PM-1 3 1/2" x 4" PHOTOGRAPHIC MICROCOPY TARGET
NBS 1010a ANSI/ISO #2 EQUIVALENT

PRECISION^{SM} RESOLUTION TARGETS

**Performance Modeling and Measurements
of Object-Oriented Software**

by

Luminita Simona Metea

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

Master of Engineering

Ottawa-Carleton Institute for Electrical Engineering
Faculty of Engineering
Carleton University
Ottawa, Ontario, Canada, K1S 5B6

16 September 1996

© Copyright 1996, Luminita S. Metea

National Library
of Canada

Acquisitions and
Bibliographic Services Branch

35 Wellington Street
Ottawa, Ontario
K1A 0N4

Bibliothèque nationale
du Canada

Direction des acquisitions et
des services bibliographiques

35, rue Wellington
Ottawa, Ontario
K1A 0N4

0-612-19501-5

0-612-19501-5

The author has granted an irrevocable non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of his/her thesis by any means and in any form or format, making this thesis available to interested persons.

L'auteur a accordé une licence irrévocable et non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de sa thèse de quelque manière et sous quelque forme que ce soit pour mettre des exemplaires de cette thèse à la disposition des personnes intéressées.

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

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

ISBN 0-612-19501-5

Canada

Dissertation Abstracts International and Masters Abstracts International are arranged by broad, general subject categories. Please select the one subject which most nearly describes the content of your dissertation or thesis. Enter the corresponding four-digit code in the spaces provided.

Electronics and Electrical

SUBJECT TERM

0 5 4 4

SUBJECT CODE

UMI

Subject Categories

THE HUMANITIES AND SOCIAL SCIENCES

COMMUNICATIONS AND THE ARTS

Architecture 0729
Art History 0307
Cinema 0900
Dance 0378
Fine Arts 0357
Information Science 0723
Journalism 0391
Library Science 0399
Mass Communications 0708
Music 0413
Speech Communication 0459
Theater 0468

EDUCATION

General 0515
Administration 0514
Adult and Continuing 0516
Agricultural 0517
Art 0273
Bilingual and Multicultural 0282
Business 0688
Community College 0275
Curriculum and Instruction 0277
Early Childhood 0518
Elementary 0524
Finance 0277
Guidance and Counseling 0519
Health 0480
Higher 0745
History of 0520
Home Economics 0278
Industrial 0521
Language and Literature 0279
Mathematics 0280
Music 0522
Philosophy of 0998
Physical 0522

Psychology 0525
Reading 0535
Religion 0527
Science 0714
Secondary 0533
Social Sciences 0534
Sociology of 0340
Special 0529
Teacher Training 0530
Technology 0710
Tests and Measurements 0286
Vocational 0747

LANGUAGE, LITERATURE AND LINGUISTICS

Language .....
  General 0679
  Ancient 0289
  Linguistics 0290
  Modern 0291
Literature .....
  General 0401
  Classical 0294
  Comparative 0295
  Modern 0297
  Modern 0298
  African 0316
  American 0391
  Asian 0305
  Canadian (English) 0352
  Canadian (French) 0355
  English 0393
  Germanic 0311
  Latin American 0312
  Middle Eastern 0315
  Romance 0313
  Slavic and East European 0314

PHILOSOPHY, RELIGION AND

THEOLOGY

Philosophy 0422
Religion .....
  General 0318
  Biblical Studies 0321
  Clergy 0319
  History of 0320
  Philosophy of 0322
Theology 0469

SOCIAL SCIENCES

American Studies 0323
Anthropology .....
  Archaeology 0324
  Cultural 0326
  Physical 0327
Business Administration .....
  General 0310
  Accounting 0272
  Banking 0770
  Management 0454
  Marketing 0398
Canadian Studies 0385
Economics .....
  General 0301
  Agricultural 0303
  Commerce-Business 0505
  Finance 0308
  History 0309
  Labor 0310
  Theory 0511
Failures 0438
Geography 0346
Gerontology 0351
History .....
  General 0578

Ancient 0579
Medieval 0581
Modern 0382
Black 0328
African 0331
Asia, Australia and Oceania 0332
Canadian 0334
European 0335
Latin American 0336
Middle Eastern 0333
United States 0337
History of Science 0385
Law 0398
Political Science .....
  General 0415
  International Law and
    Relations 0414
  Public Administration 0417
Recreation 0614
Social Work 0452
Sociology .....
  General 0426
  Criminology and Penology 0427
  Demography 0938
  Ethnic and Racial Studies 0431
  Individual and Family
    Studies 0438
  Industrial and Labor
    Relations 0429
  Public and Social Welfare 0430
  Social Structure and
    Development 0700
  Theory and Methods 0344
Transportation 0709
Urban and Regional Planning 0999
Women's Studies 0453

THE SCIENCES AND ENGINEERING

BIOLOGICAL SCIENCES

Agriculture .....
  General 0 73
  Agronomy 0 85
  Animal Culture and
    Nutrition 0475
  Animal Pathology 0476
  Food Science and
    Technology 0399
  Forestry and Wildlife 0478
  Food Culture 0479
  Plant Pathology 0480
  Plant Physiology 0817
  Range Management 0777
  Wood Technology 0744

Biology

General 0306
Anatomy 0287
Botany 0308
Botany 0309
Cell 0379
Ecology 0329
Endocrinology 0353
Genetics 0369
Immunology 0793
Microbiology 0410
Molecular 0507
Neuroscience 0317
Oncology 0416
Physiology 0333
Radiation 0821
Veterinary Science 0778
Zoology 0472

Biochemistry

General 0784
Medical 0740

EARTH SCIENCES

Biogeochemistry 0426
Geochemistry 0994

Geology 0370
Geology 0372
Geophysics 0373
Hydrology 0388
Mineralogy 0411
Paleobotany 0345
Paleontology 0426
Paleontology 0418
Paleontology 0985
Paleontology 0427
Physical Geography 0348
Physical Oceanography 0415

HEALTH AND ENVIRONMENTAL SCIENCES

Environmental Sciences 0748
Health Sciences .....
  General 0564
  Anesthesiology 0300
  Chemotherapy 0992
  Dentistry 0347
  Education 0380
  Hospital Management 0749
  Human Development 0738
  Immunology 0982
  Medicine and Surgery 0544
  Mental Health 0347
  Nursing 0569
  Nutrition 0570
  Obstetrics and Gynecology 0380
  Occupational Health and
    Safety 0384
  Ophthalmology 0381
  Physiology 0571
  Pharmacology 0573
  Pharmacy 0573
  Physical Therapy 0382
  Public Health 0574
  Radiology 0574
  Recreation 0575

Speech Pathology 0460
Toxicology 0383
Home Economics 0386

PHYSICAL SCIENCES

Pure Sciences .....
  Chemistry .....
    General 0485
    Agricultural 0749
    Analytical 0486
    Biochemistry 0487
    Inorganic 0488
    Nuclear 0738
    Organic 0490
    Pharmaceutical 0491
    Physical 0494
    Polymer 0495
    Radiation 0734
  Mathematics 0405
Physics .....
  General 0405
  Acoustics 0986
  Astronomy and
    Astrophysics 0406
  Atmospheric Science 0408
  Atomic 0748
  Electronics and Electricity 0407
  Elementary Particles and
    High Energy 0798
  Fluid and Plasma 0739
  Molecular 0409
  Nuclear 0510
  Optics 0732
  Radiation 0736
  Solid State 0411
Statistics 0443
Applied Sciences .....
  Applied Mathematics 0346
  Computer Science 0984

Engineering .....
  General 0537
  Aerospace 0538
  Agricultural 0539
  Automotive 0540
  Biomedical 0541
  Chemical 0542
  Civil 0543
  Electronics and Electrical 0544
  Heat and Thermodynamics 0545
  Hydraulic 0546
  Industrial 0547
  Marine 0547
  Materials Science 0794
  Mechanical 0548
  Metallurgy 0743
  Mining 0551
  Nuclear 0552
  Packaging 0549
  Petroleum 0745
  Sanitary and Municipal
    System Science 0790
  System Science 0790
  Technology 0428
  Operations Research 0796
  Plasma Technology 0795
  Turbine Technology 0994

PSYCHOLOGY

General 0421
Behavioral 0384
Clinical 0422
Developmental 0420
Experimental 0423
Industrial 0424
Personality 0425
Physiological 0989
Psychiatry 0426
Psychometrics 0427
Social 0428

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

**Performance Modeling and Measurements
of Object-Oriented Software**

submitted by Luminita Simona Metea, M.Sc.
in partial fulfillment of the requirements for
the degree of Master of Engineering

Chair, Department of Systems and Computer Engineering

Thesis Supervisor

Carleton University
16 September 1996

Abstract

The thesis contributes to the general effort to introduce performance analysis in the software development process and to better understand the performance implications of various object-oriented (OO) features.

