outermost actor injects the initial message for the software dye, but it may not receive a reply back. It represents the test driver of the message-sequence chart verification tool and is required by the PAMB tool. For example, the test wrapper does not receive the reply message but it starts the test by injecting the software dye. There is an option to use a different test driver actor name in the *filter.pm* part of PAMB. Examples of possible views include: client with server, clients with client manager, and servers with server manager.

7.2.1. The Server

The server actor's behaviour diagram is shown on Figure 36 below. The server waits for a request on a port; the moment a request comes in, it "executes" it and returns the result/response back to the client.

Figure 36 - The server behaviour diagram

The transition *initSrvr* initialises the server actor, i.e. in this transition any variables, etc. are being initialised. The transition *serveMe* gets triggered when a request for service comes from the client actor. The code in this transition can be seen on Figure 34 above. The transition *release* is triggered by a message from a client, when a specific client has finished work and sends a notify message to the server about that. Normally it would be used for clean-up purposes. For simplification the *release* transition contains no code.

In the current versions of ObjecTime Developer angio-traces are generated only for communication primitives, like *send*, *reply*, etc. For the purposes of this work the *busy* statement has been used to simulate the use of other software primitives and angio-traces being generated for them. In this case the *busy* statement plays the same role as the *printf* statement in debugging code, i.e. it just outputs whatever string is passed to it into the scenario's angio-trace file.

7.2.2. The Client

The client actor's behaviour diagram is shown on Figure 37 next page. The client waits for a *go* signal from the *ServerManager* actor, after which sends a request to the server; when the processed request comes back, the client accepts it and stops execution.

Figure 37 - The client behaviour diagram

The transition *initClnr* initialises the client actor, i.e. in this transition any variables, etc. are being initialised. The transition *sendReqst* is triggered by a signal from the test harness (the Wrapper actor), which signal here is manually injected. The RPL code (a subset of *Smalltalk*) in this transition follows below:

```
BUSY 'new RTSequenceOf RTCharacter 100' ENDBUSY.
```

```
SEND serverPort SIGNAL %serveMe ENSEND.
```

```
BUSY 'delete RTSequenceOf RTCharacter 100' ENDBUSY.
```

In the current versions of *ObjectTime Developer* angio-traces are generated only for communication primitives, like *send*, *reply*, etc. For the purposes of this work the *BUSY* statement has been used to simulate the use of other software primitives and angio-traces being generated for them.

The *done* transition is fired when the answer from the server comes in. It processes the answer, cleans up the environment (i.e., any variables, heap, etc.) and exits.

7.2.3. The Scenario

For the purposes of this tutorial, only one scenario has been chosen and generated. The following Figure 36 depicts the angio-trace file, as generated for that specific scenario for the client/server model by the *ObjectTime Developer* tool.

```

thread testWrapper.1 {
    1401 Wrapper:wrapper(2-1) state S1;
    1401 Wrapper:wrapper(2-1) receive %goWrapper;
    1402 Client:client(3-1) state Ready;
    1402 Client:client(3-1) receive %startClnr;
    1402 Client:client(3-1) transition sendReqst action 'new RTSequenceOf
RTCharacter 100';
    1402 Client:client(3-1) send %serveMe delay 1;

```

```

1402 Client:client(3-1) transition sendReqst action 'delete RTSequenceOf
RTCharacter 100';

1403 Server:server(4-1) state Ready;

1403 Server:server(4-1) receive %serveMe;

1403 Server:server(4-1) transition serveMe action 'new RTInteger none 1';

1403 Server:server(4-1) transition serveMe action 'delete RTInteger none 1';

1403 Server:server(4-1) reply %done delay 1;

1404 Client:client(3-1) receive %done;

1404 Client:client(3-1) transition done action 'new RTInteger none 1';

1404 Client:client(3-1) send %releaseClient delay 1;

1404 Client:client(3-1) transition done action 'delete RTInteger none 1';

1404 Client:client(3-1) state Done;

1405 Server:server(4-1) receive %releaseClient;

1405 Server:server(4-1) state Ready;

};

```

Figure 38 - Angio-trace file for single scenario

We can see on Figure 38 the output from the *busy* statements, used in the model. It is represented by the key word *action* followed by a string, containing the operation description and parameters.

7.3. The steps

This section describes briefly each of the steps (see Figure 9) in building a performance model with the PAMB tool. It assumes the initial few steps (selecting scenario(s) and generation of trace output for those) have been performed already.

Step 1: Convert angio-trace to sub-model

Here the designer selects an angio-trace file and PAMB abstracts the trace(s) into a structured graph, which provides a higher-level abstraction for the analysis. During this step PAMB detects occurrences of actor interactions and entries by matching sub-graph templates with the structured graph. This gives a partial workload model for the given scenario, called a *sub-model*. In our case, the angio-trace scenario chosen is presented on Figure 38 above.

The step can be repeated for a number of angio-trace files and the corresponding sub-models will be produced. In this step I used only one scenario and obtained only one sub-model.

Step 5: Define environment

In this step the model environment will be specified. The functionality of this window (see Figure 16) is currently not fully implemented. However, here the user will be able to specify hardware-software relations, i.e. things like which actor (e.g. task) on which CPU is to be run, etc. The top part of the window allows the addition/deletion of CPU-s, and the corresponding OS running on each CPU.

The bottom part of the window allows the mapping of the actors (tasks, in the broader sense) in the ObjectTime model to the CPU-s in the system.

Step 6: Build configuration file

A model configuration file with data about each sub-model and the whole environment is built during this step. The default name for the configuration file was chosen here and the file built.

Step 7: Build experiment file

During this step experiment LQN file with the results specified for this experiment is build.

Figure 40 - LQN load specification

The LQN model contains the data from all the specified sub-models. In this case, only one sub-model was built used in the construction of the LQN model.