The research presented in the thesis was conducted in two main directions. The first one is related to performance modeling of distributed OO software. The analytic modeling technique used is the Layered Queueing Network (LQN), an extension to the well known Queueing Network model. The thesis presents a method to map OO components to LQN model components and to compute the parameters for the LQN model from workload requirements of OO components. An LQN model is built for a system from the telecommunication field (a distributed conference-call). The model is then used to explore the overall system performance, to identify bottlenecks and to devise improvements.

The second direction of research studied the performance implications of various C++ features (creation/destruction of objects, inheritance, composition, association/aggregation) and measured their execution overheads. A better understanding of how execution time is spent in C++ software was gained and some solutions to accelerate the programs by trading off the generality of association/aggregation implementation for execution speed were proposed.

To my parents

Acknowledgments

I would like to express my deep gratitude to my supervisor, Professor Dr. Dorina Petru, who led me to this interesting and challenging research area, for her valuable guidance, encouragement and consistent support throughout the course of this work.

I would like to offer my appreciation to the members of the RADS lab, for their full support, friendship and a wonderful working climate.

The support and assistance from secretaries in the Department of Systems and Computer Engineering are also greatly appreciated.

I would like to thank Carleton University and the Natural Sciences and Engineering Research Council of Canada for their funding support.

Finally, I wish to give my special appreciation to my parents and brother for their consistent support and encouragement which they have given me throughout the research

Chapter 1

Introduction

1.1 Thesis Objectives

The object-oriented (OO) techniques for analysis, design and construction of software systems have raised a great amount of interest in the software industry due to their promised benefits such as reusability, extensibility, robustness, reliability and scalability [Wil93]. An important use of the object technologies is in building and managing distributed applications. The publication of CORBA (Common Object Request Broker Architecture) specification by the Object Management Group [OMG92] is a strong indication of a growing trend in the application of OO techniques in industrial settings.

Despite the theoretical and practical progresses made in the OO field, there are still pertinent issues which have not been firmly addressed, as pointed out by a panel of leading experts from industry and academia who came together at the invitation of IBM and ACM to forecast the future needs and directions of growth for the object technology [Pan95]. Among the unsolved issues that is mentioned the capability to evaluate performance characteristics (as for example, response time, resource utilization, throughput) and the scalability of the systems as they are being designed and implemented. In order to do so, the development team should be able to compose performance models based on the basic object model of the system under construction and on the resource requirements of

3.3.2	The Object Model of ACCS	29
3.3.3	Mainline Use Cases and Their Execution Scenarios	32
3.3.4	LQN Performance Model of ACCS	35
3.3.5	Experimental Results	39

Chapter 4

Measuring Tools and Techniques 46

4.1	Quantify	46
4.2	Getrusage system call	48

Chapter 5

OO Related Execution Overheads 54

5.1	General OO Concepts in C++	54
5.2	Constructors and Destructors	56
5.2.1	Static, Automatic and Dynamic Creation of Objects	58
5.2.2	Copy Constructor vs. Assignment Operator	60
5.2.3	Creation/Destruction of Composed Objects	62
5.2.4	Initialization List vs. Assignment in Constructor Body	64
5.2.5	Constructors in a Hierarchy	66
5.3	Member Function Calls	68
5.3.1	Call by Value or by Reference	69
5.3.2	Virtual Functions and Dynamic Binding	73
5.4	Implementation of an Object Model in C++	78
5.4.1	Attribute and Association	78

while a wide range of options exists, and continuing throughout the life-cycle. The purpose is to design and build systems that meet their performance objectives and to eliminate the need for late-fixing of performance problems. SPE is applied in two main steps. Firstly, software execution models (a kind of control-flow models) are built for a few typical, frequently executed workload scenarios. The execution models specify the activities to be executed in response to predefined events, activities which correspond to various software modules. Best-and-worst case hardware resource requirements for each activity (software module) are estimated, and then total average requirements per resource for each workload scenario are obtained. Secondly, the workload description obtained from the software execution model is mapped to a system execution model (typically a QN model) representing the hardware resources for which the software modules are competing. The software workload is usually modeled as QN customers, and the hardware as QN servers. The system model solution produces metrics for device residence times (including queueing delays), device utilization, overall response time, throughput, etc. These values can be mapped back into the software domain, finding the elapsed time, resource utilization, etc., for each software module.

SPE is applied in [Smi90] and in a later paper [SW93] to concurrent systems which follow a more traditional, process-oriented (as opposed to object-oriented) development, stressing the chronological sequence of activities executed by the system. It is natural to build flow-like software execution models, as required by SPE, in the context of a traditional software development, where the design starts with specifying the sequence of activities, then allocates the activities to various software modules.

List of Figures

Figure 1: Example of an LQN model	16
Figure 2: Object clustering based on execution paths	21
Figure 3: Example of object clusters	25
Figure 4: Object model of the ACCS application (using OMT notation)	31
Figure 5: Message sequence chart for a conference call	33
Figure 6: Message sequence chart for a regular call	35
Figure 7: LQN performance model of the ACCS	37
Figure 8: Execution path for adding a conferee	38
Figure 9: Response time for "Add a Conferee" in function of the system load	41
Figure 10: Utilization of various software and hardware servers for an increased number of call_processing_server instances	43
Figure 11: Hardware solution to the system bottleneck	44
Figure 12: Software solution to the system bottleneck (code optimization)	45
Figure 13: Coarse granularity histogram for getrusage results	52
Figure 14: Fine granularity histogram for getrusage results	53
Figure 15: Default copy constructor copies the pointer, but not the allocated memory	61
Figure 16: One-to-one relationship	80
Figure 17: Instance diagram of the association implementation	81
Figure 18: One-to-many relationship	82
Figure 19: Instance diagram of the association implementation	83

Figure 20: Object model of the point-of-sale application	85
Figure 21: Conceptual class model	86
Figure 22: Detailed class model	87
Figure 23: Relationships between objects	88
Figure 24: MSC for the creation of a salesTransaction object	89
Figure 25: MSC for calculate_total of a sales transaction	91
Figure 26: New implementation of the transaction object	94
Figure 27: MSC for the new salesTransaction constructor	95
Figure 28: MSC for calculate_total of a transaction	96

Chapter 1

Introduction

1.1 Thesis Objectives

The object-oriented (OO) techniques for analysis, design and construction of software systems have raised a great amount of interest in the software industry due to their promised benefits such as reusability, extensibility, robustness, reliability and scalability [Wal93]. An important use of the object technologies is in building and managing distributed applications. The publication of CORBA (Common Object Request Broker Architecture) specification by the Object Management Group [OMG92] is a strong indication of a growing trend in the application of OO techniques in industrial settings.

Despite the theoretical and practical progresses made in the OO field, there are still pertinent issues which have not been firmly addressed, as pointed out by a panel of leading experts from industry and academia who came together at the invitation of IBM and ACM to forecast the future needs and directions of growth for the object technology [Pan95]. Among the unsolved issues that is mentioned the capability to evaluate performance characteristics (as for example, response time, resource utilization, throughput) and the scalability of the systems as they are being designed and implemented. In order to do so, the development team should be able to compose performance models based on the basic object model of the system under construction and on the resource requirements of

individual objects. Performance analysis during the software development process will help to insure that the system satisfies its performance requirements, a very important concern especially in systems with real-time constraints. Also, such analysis helps in evaluating the utilization of various resources, in predicting potential system behavior, and most importantly, in comparing different design choices and hardware arrangements and in identifying trade-offs.

The present thesis is part of the general effort to introduce performance analysis in the software development process and to better understand the performance implications of using various OO features. The thesis research was conducted in two main directions:

- a) performance modeling of distributed OO software by applying the analytical performance model Layered Queueing Network (LQN) [WNPM95, RS95];
- b) measuring the execution overheads of different OO features in C++ programs.

1.2 Contributions

The main contributions of the thesis are as follows:

- established a method to map OO components to LQN model components and to compute the parameters for the LQN model from workload requirements of OO components;

- applied LQN to an example of distributed OO software system which was studied and modelled by simulation in a previous thesis [Liu94], explored the overall system performance, identified software and/or hardware bottlenecks and devised improvements;
- studied the performance implications of various C++ features (creation/destruction of objects, inheritance, composition, association/aggregation) and measured their execution overheads;
- gained a better understanding of how execution time is spent in C++ software and proposed some solutions to accelerate the programs by trading off the generality of association/aggregation implementation for execution speed.

The first two issues mentioned in the list of contributions are contained in the paper entitled "Analytic Performance Models of Concurrent Object-Oriented Software" [PCML95] submitted for publication in the "Theory and Practice of Object Systems" journal.

1.3 Thesis Content

Chapter 2 provides background information related to analytical performance models and to issues related to performance modeling of software systems. It presents also some of the most important object-oriented development methods.

Chapter 3 presents a detailed description of the LQN model, provides a method for building LQN models for concurrent OO systems and applies it to study the overall performance of a telecommunication system used as a case study.

Chapter 4 describes the different measuring tools and techniques that were used in the thesis to measure the performance of C++ programs.

Chapter 5 investigates specific OO related execution overheads (creation/destruction of objects, inheritance, composition, function calls) and analyzes different implementation alternatives of association/aggregation relationships in an object model. It uses a case-study for measurements.

Chapter 6 concludes the study, summarizes the findings and the future work.

Chapter 2

Background Literature

2.1 Performance Modeling of Software Systems

Queueing Networks (QNs) are well established as practical tools in computer system performance analysis and capacity planning (see, for example, [LZGS84], [Lav89]). A QN model is a collection of service centers or nodes, each with its own queue and servers, and customers or jobs representing the consumers of service. Every service center represents a common resource shared by customers and has an established discipline to dispatch the requests for service. After the service completion at a center, a customer may move to another center according to a routing chain. Except for the contention for common resources, the customers are independent. They may be divided into several classes of statistically identical customers, each class with different routing probabilities and service demands. The advantage of QNs is their combination of expressive power and solution efficiency. The notation used to describe the models is powerful and at a high level of abstraction; the models are easy to build and require relatively few parameters. Furthermore, there is a large range of analytical solution methods, some of them very efficient (especially those for the class of product form QNs [LZGS84]). However, QNs have important limitations in modeling concurrent software for distributed systems; it is

difficult, for example, to represent synchronization among customers, or to capture in a simple way the contention for software resources.

The modeling technique used in the thesis is the *Layered Queueing Network* (LQN). LQN is an extension of the QN model. The LQN model inherits the characteristics of the QN model, with the added bonus that it is able to represent the contention for software resources and it allows for nested services. Another difference is the fact that the servers may have two distinct phases of service -- the first one executed while the client is blocked waiting for a reply, and the second one executed in parallel with the client.

LQN was previously developed independently under the name of *Stochastic Rendezvous Network* (SRVN) in [Woo89], [WNPM95], [PW91], [Pet94] and under the name of *Method of Layers* (MOL) in [Rol92] and [RS95]. A performance toolset for LQN including various solving approaches (analytical and by simulation) is presented in [FHMN95]. LQN is an extension to the more widely known Queueing Network (QN) model. LQN is different from QN in two important ways: the servers can act also as clients to other servers (allowing for nested services), and the servers may have two distinct phases of service -- the first one executed while the client is blocked waiting for a reply, and the second one executed in parallel with the client. The following chapter will give a more detailed description of the LQN model.

The idea of introducing performance modeling in the software development process is not characteristic only to the OO technology. Smith introduced in [Smi90] the concept of *Software Performance Engineering* (SPE) as a systematic way to use quantitative methods to assess requirements, design and hardware alternatives, starting early in the life-cycle

while a wide range of options exists, and continuing throughout the life-cycle. The purpose is to design and build systems that meet their performance objectives and to eliminate the need for late-fixing of performance problems. SPE is applied in two main steps. Firstly, software execution models (a kind of control-flow models) are built for a few typical, frequently executed workload scenarios. The execution models specify the activities to be executed in response to predefined events, activities which correspond to various software modules. Best-and-worst case hardware resource requirements for each activity (software module) are estimated, and then total average requirements per resource for each workload scenario are obtained. Secondly, the workload description obtained from the software execution model is mapped to a system execution model (typically a QN model) representing the hardware resources for which the software modules are competing. The software workload is usually modeled as QN customers, and the hardware as QN servers. The system model solution produces metrics for device residence times (including queueing delays), device utilization, overall response time, throughput, etc. These values can be mapped back into the software domain, finding the elapsed time, resource utilization, etc., for each software module.

SPE is applied in [Smi90] and in a later paper [SW93] to concurrent systems which follow a more traditional, process-oriented (as opposed to object-oriented) development, stressing the chronological sequence of activities executed by the system. It is natural to build flow-like software execution models, as required by SPE, in the context of a traditional software development, where the design starts with specifying the sequence of activities, then allocates the activities to various software modules.

In the case of the OO development, the design process focuses first on identifying the objects and classes, their roles and relationships, their similarities and differences [LPC95]. Due to the fact that the objects attempt to encapsulate both data and behavior, and that the OO development stresses the principle of information hiding, it is easy to get caught in details of various individual objects and thus to lose track of the overall system behavior. However, the overall performance measures are a system characteristic; for example, the response times for different requests coming to the system, are determined by the length of the execution paths through the system. Therefore, the OO design methods which emphasize execution paths, as for example use cases in [JCJO92] and use-case maps in [BC95] are very appropriate for designing systems with real-time constraints, and also for integrating the performance analysis into the software development process. (Different OO design methods are reviewed in the next section.)

Modeling the performance of concurrent and distributed software is not a trivial task. A performance model has to capture the behavior of the whole system (application software, operating system, hardware devices) because all these components bring their own contribution to the overall performance. At the same time, in order to keep the model tractable, only an abstract view of the system can be represented. Another requirement is to preserve some semantic equivalence between the structure of the concurrent software and the components of the performance model. This equivalence facilitates the derivation and calibration of the model, as well as the interpretation and use of its results for further improvements of the actual software. In order to reconcile these somehow contradictory requirements, [Woo95] introduces a three-view characterization of concurrent software

systems which can be used to analyze performance information: a map view (the structural description of the software), a path view (the execution path corresponding to various scenarios) and a resource view (the hardware resources for which the software components compete). These views are also present in the performance model for concurrent OO software described in the next chapter of the thesis.

2.2 OO Methodologies

There are many object-oriented development methods that provide a step-by-step approach to building OO systems. Some of the most important methods are described in [JCJO92].

Object-Oriented Analysis (OOA) [CY91]

The OOA method by Peter Coad and Ed Yourdon is used to build an analysis model to describe the functionality of the system. The method has five steps.

The first step consists of finding the classes and objects. The techniques for finding the objects are heuristic. According to Coad, an object model has four components:

- **Problem domain** (consists of the objects relating to the business at hand);
- **Human interaction** (consists of objects, like windows and reports, that provide an interface between problem domain objects and people);
- **Data management** (consists of the objects that provide an interface between the problem domain objects and the database or the file management system);

- System interaction (contains objects that provide an interface between problem domain objects and other systems and devices).

The next four steps are: identifying structures (like generalization-specification and whole-part), defining subjects (as larger units consisting of objects), defining attributes (by identifying information and associations that should be associated with every object), and defining services (i.e. operations of the classes).

The method was designed initially to be used for small systems. It does not provide any specific support to capture all the dynamic roles played by the objects.

Object-Oriented design (OOD) [Boo91]

The OOD method introduced by Grady Booch offers a collection of techniques and heuristics that can be used to develop OO systems. OOD gives the developer the possibility to view the system from different perspectives. From a static perspective, there are the logical view (consisting of the class structure and the object structure), and the physical view (consisting of the modules of the system). From a dynamic perspective, there are the state diagram (that describes the semantics of instances of a certain type) and the timing diagram (that describes how events happen between objects).

The method refines every class and instance until it can be implemented by components and code. It uses the relationships *uses* and *instantiates* between classes. It doesn't provide a technique for finding the operations associated with each object and class.

The developer has to use his creativity in the suggested steps of the method:

- Identify classes and objects at a given level of abstraction;

- Identify the semantics of these classes and objects;
- Identify the relationships among these classes and objects;
- Implement these classes and objects.

Hierarchical Object-Oriented Design (HOOD) [HOOD89a, HOOD89b]

HOOD was initially developed for the European Space Agency. It is a method based on stepwise refinement. The method is fully geared towards Ada, thus not providing any support for inheritance and pure classes. It uses concepts like active objects (objects that can execute in parallel), and passive objects (objects that execute sequentially).

The method identifies one object for the whole system, called root object. The root object is divided then in other internal objects that are specified. The subdivision continues recursively in all objects, until a level is reached that can be implemented with Ada packages. Thus a hierarchical structure is created.

The proposed phases of the method for every object, until the lowest level is reached, are:

- Problem definition;
- Elaboration of an informal solution strategy;
- Formalization of the strategy;
- Formalization of the solution.

The method, which can be characterized as object decomposition, does not provide any help in finding the appropriate object structure. One flaw of the method is that most of the

major decisions have to be taken in the early stages of development when dividing the system into its components.

Object Modeling Technique (OMT) [RBPEL91]

The OMT method developed by Rumbaugh et al. involves many techniques and guidelines necessary in the development of a system. It uses various concepts that make it useful for expressing different modeling situations.

The method develops initially three models of the system: the object model (describes the static structure of the system and their relationships), the dynamic model (captures the temporal aspects of the object model with events and states of the objects), and the functional model (describes how output values can be obtained from input values in terms of the operations on the objects).

These models are refined in the four phases of the method: analysis, system design, object design and implementation.