Step 8: Assign parameters

In this step a range of parameters for the *\$load* variable is specified to be evaluated. This range indicates the arrival rate (i.e., customers per second) for which experiment will be conducted.

The *\$load* variable gives the event-generation speed, i.e. it represents the number of events per second in the system under experiment. This is true when we have *Open Load* selected in the PAMB during step #3 *Select Workload* in the PAMB modelling process. In the case of *Closed Load* selected in the PAMB during the step *Select Workload*, the *\$load* variable hides more information. It represents the number of events being sent to the system during every specific time interval; i.e. it represents the relation *events/time* in this case.

In this case the parameters chosen to carry out this experiment were 10, 20, 40, 60, 80, and 100 arrivals per second.

Step 9: Solve experiment file

In this step the *Specification Experimenter (Spex)* is invoked, which uses as input the built configuration and LQN-files to perform the experiment and output the results.

Step 10: Plot the experiment results

In this step PAMB uses the *gnuplot* utility to output in graphical format the results from the conducted experiment. The results from this tutorial example are shown in the next section.

In PAMB we use simple characterizations for resource demands and their representation. The most important of these characterizations is the *mean* or *expected value*, which corresponds to the average value in less formal terminology. If we have x values, then $P(x)$ would be the possibility of value x .

Informally we may define possibility as relative frequency. Of course, we must have $0 \leq P(x) \leq 1$ for all x and $\sum P(x) = 1$. The mean then is $E(x)$ and defined by:

$$E(x) = \sum xP(x)$$

In the LQN, if we assume that the flow of tasks out of a queue (for a resource) is equal to the flow into the queue, we call the flow the *throughput*. If we call utilization U the fraction of time a resource is busy (in our case, the CPU) then the throughput must be:

$$\text{throughput} = U / E(x).$$

The graphical representation in PAMB of those characterizations can be seen on the diagrams in the next section of this work.

7.4. The experiment results

The performance predictions are average values that the software developers easily understand, such as device utilization, task utilization (percentage of time the task was providing a service), waiting time at a task, waiting time to get access to a device, or the rate at which the system can execute the scenarios.

7.4.1. Single CPU system

The following diagrams (see Figure 41 a, b, c, d) represent the results of the experiments, performed with PAMB on the client/server model, running on a single (non-distributed) CPU.

Figure 41 a, b, c - Experimental results for non-distributed system: task/CPU utilization

In the case of *Closed Load* selected in the PAMB during the step *Select Workload*, the *\$load* variable hides more information. It represents the number of events being sent to the system during every specific time interval; i.e. it represents the relation *events/time* in this case.

Figure 41 d - Experiment results for non-distributed system: waiting time [s]

As we can see from Figure 41 b, the CPU utilization is about 45% at full workload. This is well below the system capabilities. A reason for such under-utilization is the simplicity of the system used here. It does not perform any substantial amount of work, thus the CPU usage is very low. Because of the same reason, the waiting time, which is the delay a subscriber experiences in the initial moment of interacting with the system, is quite small (about 2 ms for total workload) as well.

7.4.2. Two CPU system

The above diagrams (see Figure 42 a, b, c, d) represent the results of the experiments, performed with PAMB on the client/server model, running (distributed) on two CPU system.

Figure 42 a, b, c - Experiment results for distributed 2-CPU system: task/CPU utilization

As we can see from Figure 42 b, the default CPU utilization is drops to about 10% at full workload. This is expected, since we introduced additional CPU in the system. As we can see from Figure 42 d, the utilization of the second CPU is somewhat higher (about 30% at full workload), which is due to the fact that this CPU runs all the client tasks.

If we compare Figure 41 d and Figure 42 e, we can see that the waiting time for a distributed system increases ever so slightly.

Figure 42 d, e - Experiment results for distributed 2-CPU system: waiting time [s]

Because of the simplicity of the system there are no considerable differences between the corresponding diagrams on Figure 41 and 42. For example, we can see that utilization of the default CPU is better in the single-CPU system, while in the distributed system the CPU-cycles are not being utilised fully (see Figure 41-b and Figure 42-b).

7.5. Impact of Execution Time Estimates on the Performance Model

In this case a hypothetical fast CPU type (two times faster than the one in the scenario described in section 7.4.1) was chosen and demand data for it was created in the demand database of ReFCAM.

Figure 43 a, b present the CPU utilisation and waiting time in this case.

Figure 43 a, b - Experiment results for single-fast-CPU system

If we then compare the utilization and waiting time for this fast CPU with the corresponding diagrams on Figure 41 and Figure 42, we can see the differences easily.

For example, the CPU utilization on Figure 41-b for total workload is about 40%, while from Figure 43-a we see that the CPU utilization drops down to 27%, i.e. the faster CPU is under-utilised compared to the slower one. The waiting time on Figure 43-b is about 50% lower for total workload than the one on the corresponding Figure 41-c, as well.

8. Case Study - the GCS application

This chapter presents a case study of a Group Communication Server (or simply the *GCS*) system, developed originally in Carleton University and used to simulate a real-time, (and potentially distributed) application. It has been chosen for case study in this work as a typical real-time application, in which the driving scenarios are inherently concurrent, and the events initiating use cases come from multiple sources, often highly independent and uncoordinated.

The chapter first presents a brief overview of the GCS system, after which and the results from the PAMB performance modelling, done on the application.

8.1. Overview

The chosen GCS application was originally created as a course project by C. Scratchley and H. Yang using design patterns to design concurrent communication software. GCS exists to distribute notifications to a group of subscribers. Any numbers of users can subscribe/unsubscribe to the services of the GCS system. There are several subscribers' groups based on common interests. For the sake of simplicity, we assume that the number of groups is pre-defined (three group types in our case) and does not change. A subscriber may belong to a single group at a given time. He/she must choose a group when subscribing, or can ask the server for a group change during his/her subscription period.