Responsibility-Driven Design (RDD) [WWW90]

The RDD method by Wirfs-Brock et al. uses informal techniques and guidelines to develop the design of the system. It uses on top of the basic concepts, like class, object and inheritance, other concepts like responsibilities of a class (specifies the roles of the objects and the actions of the objects), and collaborations of a class (shows how the objects interact).

The method consists of two phases, each one of them comprising different activities. The exploratory phase consists of finding the classes, their responsibilities and the

collaborations between classes. The refining phase consists of refining the hierarchies between classes, of grouping classes in subsystems in order to abstract a common functionality, and of refining the protocols to show the specific signature of each operation.

Use Case Maps (UCM) [BC95]

UCM were introduced by R.J.A. Buhr and R. Casselman to express use case behavior at different levels of abstraction. UCMs are especially useful in real time and concurrent systems. Use case maps are semi-formal tools that allow the designer to reason about the responsibilities of the components without going into details about the messaging between the components. They provide a behavioral framework for making architectural decisions at a high level of design, and also for characterizing behavior at the architectural level once the architecture is decided.

Chapter 3

The Layered Queueing Network Model

By the means of a case-study from the telecommunication domain, this chapter shows how to build and parameterize an LQN model of a concurrent OO system and, most importantly, how to use the model in order to explore the overall system performance, to identify software and/or hardware bottlenecks and to devise improvements.

3.1 Description of the LQN Model

An LQN model is represented by an acyclic graph as in Figure 1, which represents an example model of a database server accepting requests from n classes of client processes and using, in turn, a file server. The nodes in the graph are software components (represented as parallelograms) or hardware devices (represented as circles). The software components can be either "pure" client classes (root nodes) or servers (intermediate nodes). The hardware devices are usually leaf nodes. The arcs represent requests between clients and servers. The "pure" clients are the ones that drive the system; they are cyclic processes which execute on their own and then send requests to the servers below. Note that the LQN model defined here is a closed model (it has a fixed number of cyclical clients which never leave the system). LQN can be extended to an open system, and some work in this direction is presented in [WNPM95].

The particularity of the software servers is that they may act as clients to other servers while serving their own clients, allowing thus for nested services. If there is contention for a server (be it software or hardware), the requests arriving when the server is busy are queued in a single input queue owned by the server, and are served according to a given scheduling policy (for example FIFO or priority scheduling). The communication between a client and a server works as follows: the client sends a request to the server and blocks waiting for a reply. The server starts serving the request, when its turn comes, by executing a so-called "first phase" of service, after which it sends a reply to the client. Upon receiving the reply, the client resumes its own execution. At the same time, the server may continue with a "second phase" of service executed in parallel with the client. The second phase is useful to model those types of services for which the client does not have to wait (for example, a print server may accept commands from its clients, but the clients do not have to wait for the printing to be completed). There is no communication between an LQN server and its client at the end of the second phase of service.

The LQN clients can be grouped into several classes, such that the clients with identical stochastic behavior belong to the same class. Each client class is represented as a "shadowed" parallelogram (see Figure 1). An LQN server can be a *single-server* (represented by a single circle or parallelogram) or a *multi-server* (represented by a "shadowed" circle or parallelogram). A single-server has a single execution thread, while a multi-server represents a set of identical server instances, each with its own execution thread, serving requests from a common queue. The instances of a software multi-server can share the same CPU (meaning that they are *virtually* concurrent) or can run on a

multiprocessor with any number of CPUs (meaning that their concurrency is *real*). The number of instances in a multi-server can be either finite or infinite. The name infinite-server means that there is always an available instance for every request that arrives, so that the requests are never queued (which explains why infinite-servers are also named delay-

Figure 1: Example of an LQN model

servers). In Figure 1 there is a software multi-server with a fixed number of instances representing the Database server, and an infinite-server representing the "think" time spent by clients at the terminals.

Another important characteristic of the LQN servers is that they can offer a range of services, each one with a different execution time and number of requests to other servers. These different services are named *entries*, and are represented by parallelogram "slices". For example, in Figure 1 the File-server has two entries, read and write. All the entries of a server share the same implicit server queue (even if the communication graph contains separate request arcs for different entries of the same server). The notion of a range of services is also extended to the hardware servers (for example, requests for different execution times may arrive to a CPU). Since the hardware servers are leaf nodes and do not send any further requests, we decided to simplify the graphical notation and not to represent explicitly the hardware server entries.

Quantitative parameters associated with the LQN nodes and arcs characterize the following aspects: the number of clients in each client class; the number of servers in each multi-server; the mean execution times of each client and server; the mean number of requests sent by each client and intermediate server to other server entries; and the communication delay (if any) associated with a request.

In order to unify the notation, let us consider that a pure client has an entry and two execution phases which are repeated in a cyclical fashion, without accepting any request at the beginning of the cycle. Let denote by I_i a single-threaded software instance that belongs

to node i (i.e., I_i can be a client, a single-server or an instance of a multi-server). Note that all instances I_i of a node i are statistically identical, therefore indistinguishable. The phase p of an entry e of I_i is characterized by the following parameters:

s_{iep} = execution time of I_i , entry e , phase p

$y_{(iep)(jc)}$ = mean number of requests from I_i , entry e , phase p to node j entry c

$d_{(iep)(jc)}$ = mean communication delay associated with a request from I_i , entry e , phase p to node j entry c

The behavior of LQN clients and servers follows a number of stochastic assumptions which were adopted in order to allow for an analytical solution. Each entry e of I_i is a sequence of phases, and each phase p is a sequence of execution periods separated by requests to servers. Let Y_{iep} denote the mean number of requests made by I_i , entry e , phase p (i.e., the sum of all $y_{(iep)(jc)}$ over j and c). The number of execution periods in the phase is $(Y_{iep} + 1)$. During each execution period, the CPU is required for an exponentially distributed time with a mean of $s_{iep}/(Y_{iep} + 1)$. After every execution period, the next event is randomly selected: either a request to server j entry c with the probability $y_{(iep)(jc)}/(Y_{iep} + 1)$ or the end of the phase with the probability $1/(Y_{iep} + 1)$

Later in the paper we will use the following parameters of an entry e of I_i :

$$S_{ie} = \sum_p s_{iep} = \text{total execution time of } I_i, \text{ entry } e$$

$$Y_{(ie)(jc)} = \sum_p y_{(iep)(jc)} = \text{mean number of requests from } I_i, \text{ entry } e \text{ to node } j \text{ entry } c$$

It is worthwhile to note that LQN abstracts the behavior of software clients and servers in a way similar to the stochastic routing of customers in a QN model. It is not possible to prescribe exactly the order of requests made to various servers during a phase, nor the exact number of requests made each time the cycle is repeated (only their average number). Such a stochastic characterization of software behavior was proven to work well in models of large systems, where the performance model components represent large pieces of software with complex data dependencies that introduce variability in execution. By solving an LQN model we obtain average steady-state performance measures: average client throughputs, average queue lengths and queueing delays, average response times, server utilizations, etc. It is important to note that the queueing delays due to contention for software and hardware resources are an integral part of the end to end response times that characterize the responsiveness of the system. Thus we can explore the effect of contention for common resources on system performance, find out when the system reaches saturation, and which component(s) get saturated first. The answers to these questions point to solutions for performance improvement.

3.2 Building LQN Models for Concurrent OO Systems

An OO system is essentially a society of cooperating objects [Bri91]. Objects interact with each other through message passing. Because independent, self contained objects can be executed simultaneously, concurrent object-oriented systems constitute a natural extension to OO systems. Concurrency is one of the important issues that must be properly addressed in many application domains, such as transaction processing, embedded

software, process control and telecommunications, for the presence of concurrency may introduce problems such as deadlock, starvation, race conditions, critical sections [BD93] (detailed definitions can be found in [Tan92]). Concurrency also introduces contention for common resources, which in turn, affects the performance of the systems. The latter effect of concurrency is the main concern of this thesis.

In concurrent OO systems, concurrency can be either real or virtual, depending on the actual number of processors involved. *Real concurrency* means that two or more objects in the system are executing on different processors at the same time, while *virtual concurrency* refers to the situation where two or more objects are actually running on the same processor. We are interested in building performance models for distributed applications where both types of concurrency are present.

The researchers have different points of view when it comes to deciding when to consider concurrency issues in the process of designing concurrent OO systems. Many researchers think that even though concurrency is important, it should still be addressed after the identification of objects and classes [Boo91]. Conversely, Schumate (in [Fir93]) thinks concurrency is critical and must be dealt with first. Firesmith disagrees (Ibid.): the best approach seems to consist in considering both object identification and concurrency issues simultaneously, with concurrency and timing issues addressed in terms of both

objects and messages. Performance modeling takes into account the objects, their behavior and collaborations, as well as the concurrency issues.

We base our modelling technique on the following assumptions about concurrent OO systems. The basic communication mechanism between objects is message passing. All processing requests are sent to the appropriate object by embedding them into a message. Upon receiving the message, the receiver object processes it (that is, executes the requested

Figure 2: Object clustering based on execution paths

method) and sends, if necessary, future requests to other objects and waits for their reply. Different messages can have different processing requirements, frequency, and patterns of future requests. An object can process at any time at most one message, which means that the object has a single execution thread. Some objects may receive messages from multiple sources, in which case buffering facilities are provided. We assume here that the buffers are large enough such that no requests are lost at any time. The messages waiting in queue are served according to a given scheduling policy (e.g. FIFO). It is also possible to have multiple instances of the same object class serving requests from a common queue. All these assumptions make for a straightforward mapping of the objects to LQN elements (i.e., either clients or intermediate servers).

Another aspect we have to consider is that concurrent OO systems implementing distributed applications are, by their nature, large and complex, containing a great number of objects running on different hardware systems. As already mentioned, a performance model has to represent both the software and hardware components. However, we have to keep in mind that the size of a performance model is limited by the computational complexity of its solution methods [WNPM95]. When a distributed application contains a large number of objects, a one-to-one mapping of the objects into LQN elements is impractical because the model will become intractable (i.e., impossible to solve analytically). Therefore, we will group close collaborating objects into disjoint clusters; each cluster will be modeled in turn as an LQN element: a client, a single-server or an instance of a multi-server. This grouping has to consider not only the object relationships, but also their concurrency, since all the objects inside a cluster must be allocated on the

same processor and must be able to run under a single execution thread. In order to do a proper mapping of object clusters into LQN elements, the message queues are allowed only at the borders between clusters, not inside a cluster.

It is interesting to note that the clustering step uses the "map" view of the three-view model introduced in [Woo95]. Figure 2 illustrates object clustering in a hypothetical system (some of the objects may contain nested objects). The execution paths for two major scenarios (corresponding to the "path view") are also shown in the figure. Each such path, beginning and ending at a client, defines an end-to-end response time of the system. This includes the execution times of different components along the path, as well as the queueing delays due to contention for common resources. The "resource" view will be applied when considering the hardware resources (processors, I/O devices, communication networks, etc.) on which the software will run. Hardware resources are usually mapped to LQN leaf servers, as mentioned in the previous section.

In order to determine the parameters of an LQN model, we have to select first a few representative use cases (scenarios) that correspond to a typical, frequent usage of the system. A class of clients will be responsible for creating the load corresponding to each major use case. In the next step the LQN parameters are computed by following a different procedure for software and hardware components. For software components we identify all the services requested from every object cluster during the execution of every use case. The set of services offered by cluster i will become the set of entries of an instance I_i . For each

entry e of I_i we have to obtain the mean execution time S_{ie} and the mean number of requests $Y_{(ie)(jc)}$ to other entries.

The computation of S_{ie} begins with the identification of the set M_{ie} of all the object methods owned by objects inside the cluster, that are executed when the service represented by entry e is requested. For each method $m \in M_{ie}$ determine the following values:

v_m = average number of times m is executed for a single request of the service represented by entry e of I_i

δ_m = average invocation overhead (includes binding, message passing, etc.)

t_m = average execution time of the body of the method

Determine then the average entry execution time as:

$$S_{ie} = \sum_{m \in M_{ie}} v_m (\delta_m + t_m)$$

The computation of $Y_{(ie)(jc)}$ begins with the identification of the set M_{jc} of all the object methods inside cluster I_j which are executed when the service represented by entry c of I_j is requested. For each method $m \in M_{ie}$ and $k \in M_{jc}$ determine:

u_{mk} = average number of invocations of method k by method m

For example, Figure 3 illustrates two clusters I_i and I_j , where I_i provides a service e and I_j provides a service c . The sets M_{ie} and M_{jc} are:

Figure 3: Example of object clusters

$$M_{ie} = \{ \langle O_1, 1 \rangle, \langle O_2, 2 \rangle, \langle O_3, 3 \rangle \}$$

$$M_{jc} = \{ \langle O_4, 1 \rangle, \langle O_4, 2 \rangle, \langle O_5, 2 \rangle, \langle O_6, 1 \rangle \}$$

Notation $\langle O_n, l \rangle$ represents $\langle \text{object}_n, \text{method } l \rangle$.

In this example the parameter u_{mk} , $m \in M_{ie}$, $k \in M_{jc}$, is different of zero only for the following two calls:

- from $\langle O_2, 2 \rangle$ to $\langle O_4, 1 \rangle$
- from $\langle O_3, 3 \rangle$ to $\langle O_4, 2 \rangle$

In general the mean number of visits $Y_{(ie)(jc)}$ is determined as:

$$Y_{(ie)(jc)} = \sum_{m \in M_{ie}} \sum_{k \in M_{jc}} v_m \times u_{mk}$$

The LQN arcs may also have attached communication delays, which are evaluated considering the transfer rate of the communication networks and the average lengths of messages.

For hardware components which are leaf nodes in the LQN graph, the parameters S_{ie} are determined differently if the node is a CPU or another type of device. For a CPU node the service times for different entries are those corresponding to the software components running on that CPU. For other devices (such as peripheral devices, buses etc.) the value of the parameters S_{ie} is based on the duration of the corresponding service.

The process of building a performance model helps the developers to acquire a good understanding of the run-time behavior of the system, due to the fact that they have to

follow the most frequently execution paths and identify all the objects and methods along those paths. Determining the parameters of a performance model is a tedious process that can be aided by CASE tools once we know the resource requirements for each object (method) and the invocation frequencies. As mentioned in [Woo95] a potential weakness is the use of uncertain (guessed) values for the execution times of different methods when the analysis is done in the early phases of a project started from scratch. The uncertainty can be reduced either by reusing components already built (for which performance information should be available) or by implementing and measuring those components that are frequently executed and have a strong impact on system performance. An alternative approach to parameterizing a model is to establish time budgets for different components along the execution paths, then to verify and balance these budgets by solving the model under different loads and configurations. Once found, appropriate time budgets which insure that the product will meet its performance requirements should be enforced during the implementation.

3.3 Case-study: an Automatic Conference Call System (ACCS)

This section describes ACCS, a distributed OO system used as a case-study in the rest of the chapter. ACCS was first presented and then analyzed by the means of a simulation model in [Liu94]. In this section an LQN model for ACCS is discussed, then solved analytically. The analytical results are consistent with the simulation results from [Liu94]. The analysis of the model helps understanding performance trouble spots in the system and points to solutions for improvement.

3.3.1 ACCS Description

ACCS is a service available to the business subscribers of a telephone company. The service is running inside a telephone switch system on top of the Basic Call layer. In a nutshell, the ACCS service will allow any group of users to establish and moderate a voice conference call which simulates the real face to face meeting. Each conference call has a single conference controller whose role is similar to that of a meeting chairperson or moderator. This person will be able to perform the following actions:

Initiate a conference call: The conference calling service must be invoked by the served user, employing an initiating conference call request. When the service provider receives the request to allocate resources for the conference call, it checks to see that the requested conference can be established. This procedure includes conference resource availability check. The required resources are then allocated.

Add new party: The conference controller can request that a new party be added to a new or existing conference call. The conference call server will set up a call with the new party indicated by the party ID provided by the controller. The conference call server will make use of a subscriber directory to derive the actual phone number from the party ID. The subscriber directory also maintains the common name and picture of the party. Upon the establishment of the call, the new party will be connected to the conference.

Terminate the conference: The conference controller can request that the conference be terminated, which means every party associated with a particular conference be disconnected, and all conference resources be de-allocated.

During the initiation phase, the conference controller invokes the conference call service. Once the conference is in the active state, the service is under the control of the conference controller. Any party other than the conference controller is called a conferee. All participants in the conference call are called parties.

3.3.2 The Object Model of ACCS

The ACCS can be implemented as an object-oriented model. All requests and replies are done by using messages. From the ACCS description presented in the previous section a set of objects can be identified, each of them with distinct responsibilities in the application. These objects also have similarities, for example, similar role, behavior or responsibilities. The object model in Figure 4 depicts all the relationships among these objects, such as uses, subclassing, aggregation, etc. and the cardinalities for each relationship according to the OMT notation from [RBPEL91]. (The OMT notation is presented in Appendix A).

Conference_call and *regular_call* are two *call* classes in the telephone system. *Conference_call* keeps track of the information pertaining to a conference call, such as *conf_controller* id, number of conferees, each *conferee*'s id, the current state of the conference call. It also uses *conf_call_circuit* resources obtained through *conf_call_server* and *line* resources obtained through *call_processing_server*. *Conference_call* collaborates with the following: *conf_controller*, *conferees*, *conf_call_server*, *line* and *conf_call_circuit*.