Subscribers can submit two types of items to the server: messages and documents. An item (message or document) can be either public (for the entire community) or private (for the submitter's group). Once submitted, an item will not change its class (public or private), nor the group it was intended for.

Messages and documents are processed differently by the communication server:

- The communication server broadcasts messages either to the entire subscriber community (public messages) or to the submitter's group (private messages). After that, messages are archived, and can be further viewed by current subscribers upon request.
- A new subscriber can subscribe to a specific group and submit/request documents, or send public/private messages.
- Submitted documents can be updated/modified by other subscribers upon request (if they have the appropriate privileges). Depending on the document's class (public or private) the server will notify the intended recipients when it receives a new document or a new update of an older document. The server keeps a history of the updates for each document consisting of a limited number of the most recent versions. It is up to a subscriber to retrieve a document for

reading or updating. Several readers are allowed to access a document at the same time, but only one subscriber can modify it at once.

Each subscriber has a set of access rights (no access, read only, update, etc.) with respect to different classes of items (public, private). For simplicity reasons, these access rights are fixed during the subscription period of a client.

Figure 44 - High-level UCM view of the GCS application for client subscription, request, and save document scenarios

For the purposes of this work their design was re-implemented as an ObjecTime model and used as a typical example of a real-time system: highly reactive, driven by unpredictable ordering and timing of events.

This ObjecTime model simulates concurrency by switching among different paths in the software. Every thread (see the use-case map on Figure 44) potentially could connect number of processes.

Figure 44 above presents the overall functional view of the *Group Communication Server* system applying the concept of use-case maps as suggested in the [Buhr96] work. It shows only the main threads of execution in the application. The figure depicts in a high-level the three main scenarios, considered in this section - the client subscription, client requesting a document, and submitting this document back to the database. The start point is the black dot, after which a new subscriber is incarnated by the system and placed in a pool of subscribers. The newly created subscriber then

subscribes to a specific group and is imported into the corresponding *ServerMgr* object, responsible for handling requests to/from this group of subscribers. The subscribers then makes a request for a document to its server manager, which in turn submits it to the *Document_Map* object. The *Document_Map* object fetches it from the database and sends it back to the server manager, from which the requests came. The server manager in its turn passes the document to the subscriber, which requested it in the first place.

8.2. The Software Design

The Group Communication Server (GCS) system has the overall structure presented on Figure 45 below. The figure displays the structure diagram for the *ServerManager* actor, which is the top actor, embedding all the major components of the GCS application. On Figure 44 we can see the main actors representing real objects in the problem domain. There are a number of *Client(subscriber)* actors, *Server*, *ServerManager*, a *Document_Map* actor and database *DbDiskX* actors.

Figure 45 - The GCS system - Structure diagram of GCSsystem actor

As mentioned in chapter 7, due to design restrictions in the PAMB tool the outermost actor should be called *wrapper* and is the test actor that executes the test. It is needed because the outermost actor injects the initial message for the software dye, but it may not receive a reply back. The PAMB tool requires the actor as entry point in an angio-trace. Here the top *GCSsystem* actor is embedded in such an actor called *Wrapper* - however the actor is not shown on Figure 45.

The next sections describe in more detail the system functionality through presenting the main use cases of the application. The main actors in the system are described as well.

Performance Requirements to the GCS system

While designing a system like the GCS system there is some performance requirements to consider in regard to the overall architecture. Too many concurrent processes will introduce excessive overhead for process switching, communication, and data protection while too few may introduce important delays because of non-urgent tasks being done before important ones can begin, or due to blocking.

Assuming exponential delays in the response times of the system, we can use the mean response time to set some concrete performance requirements for the GCS system. The author has chosen to use the mean: $(1 - e^n)$ in defining the performance requirements to the system. The mean is 65%, two times the mean is 85%, and we have chosen in this case to use three times the mean, which is about 95%.

We set here some hard requirements for the system, as well. In defining them, we come mostly from some general experience and common sense. For CPU and task utilization we specify about 90%, i.e. we require slight under-utilization in the system. The waiting time for a subscriber we set to be no more than 0.1 seconds for full system workload. The service time for the GCS system we set to be no longer than 0.4 seconds for full system workload.

The design should ensure that 95% of the messages in the system are delivered in acceptable time limit and there are no lost messages/requests.

The GCS_Client actor

The GCS_Client actor is a subclass of the Client class. There are three subclasses of this actor class - *GCS_Client1*, *GCS_Client2*, and *GCS_Client3*. Each one of them imitates a subscriber, belonging to a specific subscriber group. There are four methods in this class, used to allow a client to send request to the server to *subscribe*, *unSubscribe*, *getDocument* with given name, or *putDocument* with given name.

The ServerManager actor

The *ServerManager* actor is the one that contains a server object of specific subscription group type. It provides the main functionality for handling requests from subscribers in a specific group.

There are three objects of the *ServerManager* class, each attending to different subscriber group. When a request comes in from a client/subscriber in this group, the corresponding *ServerManager* object will direct the call to a server object (see Figure 46-a), responsible for processing the request.

Figure 46-a - *ServerManager* actor - Structure Diagram

The server object then will use the *Document_Info* object to access the specific document in the database. Figure 46-b depicts the behaviour diagram of the *ServerManager* actor, which behaviour is very straightforward - it reacts to requests coming from the server object that serves a specific group of subscribers; as well as to requests coming from the GCS system.

Figure 46-b - *ServerManager* actor - Behaviour Diagram

There are separate transitions for each request that might come in - for getting a document from the database, storing it back in, as well as subscription/un-subscription of clients, and processing of messages to subscribers in the system.

The ServerGroupX actor

The GCS system defines a *Server* class, which has three classes derived from - *ServerGroup1* through *ServerGroup3*. Each of them is designated to serve subscribers of specific subscriber group. Figure 47 below depicts the behaviour diagram of the *Server* actor.

Figure 47 - Server actor - Behaviour Diagram

Each *ServerManager* object at initialization time would create a *ServerGroupX* object, where *X* is the subscriber group type. Any requests from clients in this group are to be handled by this server object.

The Document_Map actor

By creating a static object of type *Document_Map*, the GCS system keeps a map of the documents submitted by clients. The servers of each subscriber group can search the information of documents through this map.

The communication server will archive the submitted items in a database. Old documents are kept for a limited period of time, then discarded. The server provides upon request an index of the available items that were submitted/updated during a specified time period.

The communication server is able to accept and process subscriber requests concurrently, and to provide a fast feedback to the users.

The database server that archives the items must allow for concurrent processing of more than one request at a time. For example, while a database request is waiting for a disk I/O, another request may be processed by the database management system running on the CPU. This is one reason the model implements as separate actors the database disk(s).

Figure 48 - GCS system - Database Class Hierarchy

As we can see on Figure 48 every instance of *Document_Info* has a list of its own subscribers. Once the document is updated the server notifies all the subscribers.

If we define a real-time system as a system that reacts to stimulus producing consistent results in a timely fashion, then the demonstrated Group Communication Server system is a good example of a real-time (and potentially distributed) system, which exhibits the following characteristics:

- *Concurrent* - multiple requests from multiple users come in unpredictable in time and requires concurrent execution.
- *Event-driven* - stimuli typically random ordered events.
- *Reactive* - the system responds to stimuli from the outside world.
- *Distributed* - the system's processing units run on different CPU-s.

The implementation of the *Group Communication Server* system consists (see Figure 49) of 3 server manager (*ServerManager* class) actors, which are dynamically created by the behaviour of the *GCSys* actor at initialization time. Each server manager is incarnated and assigned a particular server type (*ServerType1* through *ServerType3*), for which it is responsible.

The server manager during its initialization incarnates a server actor of the corresponding group within its decomposition.

request to the *Document_Map* object, which performs the necessary database operations and returns the requested document to the server manager. The server manager object of the subscriber group then passes the document to its server object, which takes care to pass it to the subscriber who requested it. The client simulates some action upon the document (by some random delay), after which sends it back (stores it) into the database. The client might (based currently on random number generator) send a public or group message to the server to be broadcasted. At the end the client/subscriber requests to be released through the server connection. The server passes this request to its server manager, which then deports (i.e., un-subscribes) the client, and informs the GCS system of this. When this is completed, the GCS system destroys the client actor.

Figure 49 presents the overall class diagram of the GCS application (it omits some of the minor classes and actors), as implemented by the ObjecTime's GCSysystem model.

For a sample of the run-time output of the GCSysystem model, please see Appendix A.

8.3. The Scenarios

This section describes the performance modelling done using PAMB with the *Group Communication Server* (GCS) system.

There are several main scenarios (use cases) in this application, among which three were chosen to generate angio-trace files for and perform experiments with them.

1. Client makes a request to subscribe to the GCS system, to a specific subscriber's group.
2. Client makes a request to obtain a specific document from the database.
3. Client submits a document to the database.

For each one of those scenarios a separate analysis was done.

8.3.1. Subscribing a client

In this scenario a subscriber makes a request to the GCS system to subscribe to a specific group. The GCS system sends the request for subscription to the server manager, responsible for serving the corresponding subscriber group. Upon receiving the request, the manager sends it to the server of this group, which performs the necessary actions and notifies the new subscriber for the output of the action.

8.3.2. Sending a request for a document

In this scenario, the GCS system serves a subscriber from a random subscription group. This subscriber is attached to server manager, which handles clients of this specific group. The subscriber makes a request to the server manager for subscription to its specific group. After confirmed successful subscription, the new subscriber sends a message of class *group message* to the server. Upon receiving the message, the server sends it to all current subscribers in this group.

8.3.3. Saving a document

In this scenario, the GCS system serves a subscriber from a random subscription group. This subscriber is attached to server manager, which handles clients of this specific group. The subscriber makes a request to the server manager for subscription to its specific group. After confirmed successful subscription, the new subscriber sends a message of class *group message* to the server. Upon receiving the message, the server sends it to all current subscribers in this group.

8.4. Impact of Execution Time Estimates on the Performance Model

The following sections present the results of the evaluations based on scenarios of the three main use cases described in section 8.3 above. For each scenario, the parameter *\$load* gives the rate of arrival of requests per second to the system.

8.4.1. Client subscription scenario.

In this scenario a client makes a request to the GCS system to subscribe to a specific group. The system forwards the request to the corresponding server manager, which takes care to complete the subscription.

Here Figure 50 a, b, c, d, e depicts the performance modelling results, predicted for the GCS system executed on a single *PowerPC* CPU, running Tornado 1.0.1 RTOS.

The diagrams on Figure 50-a, -b, and -c show the CPU utilization and the utilization of some of the involved tasks. The utilization diagrams for the rest of the involved components are not shown here mostly due to space considerations and because they do not carry important information for the purposes of this experiment. The shapes of curves on the utilization diagrams for the rest of the system tasks are very similar to the ones, depicted below.

We can see from those diagrams that at maximum load in the system, the CPU utilization is at about 55%, which is well in the range of the requirements, specified at the beginning of this chapter. The utilization for the rest of the tasks in the GCS system varies in the range of 30% to 50%, which satisfies the requirements as well.

Figure 50 a, b - GCS system - client subscription, CPU/task utilization

One reason for this relatively low utilization is the fact that the performance model does not capture all the demand data (see section 8.5 for more detail).

Figure 50 c - GCS system - client subscription, client task utilization

The diagrams on Figure 50-d, -e below show the waiting and service times for this scenario. By *service time* here is meant the expired time since a request was submitted by a subscriber to the GCS system until the moment response came back to this subscriber.

Figure 50 d - GCS system - client subscription, waiting time [s]

By *waiting time* here is meant the delay a subscriber would experience in the initial moment at interacting with the system.