Regular_call has similar responsibilities with *conference_call*, keeping track of the state information related to a regular call. It uses *line* resource obtained through *call_processing_server* and collaborates with *reg_caller*, *reg_callee*, *line*, and *call_processing_server*.

Conf_controller, *conferee*, *reg_caller* and *reg_callee* are subclasses of the telephone *user* class. *Conf_controller* controls the conference call, sends various processing requests to *conf_call_server*, such as to initialize a conference call, to add a new party and to terminate a conference call.

Reg_caller object controls the regular call. It sends connecting and terminating processing requests to *call_processing_server*.

Conferee and *reg_callee* objects behave very similarly. They receive connecting or terminating requests from *call_processing_server* and react according to their current state.

Conf_call_circuit and *line* are subclasses of the *resource* class. *Conf_call_circuit* keeps track of the state information for each conference circuit in the system while *line* represents the state for each line. *Line* class is further divided into three subclasses depending on the type of line used. They are *fiber*, *copper* and *other*.

Server is an abstract class with two subclasses: multi-server and single-server. The *multi-server* class has three subclasses: *conf_call_server*, *call_processing_server* and *database_server*. *Conf_call_server* receives and processes requests from *conf_controller*, sends call processing related requests to *call_processing_server* and sends conference

Figure 4: Object model of the ACCS application (using OMT notation)

queries to *database_server*. It also acquires and releases *conf_call_circuit* resource on behalf of the conference call.

Call_processing_server receives and processes requests from *conf_call_server* and *reg_caller*, sends translation and routing queries to *database_server*, and sends connecting and terminating requests to *conferees* and *reg_callees*. It also acquires and releases *line* resources on behalf of the *conference_call* and *regular_call*.

Database_server receives and processes requests from *conf_call_server* and *call_processing_server*. It also sends read and write messages to the disk.

Conf_controller_terminal, *conferee_terminal* and *regular_terminal* are three subclasses of the *terminal* class. *Conf_controller_terminal* collaborates with *conf_controller*. It is the physical device to send out requests issued by *conf_controller*. Similarly, *conferee_terminal* is the physical device to send out requests issued by *conferee*, *regular_terminal* is the physical device to send out requests issued by *reg_caller* and *reg_callee*.

3.3.3 Mainline Use Cases and Their Execution Scenarios

The mainline use cases represent the key scenarios occurring in a system and are important because they define the response times of interest for performance analysis. We consider in this case-study two scenarios corresponding to the execution of a conference call and of a regular call. All the interactions among conference call members and the sequence of messages and events occurring during a conference call are captured in the message sequence chart (MSC) shown in Figure 5. The execution scenarios are as follows:

First, a conf_controller picks up the phone and dials the feature activation code to

Figure 5: Message sequence chart for a conference call

activate the conference call, which generates an *init* message to *conf_call_server*. The *conf_call_server* then tries to acquire a conference circuit. If the circuit resource is successfully obtained, the *conf_call_server* notifies the *conf_controller* to proceed by sending the reply *yes*, otherwise, the *conf_controller* has to wait and try again. Upon receiving the *yes* notification, the *conf_controller* sends an *add* request to *conf_call_server* to add all the conferees. In order to retrieve all information about the conferees, such as image, phone number, name, etc., *conf_call_server* sends a *conf_query* message to a database server. If the query is successful, *conf_call_server* sends a *connect* message to *call_processing_server* passing the conferee's directory number and requests to set up a connection between *conf_controller* and the conferee. *Call_processing_server* first checks if there is a free line. If so, the switch will translate and route the call to the proper destination (conferee) through the line. The destination conferee's phone rings.

Once the conferee's phone goes off the hook, the connection is established, logs, operational measurement and billing system are notified, and the call goes into the *talking* state (shown as a heavily shaded bar in Figure 5).

When the conference call is about to finish, the *conf_controller* sends a *terminating* message to the *conf_call_server*. *Conf_call_server* forwards the message to the *call_processing_server* which disconnects the conferee and de-allocates the line resources assigned to the call. After all the conferees are disconnected, the *call_processing_server* notifies *conf_call_server* by sending the message *finish*. Upon receiving *finish* notification, *conf_call_server* frees the conference circuit used and the conference call is finished.

Similarly, the message sequence chart for regular call is represented in Figure 6.

Figure 6: Message sequence chart for a regular call

During performance analysis we may be interested in end-to-end response times for complete use cases, or only in response times for use case fragments. In the conference call example presented in Figure 5, we want to determine the response time for each of the main functions of a conference call: initiate the call, add conferees and terminate the call (ignoring the talk part).

3.3.4 LQN Performance Model of ACCS

An LQN performance model of the ACCS application is presented in Figure 7. It models the behavior of the system, based on the message sequence charts from Figure 5 and 6. All the elements of the LQN model are objects, not classes as in the object model

from Figure 4 because we are interested to model the run-time behavior of the system, not its static structure.

The LQN model has two client classes: `conf_controller` and `reg_caller`. They are the elements that drive the system, interacting directly with the user. Phase 1 of `conf_controller` models the time between two conference calls, while phase 2 is the time necessary to initiate a conference, to add all the conferees to the conference, to talk and to terminate the conference. Phase 1 of `reg_caller` represents the time the user is away from the system, while phase 2 is the time necessary to connect a callee, to talk and to terminate the call.

`Conf_call_server`, `call_processing_server` and `db_server` are intermediate servers. Each of them is a multi-server with multiple instances running on the same CPU, and sharing the same queue. Each of them can provide different types of services to their clients, and may need in order to do so the services offered by other servers.

Hardware resources are represented as leaf nodes. We have represented explicitly the disk and the CPUs on which the servers are running. Our concern is the use of computer resources for the set-up of a call, not for the talk part itself. For this reason, we have represented the talk part by delays, `conf_talk_delay` and `reg_talk_delay`, respectively. We haven't represented in detail the communication resources and the terminals, but their effect is integrated in the talk delays. Both `conf_talk_delay` and `reg_talk_delay` are represented as infinite servers.

The execution path in Figure 7 represents the behavior of the system when adding a conferee to a conference call. It shows the same behavior as the MSC from Figure 4, except

Figure 7: LQN performance model of the ACCS

Figure 8: Execution path for adding a conferee

that here we also represent the visits to the hardware resources. The execution path is as follows: in order to add a conferee to the conference call, `conf_controller` invokes the function `add` of `conf_call_server`. To run it, `conf_call_server` uses the `cc_cpu`. Then `conf_call_server` has to call the function `conf` (`conf-query`) of the `db_server` which in turn uses the disk and the `db_cpu`. Upon receiving the reply from `db_server`, `conf_call_server` uses the function `ccon` (`conferee connect`) of the `call_processing_server`, which needs in turn the `cp_cpu` and sends a connect request to the conferee. The conferee is thus added to the conference call.

The parameters for the LQN model were determined by identifying all the methods executed along the path, and by aggregating accordingly the values for the execution times and visits ratios used in the simulation model presented in [Liu94]. This aggregation has been done with the help of the TimeBench tool [FHM95]. The top level of the ACCS system is presented without aggregated parameters in appendix B1 and with aggregated parameters in appendix B2. Each TimeBench task has a more detailed internal structure, not shown here due to space limitations.

3.3.5 Experimental Results

In this section we are analyzing the LQN model from Figure 7, using its results to draw conclusions about the ACCS performance and to find ways to improve it. The solutions for the performance model were obtained with the analytic LQN solver described in [FHMN95]. The analytical results were within 5% of the simulation results from [Liu94].

The first set of experiments has the goal to identify the system bottleneck. We will observe how the system behaves under an increased load, until it reaches saturation. The system component that reaches saturation first becomes the bottleneck. This may be a software or a hardware component, and the methods to relieve the bottleneck are different in each of these cases. We will study as an example, how the response time for the operation “add a conferee” grows in function of the total number of users in the system. The other operations related to the conference call have a similar behavior. (In all the experiments we have maintained a fixed ratio of conference calls to regular calls of 1:10).

During our preliminary experiments we have observed that *conf_call_server* is the system bottleneck. In [NWPM95] it was shown that the number of software server instances has a strong impact on system performance, especially for highly utilized servers. Intuitively, a low number of server instances available to serve a large number of requests leads to longer queues at the software servers, while the servers below may be under-utilized. If we increase the number of software server instances, even if they run on the same CPU, the queues are “pushed-down” to lower level servers and the performance is improved, until these other servers begin to saturate. This effect is illustrated in Figure 9, where the response time for “adding a conferee” is shown for a different number of server instances (parameter *cc*) as a function of system load. (All the server instances are running on the same processor *cp_cpu*). Note that the system load consists of *Nr* regular callers and *Nr/10* conference call groups, each group having 5 conferees in average. (On the x-axis in Figure 9, 11 and 12 is represented only *Nr* for simplicity reasons.) In order to keep down

the number of experiments, we have fixed the number of instances of the other software servers: 8 for *call_processing_server* and 4 for *db_server*.