Figure 50 e - GCS system -client subscription, service time [s]

We can see that at maximum load in the system the service time is well in the specified range (about 110 ms) of requirements for GCS.

8.4.2. Request to obtain a document.

In this scenario a client made a request to obtain a document for editing/viewing purposes. The request is sent to the *Document_Map* object, which handles all the access to the document database, i.e. objects *DbDisk1* and/or *DbDisk2* (see Figure 48).

Here Figure 51-a, b, c, d depicts the performance modelling results, done for the GCS system running on three separate *PowerPC* CPU-s (each model PPC-603), each running Tornado 1.0.1 RTOS.

Figure 51 a - GCS system -
request for a document, service
time [s]

The diagrams on Figure 51-a, -b show the service and waiting times for this scenario. We can see that at maximum load in the system the service time is well in the specified range (about 32 ms) of the requirements.

Figure 51 b - GCS system -
request for a document, waiting
time [s]

The diagram on Figure 51-c shows the utilization of the system CPU. Again, the utilization diagrams for some of the involved components are not shown for this scenario because of space considerations and because they do not carry important information for the purposes of this work. One point to mention here is that they are all very similar in shape.

Figure 51 c - GCS system - request for document, CPU utilization

One interesting point here to see is that the CPU utilization is about 73% at full system workload. Even though this is in the range of specified system requirements, it is a higher utilization than in the other graph. Main reason for this is the fact that in this scenario there are more tasks involved, i.e. the system performs more work thus the CPU utilization increases.

8.4.3. Request to save a document.

In this scenario a client made a request to save a document. After saving it, the corresponding group server sends a notification message to all subscribers in this group.

Here Figure 52 a, b, c, d depicts the performance modelling results, done for the GCS system running on a single *PPC-603* CPU, running Tomado 1.0.1 RTOS.

The diagrams on Figure 52 a, b show the CPU utilization and the utilization of one of the database disk tasks. The rest of the diagrams, showing the utilization of the rest of the components are not shown here because of space considerations and because they do not carry important information for the purposes of this work.

If we compare Figure 51-c and Figure 52-a we can see in both cases the CPU utilization being very similar (about 70% at full workload). This is because these two scenarios are very similar - they involve very much the same number of tasks and the work the system does is similar.

Figure 52 a, b - GCS system -
request to save a document,
CPU/task utilisation

The diagrams on Figure 52-c, -d below show the delay and service times for this scenario. We can see interesting phenomena in this case - that at small workloads the distortion of the experiment points is quite high. This could be due to a number of reasons. Another reason could be additional "noise" being brought in by the method, used by ReFCAM to interpolate collected the demand data. There is some error introduced by the PAMB tool, as well - it is known it analyser has some problems, which have manifested themselves in another occasions.

Figure 52 c, d - GCS system - request to save a document, waiting and service times

8.4.4. Request to obtain a document, distributed system.

This is experiment with the same scenario as in 8.4.3 with the exception that some of the tasks have been distributed to run on separate CPU-s.

Here Figure 53 a, b, c, d depicts the performance modelling results, done for the GCS system running on a 2 and 3 CPU-s (type *PowerPC-603*), each running Tornado 1.0.1 RTOS.

From the diagrams on Figure 51-a and 53-a, we can see how the service times change when the workload is distributed over more CPU-s. For the single CPU system the service time for total workload is about 315 μ s, for the two-CPU system it drops down to about 305 μ s, and for three-CPU system it drops further down to 265 μ s.

Figure 53 a, b - GCS system - request a document, distributed system scenario, service time

The diagrams on Figure 53-c, -d show the utilization of two of the system CPU-s. The other diagrams, showing the utilization of the rest of the components are not shown here because of space considerations and because they do not carry important information for the purposes of this work. The shapes of those other task utilization diagrams are very close to the ones, shown on Figure 53-c, -d.

Figure 53 c, d - GCS system - document request, distributed system scenario, CPU utilization

8.4.5. The three scenarios combined

Each of the three scenarios considered to this moment gave us a different part of the picture about how the GCS system will perform. A combination of the three scenarios would give us the whole, more complete picture.

Currently the PAMB tool does not provide the possibility for automated combining of scenarios. There is some work being done in this direction and a solution is close to be implemented. However, it is possible to combine scenarios manually: for this purpose the angio-traces of the three scenarios have to be merged into one single angio-trace, after which feed the combined angio-trace into the PAMB tool and perform experiments.

8.5. Impact of Design Changes

This section shows how the changes in the considered software systems impact the performance. In the previous design we had the *Document_Map* object interacting directly with the database disks for the purpose of storing/fetching documents. However, such a solution does not offer very much flexibility - a change in the database structure would require the corresponding change in the *Document_Map* object's functionality. A valid designer decision here would be to introduce additional layer on top of the database and leave the *Document_Map* class handle only the flow of incoming and outgoing documents. The newly introduced layer (class *DBmanager*, see Figure 54) would handle the database - on which disk to store a newly submitted document, for example, etc.

This solution is depicted on Figure 54 below. The *DBmanager* object provides the additional layer of functionality for handling the database.

Figure 54 - GCS system - New overall class hierarchy

After the GCS system was changed as described above, additional performance modelling was done to determine the performance impact of this decision on the system.

Figure 55 a, b - GCS system - save a document request, service and waiting times

We can now compare the results, as depicted on Figure 55 above and on Figure 53, which depicts the results for the same scenario in the old system design. The service time for the old system design at maximum load is about 315 [ms], while in the new system there is an increase of about 41% - it is about 448 [ms]. If we take a look now on the waiting times - for the old system at maximum load it is about 660 [μ s], while in the new system there is an increase of about 18% - it is about 770 [μ s]. Both result diagrams provide the same answer to the software designer: there is a substantial performance hit introduced in the new system design through the change to the system architecture.

8.6. Real target experiments.

As mentioned earlier in this document, the generation of angio-traces (necessary in the performance model building process) requires that the designs are developed using ObjecTime's RPL. However, for

the purposes of this work, once the performance model for a software design with demand data for specific platform has been built, testing on this target platform was needed for prove of concept.

8.6.1. Model creation

The original ObjecTime model of the GCS system was implemented using *RPL* (a subset of Smalltalk) for the purposes of generating the angio-trace outputs for the different scenarios. Subsequently, for the purposes of this work the original GCS system was re-implemented as a C++ model (called here GCS++), compiled for the chosen target platform and loaded on it for testing purposes.

Conceptually there are no differences between the GCS++ and GCS models. However, there are some additions to the GCS++ model, which come mostly from adding some code to time the execution times of different sections of the model code. The ObjecTime microRTS implements the *getTime()* function, which was used throughout the model for timing purposes. It is clear that the use of this function imposes additional performance hit and therefore, introduces additional error in the measurements. However, for the purposes of this work this error was not taken in consideration.

8.6.2. Measurements taken

The target platform was a PowerPC board, running Tornado 1.0.1 RTOS. The measurements performed on this target showed about 30% to 35% increase in the waiting and service times, compared to the performance model predictions. A part of the reason for this the author sees in the differences between the simulation and real-target models: the latter one uses some additional function calls (like *getTime()*, to time start and end points in the executed code), which are not accounted for in the simulation model. Another reason for this is the fact that currently demand data is not available for all target RTS functions - like entrance/exit in/from state machine transitions, etc.

Since measurements for full system workload were very difficult to instrument and measure, a simpler approach was taken - multiplying the single-load measurements to obtain the results for maximum load.

Despite the fact that the measurements were not correct and introduced some percentage of error as result, the experiments show that currently about 70% of the total workload are captured by ReFCAM and reflected in the performance model.

9. Conclusions

This chapter summarises the work performed in this thesis and presents the main areas, which would require additional research and work to be done.

The author hopes that this work has given you a taste for PAMB and the suggested new way of collecting resource demand data and managing resource functions. As we have seen, PAMB is designed to be an environment that enables the performance modeling of complex real-time and/or distributed systems. It offers a unique object model, integrated event and cyclic processing, an understandable run-time architecture, and sophisticated modal configuration management. It features superb run-time experimentation and debug. For software designers, developing a complex, real-time (and possibly, distributed) systems, the PAMB tool is a very viable option to do performance modeling.

There is not a lot similar research done, which this work presents. One such research is the work of Steppler and Lott [Steppler97]. It introduces a new tool, named SDL Performance Evaluation Tool (SPEET), for the performance analysis of formally specified systems under real-time conditions. SPEET facilitates the simulation and emulation of several formal specifications at the same time. The systems to be simulated, respectively emulated, can be triggered by traffic load generators and can be interconnected with transmission links, which correspond to physical channels. The user can easily define probes within the formal specifications. The data of these probes generated during simulation runs, respectively emulation runs, can be statistically evaluated. SPEET provides a new CASE oriented solution for product development using formal methods.

9.1. Success and contributions

There are several main points to highlight in this work:

- The described in this work test harness for automatic collection of resource demand data has been successfully employed for the two years in ObjecTime Ltd., in the Product Verification department.
- The Resource Function Collection And Management (ReFCAM) tool provides a very tight integration with the PAMB utility for resource demand data manipulation.
- The ReFCAM tool provides very good capabilities for maintenance and automation of resource demand data.
- The addition of the resource function selecting steps to the PAMB *workflow* process increases the level of understanding and simplifies the use of the tool.

- The developed suite of synthetic applications (SAS) provides easy and quick way for collecting resource demand data from the target platform(s) of choice.
- The presented here work implements a new approach in that it groups resource functions by some of their independent parameters into resource environments. Thus, this new approach makes the practical use of resource functions much easier to implement and comprehensive to use. There is some work done in regard to resource function parameters [Smith90] but not a lot to simplify their use and practical usage.
- More than one (total of four) resource environments were calibrated and demand data collected for.
- A sample C++ model (real-world size) was designed and developed as part of this work. This model was used to evaluate ReFCAM in the context of PAMB (see chapter 8).
- The testing with the C++ model on a real-time target platform shows that about 70% of the workload is captured by the performance model.
- This work includes a short tutorial on PAMB and ReFCAM, and the separate steps in the PAMB workflow process.

One of the contributions of this work is the generality of the method developed here. For the purposes of the work the author used ObjecTime's *Developer* and its capability to generate angio-traces. However, any design environment with capabilities for runtime trace-generation can be used with the tools developed here. The ReFCAM utility has the capability to plug-in different drivers, which would process the output from synthetic applications, developed and run with other tools. There are a number of design tools on the market, offering similar features of those provided by ObjecTime's *Developer* - for example, *StateMate*® MAGNUM™ from *iLogix*, to name one. *StateMate*® MAGNUM™ is a graphical simulation and software synthesis tool used for the rapid development of embedded systems. Operating on an engineering workstation or PC, *StateMate* creates a visual specification, an executable model of the system, that clearly and precisely represents the intended functions and behaviour of the system being specified. Provided that a synthetic application suite is developed to capture the performance demands of *StateMate* software model, the performance data can be imported by the ReFCAM tool and stored for use by PAMB.

In the literature there are a number of works, which present different ways for constructing and handling multi-dimensional resource functions. However, the author in his research has not found anything similar to the suggested in this work approach.

9.2. Limitations

The work has done some research in the area of resource functions. However, this is a very large research area and there is need for additional studies to be done in the area of complexity of resource functions and the impact on creation and management of resource functions.