In all the curves the response time is almost constant for a small load, but increases

Figure 9: Response time for “Add a Conferee” in function of the system load

sharply when the load is high. As we increase the number of *conf_call_server* instances, the knee of the curve shifts to the right. For *cc=16* we get good response times even for high numbers of simultaneous conference calls.

The steep aspect of the curves is produced by the saturation of one or more of the resources when its utilization approaches unity (known as bottlenecking). By analyzing the experiment results we detect that *call_processing_server* is indeed the software bottleneck because it has the highest utilization for high loads while the underlying servers are not

highly utilized. As Figure 10 illustrates, the utilization per instance of *call_processing_server* is almost 100% in the case of just one instance, while *cp_cpu* on which the *call_processing_server* runs is only 50% utilized.

As mentioned before, a way to improve the system performance in the case of a software bottleneck is to increase its number of instances. By doing so, the utilization per *call_processing_server* instance will decrease while the utilization of the *cp_cpu* resource on which all *call_processing_server* instances are concurrently running increases. After the cross over point between the utilizations of *call_processing_server* and *cp_cpu*, it is *cp_cpu* that limits the performance of the system. From Figure 10 it is also evident that *cc_cpu* has a very low utilization, while the utilization of *db_cpu* increases with the number of *call_processing_server* instances, which means that *db_cpu* will become the next system bottleneck if the present one is removed.

After increasing to the limit the number of server instances and moving the problem to the CPU level, the bottleneck problem can be solved by a hardware solution. It has to be decided on which CPU to add more processing power by either replacing it with a faster CPU or by adding another CPU. Figure 11 illustrates a well known fact: if we add more processing power to a non-saturated server (in this case, *cc_cpu*), there is no improvement in the response time; on the other hand, if we add the extra processing power to *cp_cpu*, the response time improves considerably. When deciding where to add the extra processing power we have to find out which is the most highly utilized CPU because it is the one that limits the performance of the system. If we add the extra processing power to a CPU with

Figure 10: Utilization of various software and hardware servers for an increased number of *call_processing_server* instances

a lower utilization we do not have notable improvements in the performance of the system, and the expense does not pay off.

Instead of the hardware solution applied before, we can find a software solution to relieve the bottleneck. For example, we can optimize the code corresponding to different servers, making it faster. Again it is important to identify where the optimization is necessary. If we optimize the code for *conf_call_server* (not a bottleneck) such that the service time requirements have been reduced to 80%, we can see from Figure 12 that there are no gains in the response time for adding a conferee. On the other hand, if we reduce to 80% the execution time of the software bottleneck, *call_processing_server*, the knee of the curve shifts to the right notably. The explanation behind this behavior is that when we

Figure 11: Hardware solution to the system bottleneck

improve the service times for *conf_call_server*, the system bottleneck is still *call_processing_server*. Requests are sent to *call_processing_server* at a higher speed. As the processing power of this server has not changed, the requests will spend more time in the queue waiting for service, and the performance remains almost unchanged. On the other hand, if we reduce the service time for *call_processing_server*, the requests sent by *conf_call_server* will be served at a higher rate and their queueing time will be substantially reduced. Note that the software solution is very efficient: a reduction by 20% of the execution time at the software server which is the bottleneck brings the same performance gain as the addition of a second CPU.

Figure 12: Software solution to the system bottleneck (code optimization)

This chapter presents an approach to build performance models for distributed and concurrent OO systems that can be applied during the development process of such systems. The analytic modeling technique used is the Layered Queueing Network, an extension to the well known Queueing Network model. LQN is more appropriate than QN for modeling OO systems because it allows for nested service requests, which represents well the way the objects communicate.

By applying LQN to an example system (a distributed conference-call) it is shown how to build an LQN model and, most importantly, how to use the model to explore the overall system performance, to identify bottlenecks and to devise improvements.

Chapter 4

Measuring Tools and Techniques

This chapter describes the different measuring tools and techniques that were used to measure the performance of C++ programs in this thesis.

4.1 Quantify

The performance of an application is affected by the many interactions between its own code, third-party libraries, operating system and hardware, networks and other processes, and therefore, it is often difficult to understand why a program takes a certain time to execute. Quantify [Qua] is a software tool used to measure the execution time of C and C++ sequential programs.

Quantify provides accurate measurements for code that can be compiled under the tool, and less accurate for that code for which the source code is unavailable (like operating system calls, for example).

Available source code. Quantify uses both information provided by the compiler, and information collected at run time.

The compiler divides the code in basic blocks. A basic block is a sequence of instructions that are executed always together in succession. For each basic block, Quantify uses the compiler output and calculates its execution time based on the number of

instructions it executes, the number of machine cycles each instruction requires and the machine's clock rate (typically expressed in millions of instructions per second). Quantify determines the machine type and clock rate at instrumentation time.

Run time information gathered by Quantify regards the number of times every basic block is executed. By multiplying this number with the number of machine cycles necessary to execute the block it obtains the total execution time.

Each function records both the time spent in its own code (the function time) and the time spent in functions that it called (the descendant's time).

Performance data can be collected over the entire run or only for certain code fragments of interest. It can be done at different granularity levels: line, basic block or function (with or without descendants).

There is a limitation due to the use of compiler output: the performance impact of memory caching cannot be predicted.

Non-available source code (for example, system calls and third-party libraries). A system call is executed whenever a program requests a service from the operating system. Each system call switches the processor from user state to kernel state in order to process the specific request, and then back to user state after completion of the required service. In order to measure the time necessary to execute the system calls, Quantify identifies and inserts a code sequence around each system call. It has two options for timing system calls, by measuring the *elapsed time* (wall-clock) or the *kernel time*. The measured time is

converted to machine cycles based on the machine's clock rate and then added to the function making the system call.

In the first option, Quantify uses calls to *gettimeofday* to measure the elapsed time for a system call. If a context switch happens during Quantify's request to *gettimeofday*, the measured time of the operating system call appears longer than it actually was because Quantify can not detect that a context switch occurred.

In the second option, Quantify measures the change in user and system time recorded by the kernel for the process, using *getrusage* calls. The measurement reported consists of the amount of time the kernel has spent to service the program, without any time the program or the kernel waited for other processes before the program could be executed. However, the use of *getrusage* does not provide perfectly accurate results, as shown in the next section.

4.2 Getrusage system call

Getrusage is a system call that returns information about the resources utilized by the current process, among others about the CPU time spent executing in user mode and in system mode. This information is collected by the operating system by sampling the state of the system at every tick of the clock, and its accuracy depends on the clock resolution. Therefore, the measurements are more accurate for longer chunks of code and less accurate for shorter ones.

There are two main alternatives to measuring the CPU demand for a short chunk of code [Vet95], as well as a combination of the two.

The first alternative is to measure the CPU demand of a large number of replications of the code chunk and then calculate the average. This is the most common way to instrument the source code. The disadvantage of this approach is the necessity to have consecutive replications of the code chunk that is measured, which might be impossible in a real application.

E.g.:

```
getrusage (start);  
replicate chunk m times;  
getrusage (end);  
calculate average;
```

The second alternative is to measure the CPU demand for every execution of the code chunk. The code chunk is then executed enough times to get a statistically significant average. This approach offers also the possibility of analyzing the distribution of CPU times. However, it has the disadvantage that it can be used only when the CPU demand for the chunk of code is bigger than the resolution of CPU timers. The resolution of CPU timers is 10 ms on a typical UNIX system, and less than 100 microseconds on a Solaris system. If CPU accounting is implemented by sampling, a large number of samples (order of 10,000) will still produce a reasonably accurate average CPU demand.

E.g.:

```
loop n times  
{  
    getrusage (start[n]);  
    code chunk;  
    getrusage (end[n]);  
}
```

calculate average;

A hybrid alternative which combines both previous alternatives is to repeat enough times measuring the CPU demand for a large number of replications of the code chunk. It has the advantage that it requires less replications than the first alternative. The number of necessary replications is chosen in such a way that the quantity measured (the replicated code) is bigger than the overhead of *getrusage* itself. The replicated code chunk is then executed enough times to obtain a statistically significant average.

```
E.g.:          loop n times
                {
                getrusage (start[n]);
                replicate code chunk m times;
                getrusage (end[n]);
                }
                calculate average;
```

There are a number of pitfalls related to using *getrusage* for measuring the CPU demand of a code chunk:

- *Insufficient resolution of the machine is affecting the measurements.* On each tick of the system clock, the CPU state is examined. If the processor is in user mode, the current process is charged with one sampling interval of user time. Similarly, if the processor is in system mode, the current process is charged system time. Implementations of the UNIX operating system provide coarse-grained, statistical measurements of CPU utilization. For example, a process with periodic behavior that runs in phase with the system clock might always surrender the CPU before

the clock interrupt, thereby accumulating no usage time. The same thing might happen if the CPU demand for the process is less than the resolution of the system clock. The first case can be solved by repeating the measurements 10,000 to 100,000 times to produce a reasonably accurate average. A solution to the second problem is to measure the replicated code chunk, where the number of replications is chosen in such a way that the replicated code is bigger than the CPU resolution.