The testing with the C++ model on a real-time target platform showed about 30% error. That means about 70% of the workload is captured by the performance model. This is mainly due to the fact that currently there are no measurements taken for some activities in the microRTS - like critical sections, entry/exit in/from state machine transitions, etc. Further enhancement of the SAS set of synthetic applications would allow for capturing higher percentage of the workload. However, even in the current state the tool provides good picture about the potential performance hits in a software system. There are (even commercial) systems, which allow for error in their estimates in the range of 150% to 200% error. The nature of performance predictions is such that it allows for such wide range of error - as far as it predicts the exact place for performance bottlenecks in a system.

Model combining multiple scenarios is a natural extension of PAMB, not yet available. Further work in this direction is needed.

Appendix A

This appendix displays excerpts from the output of the *Group Communication Server* system, which is used in chapter 8 as a case study for this work.

```
0      System

505    server (16-1) ServerGroup1
      Subscribing client of 'Group1'

1105   server (18-1) ServerGroup3
      Subscribing client of 'Group3'

1305   server (16-1) ServerGroup1
      Subscribing client of 'Group1'

1606   clients (19-1) GCS_ClientGroup1
      Sent command to GCS type1 server: g u myDoc1.doc

1906   clients (21-1) GCS_ClientGroup3
      Sent command to GCS type3 server: g u myDoc3.doc

1907   server (18-1) ServerGroup3
      Sending to Doc_Map request: g u myDoc3.doc

2005   server (17-1) ServerGroup2
      Subscribing client of 'Group2'

2305   server (18-1) ServerGroup3
      Subscribing client of 'Group3'

-----

3806   clients (25-1) GCS_ClientGroup3
      Sent command to GCS type3 server: p u myDoc3.doc

4105   server (17-1) ServerGroup2
      Subscribing client of 'Group2'

4205   server (16-1) ServerGroup1
      Sending to Doc_Map request: p u myDoc1.doc

4205   server (16-1) ServerGroup1
      Subscribing client of 'Group1'

4505   server (16-1) ServerGroup1
```

```

        Subscribing client of 'Group1'
4506 clients (31-1) GCS_ClientGroup2
        Sent command to GCS type2 server: p u myDoc2.doc
        -----
        Sending to Doc_Map request: p u myDoc3.doc
5905 server (16-1) ServerGroup1
        Sending to Doc_Map request: p u myDoc1.doc
5905 server (16-1) ServerGroup1
        Subscribing client of 'Group1'
5906 clients (21-1) GCS_ClientGroup3
        ---Got group message: 'Message to save doc? I am client from group 3...'
5906 serverManager (13-1) Wrapper
        Unsubscribing client 21-1 from GCS system.
6205 server (17-1) ServerGroup2
        Subscribing client of 'Group2'
        -----
6605 server (17-1) ServerGroup2
        Subscribing client of 'Group2'
6606 clients (27-1) GCS_ClientGroup1
        Sent command to GCS type1 server: p u myDoc1.doc
6606 clients (24-1) GCS_ClientGroup2
        ---Got group message: ' Message to save doc? I am client from group 2...'
6606 serverManager (10-1) Wrapper
        Unsubscribing client 24-1 from GCS system.
6608 serverManager (13-1) Wrapper
        Unsubscribing client 25-1 from GCS system.
        -----
7405 server (18-1) ServerGroup3
        Subscribing client of 'Group3'

```

References

- [Booth80] Booth, T. L. and Wiecek, C. A., Performance Abstract Data Types as a Tool in Software Performance Analysis and Design, IEEE Transactions on Software Engineering, Vol. 6, No. 2, 1980, pp. 138-151.
- [Börger89] E. Börger, et al., Computability, Complexity, Logic, Studies in Logic and the Foundation of Mathematics, Vol. 128, 1989.
- [Boros97] Endre Boros, Peter Hammer, et al., Logical analysis of numerical data, Mathematical Programming, Vol. 79, No. 1-3, October 1997, pp. 163-190.
- [Borwein87] J. Borwein, Peter Borwein, Pi and the AGM, John Wiley & Sons, 1987.
- [Box87] Box, G. E. P., Empirical Model Building and Response Surfaces, John Wiley, 1987.
- [Buhr96] Buhr, R. J. and Casselman, R. S., Use Case Map Approach to High-Level Design of Object-Oriented Systems, Prentice-Hall, Inc., 1996.
- [CCITT] CCITT, Recommendation Z.120: Message Sequence Chart (MSC), Technical Report, Geneva
- [Courtois97] Courtois, M., An Automated, Incremental Resource Function Construction Technique, Master's Thesis, Carleton University, 1997.
- [DCE92] Open Software Foundation, Distributed Computing Environment: an overview, Technical Report, 11 Cambridge Center, Cambridge, MA 02142, January 1992
- [Flanagan92] J. Kelly Flanagan, et al., BACH: BYU Address Collection Hardware, The Collection of Complete Traces, Computer Performance Evaluation '92: Modelling Techniques and Tools, Proceedings of the 6th international Conference on Modelling Techniques and Tools for Computer Performance Evaluation, Edinburgh, Scotland, 16-18 September, Edinburgh University Press, 1992.
- [Franks95] Greg Franks, et al., A toolset for performance engineering and software design of client-server systems, Performance Evaluation, Vol. 24, 1995, pp. 117-136
- [Hofmann88] R. Hofmann, et al., An Approach to Monitoring and Modelling of Multiprocessor and Multicomputer Systems, Performance of Distributed and Parallel Systems, Proceedings of the

IFIP TC 7/WG 7.3 International Seminar on Performance of Distributed and Parallel Systems,
Kyoto, Japan, 7-9 December, 1988

- [Hofmann94] R. Hofmann, et al, Distributed performance monitoring: methods, tools, and applications, IEEE Transactions on Parallel and Distributed Systems, 5(6):505-597, June 1994
- [Hrischuk95a] Curtis Hrischuk, The automatic generation of software performance models using prototypes. Master's Thesis, Dept. of Systems and Computer Engineering, Carleton University, 1995.
- [Hrischuk95b] Curtis Hrischuk, J. Rolia, and C. M. Woodside, Automatic generation of a software performance model using an object-oriented prototype. In International Workshop on Modelling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS'95), 1995, pages 399-409
- [Hrischuk97-S] Curtis Hrischuk, A General Approach to retrieving resource functions. Technical report, Carleton University, September 1997.
- [Hrischuk97-O] Curtis Hrischuk, Towards a performance analysis component for ObjecTime, Technical report, Carleton University, October 1997.
- [Hrischuk99] Curtis Hrischuk, Trace-based Characterization for Generating Performance Software Models, IEEE Transactions on Software Engineering, vol. 25, no. 1, Jan./Feb. 1999.
- [Hussain96] K. Hussain, Perl 5 Unleashed. SAMS Publishing, 1996.
- [Jacobson95] Ivar Jacobson, Use Cases. SAMS Publishing, 1996.
- [Jain91] R. Jain, The Art of Computer Systems Performance Analysis, John Wiley & Sons, New York, New York, 1991
- [Kroeger92] Reinhold Kroeger, et al., JEWEL: Experiences with a Distributed Measurement System, Computer Performance Evaluation '92: Modelling Techniques and Tools, Proceedings of the 6th international Conference on Modelling Techniques and Tools for Computer Performance Evaluation, Edinburgh, Scotland, 16-18 September, Edinburgh University Press, 1992.
- [Levin96] L. Levin, Computational complexity of functions, Theoretical Computer Science, Vol. 157, No. 2, May 05 1996, pp. 267-271

- [Myers89] Myers, R. H., et al., Response Surface Methodology: 1966-1988. Technometrics, Vol. 31, No. 2, 1989, pp. 137-157.
- [Menasce97] Menasce, D., A Framework for Software Performance Engineering of Client/Server Systems, Proceedings in CMG'97, Orlando, Florida.
- [Neilson95] J. E. Neilson, et al., Software Bottlenecking in Client-Server Systems and Rendezvous Networks, IEEE Transactions on Software Engineering, Vol. 21, No. 9, 1995, pp. 776-782
- [Ogihara96] Mitsunori Ogihara, Functions computable with limited access to NP, Information Processing Letters, Vol. 58, No. 1, April 08 1996, pp. 35-38
- [Ousterhout94] J. K. Ousterhout, Tcl and the Tk Toolkit. Addison-Wesley, 1994.
- [Prenter75] P. M. Prenter, Splines and Variational Methods, John Wiley, 1975.
- [Press88] W. H. Press, et al., Numerical Recipes in C, The Art of Scientific Computing. Cambridge University Press, 1988.
- [Riek91] Maurice van Riek, A General Approach to the Monitoring of Distributed Memory Machines (A survey). Ecole Normale Supérieure de Lyon, June 1995.
- [Rolia95] J. A. Rolia and K. C. Sevcik, The Method of Layers. IEEE Transactions on Software Engineering, Vol. 21, No. 8, 1995, pp. 689-700
- [Rumbaugh91] J. Rumbaugh, et al., Object-Oriented Modelling and Design, Prentice-Hall, Inc., Englewood Cliffs, NJ07632 USA, 1991
- [Sauer81] C. H. Sauer, K. Mani Chandy, Computer Systems Performance Modelling, Prentice-Hall, Inc., Englewood Cliffs, NJ07632 USA, 1981
- [Selic94] B. Selic, et al., Real-Time Object-Oriented Modelling. John Wiley, 1994
- [Smith90] Smith, C., Performance Engineering of Software Systems, Addison-Wesley Publishing Co., New York, NY, 1990

- ❑ [Snelick90] Snelick, R., S-Check: a Tool for Tuning Parallel Programs, National Institute of Standards and Technology, robert.snelick@nist.gov, 1990

- ❑ [Steppler97] Steppler, M., Lott M., SPEET - SDL Performance Evaluation Tool. In Cavalli, Ana and Sarma, Amardeo, editors, Proceedings of the 8th SDL Forum: SDL '97 - Time for Testing - SDL, MSC and Trends, pp. 53-67, 23rd-26th September 1997, Evry, France.

- ❑ [UML97] Object Management Group, UML Semantics Version 1.1, OMG technical note ad/97-08-04, Object Management Group, September 1997

- ❑ [UML98] Object Management Group, UML Meta-model Abstract Syntax 1.3V4, OMG technical note ad/98-12-04, Object Management Group, December 1998

- ❑ [Vetland93] Vetland, V., Measurement based Composite Computational Work Modelling of Software, Ph. D. Thesis, University of Trondheim, April 1993

- ❑ [Vetland96] V. Vetland and C. M. Woodside, A Workbench for Automation of Systematic Measurement of Resource Demands of Software Components, Technical report, Carleton University, SCE-96-06.

- ❑ [Vetland97] V. Vetland and C. M. Woodside, Systematic Performance Characterization of Software Components, Technical report, Carleton University, SCE-97-06.

- ❑ [Wybraniec88] Dieter Wybraniec and Dieter Haban, Monitoring and Performance Measuring Distributed Systems during Operation, ACM 0-89791-254-3/88/0005/0917, p. 197 - p. 206.

- ❑ [Woodside95] C. M. Woodside, The Stochastic Rendezvous Network for Performance of Synchronous Client-Server-Like Distributed Software, IEEE Transactions on Computers, Vol. 44, No. 1, 1995, pp. 20-34.

- ❑ [Woodside98] C. M. Woodside, Curtis Hrischuk, Stefan Bayarov, Bran Selic, A Wideband Approach to Integrating Performance Predictions into a Software Design Environment, WOSP-98.

- ❑ [Xu96] Xu, Z. and Hwang, K., Modelling Communication Overhead: MPI and MPL Performance on the IBM SP2, IEEE Parallel and Distributed Technology, Spring 1996, pp. 9-23