- *Instrumenting overhead has been forgotten.* When the code chunks are small, the overhead of the starting/stopping of timing, the loop control and the overhead of `getrusage` itself may be significant. Experience shows that it might be necessary to subtract the resource demand of one timing call (`getrusage`) from the measured resource demand in order to estimate the CPU time of the code chunk. The resource demand for `getrusage` itself can be measured by using the first approach for, let's say, 1,000 replications of the system call. Measuring the overhead of a loop implemented by a *for* is very difficult because of compiler optimizations.
- *Competing processes may distort measured CPU demand.* The measurements are load dependent. It is hard or even impossible to control the load in the system in order to obtain reliable measurement data.

As it was mentioned before, the second and third alternatives can be used to analyze the distribution of CPU demand times. By representing with the help of a histogram the distribution of the CPU demand for a code chunk in the case of a certain number of measurements, those results that are completely out of a reasonable range can be discarded.

Furthermore, the interval where most of the values lie can be analyzed for a finer-grained distribution of the values.

Figure 13: Coarse granularity histogram for getrusage results

Figure 13 and 14 illustrate the results obtained by measuring repeatedly 1,000 times a code chunk consisting of 500 replications of a function call. Even if the measured quantity has a deterministic distribution, the results obtained with getrusage are not deterministic. 2.5% of the measurements shown in Figure 13 were discarded, being too much out of range (over 0.65). Figure 14 shows a finer granularity histogram for the remaining 97.5%.

This example illustrates that getrusage results have to be taken into consideration very carefully. This applies also to measurements obtained with Quantify when getrusage is

Figure 14: Fine granularity histogram for getrusage results

invoked for timing system calls. It is worth mentioning that Quantify User's Guide [Qua] never mentions the accuracy problems due to getrusage.

Chapter 5

OO Related Execution Overheads

This chapter studies and measures the execution costs of some of the most important OO features found in C++ sequential programs, like object creation/destruction, simple and polymorphic function calls, implementation of object relationships. Expensive overheads are brought to the attention of the reader and run-time saving tips are given throughout the chapter. The execution costs of OO features come into play when mapping the components of the OO programs into LQN elements, as shown in Chapter 3.

An important limitation of the scope of the thesis must be mentioned here: due to time constraints we have not considered execution overheads that appear when the C++ objects are allocated to more than one operating system process or are distributed among different processors.

5.1 General OO Concepts in C++

A program in C++ is built of a collection of cooperating objects. Every *object* is characterized by its state (its properties or attributes and their values), its behavior (the actions that can be performed on the object), and its identity (the name that distinguishes it from other objects).

The state of the object is represented by its *member data*. The behavior of the object is represented by its *member functions*, called sometimes also *methods*. All objects are distinguishable and have their own identity.

A *class* is a set of objects with similar state and behavior. An object is an instance of a class.

Access to the member data and functions of a class can be private or public. Member data and functions that are *private* can be accessed only within the class. Member data and functions that are *public* can be accessed from outside the class.

Another important OO feature in C++ is *inheritance* by which a hierarchical arrangement can be imposed on classes so that some of the properties of one class can be inherited from another one [DH95].

Polymorphism is the ability to use the same name for different actions on objects of different classes. Polymorphism is implemented by virtual functions.

As already mentioned, an OO program is composed of a collection of cooperating objects. The relationships between these objects can be either whole/part relationships or associations (as described in Section 5.4.2 and Section 5.4.1). These relationships are “traveled” at runtime when an object invokes a method of a related object, and therefore the way the relationships are implemented will affect strongly the execution time of the program.

C++ has some provisions for performance enhancement like inline functions and friend functions. *Inline* functions are functions preceded by the keyword *inline*. The call to the

inlined function is replaced by the compiler with the corresponding function code body, thus eliminating the function call overhead. The compiler may decide to ignore the inline directive in some cases (recursive or complicated functions). *Friend* functions have access to non-public members of the class granting friendship, without being function members of that class.

The effect on performance of using inline and friend functions was not studied in this thesis due to time and space constraints. We have concentrated the effort on studying more fundamental C++ features such as object creation/destruction, member function call, and implementation of object relationships.

All the measurements in this chapter have been done using Quantify on an Ultra Sparc machine with a CPU clock speed of 143 MHz. The results of the experiments are expressed in machine cycles and microseconds.

5.2 Constructors and Destructors

One of the most important OO overheads is related to the creation/destruction of objects.

Each class in C++ has to provide a special member function, called the class *constructor*, which is used to initialize an object when it is created. A constructor can be overloaded corresponding to initialization of objects from different kinds of information. Functions from the same scope are said to be *overloaded* if they have the same name, but different formal argument types.

The creation of an object follows two steps [Pap95a]:

- 1) Memory allocation.
- 2) Memory initialization.
  - a) Initialize base class object.
  - b) Initialize all member objects.
  - c) Execute constructor body.

Step 1 is always executed before step 2. Step 1 depends on the storage type of the objects (static, automatic or dynamic). Their performance costs vary largely, as shown in Section 5.2.1. Substep 2.c is always executed. Substep 2.a is executed in the case of inheritance (see Section 5.2.5) and substep 2.b in the case of composition (see Section 5.2.3).

Every class contains at most one *destructor* member function. If not provided explicitly by the programmer, the compiler generates a default destructor. The destructor executes upon object termination. It returns the resources held by the object. A user defined destructor has to be provided for the classes that use pointers and memory allocation at run-time.

The destruction of an object follows two steps:

- 1) Memory deinitialization.
  - a) Execute destructor body.
  - b) Deinitialize all member objects.

c) Deinitialize base class object.

2) Memory deallocation.

Objects are dismantled in the exact reverse order in which they are build. Step 1 has three substeps. Step 1.a is always executed. Step 1.b is executed in the case of composition and step 1.c in the case of inheritance. Destructor calls are always implicit, executed by the compiler.

From the measurements on different examples we have observed that the destructor and constructor execution times for the same objects are of the same order of magnitude.

Performance tip: Remember to account for the execution time of all the constructors and destructors invoked when estimating the execution time of a C++ fragment. It is very easy to overlook especially the performance costs of the destructors, since they are never called explicitly by the user, and in many cases not even defined explicitly in the program.

5.2.1 Static, Automatic and Dynamic Creation of Objects

An object can have automatic, static or dynamic persistence. An *automatic* object can be created at any point in the program when its declaration is reached and its constructor is executed, and terminates automatically at the end of the scope in which it was declared (when its destructor is implicitly invoked). A *static* object is created when the program starts and is terminated automatically when the program finishes execution. A *dynamic*

object can be created anywhere in the program, and is terminated under the programmer's control.

The three types of object declarations are as follows:

```
static A a;           // creates a static A object
```

```
A b;                 // creates an automatic A object
```

```
A *ptrA = new A;     // creates a dynamic A object addressed by ptrA
```

A dynamic object is created with *new* and has to be terminated explicitly by a call to *delete*. Two things happen when *new* is called: first, memory is allocated, and then one or more constructors are called. A call to *delete* first calls one or more destructors and then deallocates the memory.

A programmer has to be aware of the fact that run-time object creation (for dynamic objects) is much more expensive than compile-time object creation (for static and automatic objects), because of the system calls *new* and *delete*.

For example, let us consider the case of a class A that has both a constructor and a destructor with empty body and no member data:

```
class A  
{  
    public:  
        A() {}           // constructor  
        ~A() {}          // destructor  
}
```


The creation of an automatic class A object takes 5 machine cycles (or 0.034 microseconds): 3 machine cycles to call the constructor and 2 to execute it. On the other hand, the creation of a dynamic class A object takes 471 machine cycles (or 3.2937 microseconds).

5.2.2 Copy Constructor vs. Assignment Operator

A class should also provide copy functions besides the class constructor(s) and destructor member functions, which implement user-defined object copy. There are two kinds of such functions: the copy constructor and the assignment operator, declared as follows:

```
A (const A&);                                // copy constructor  
  
A& operator= (const A&);                    // assignment operator
```

A copy constructor is a member function that initializes an A target object from an existent source A object. An assignment operator is a member function that assigns to an A target object from another existent source A object.

If a class definition does not declare explicitly copy constructor and assignment operator functions, the compiler generates default copy functions if needed.

A copy constructor generated by the compiler performs member by member copy initialization. Each target object data member is copy initialized with the source object data member of the same type. Similarly, an assignment operator generated by the compiler

performs member by member assignment. Each target object data member is assigned with the source object data member of the same type.

A default copy constructor or assignment operator can corrupt a program in case of classes with a data member that is a pointer to run-time allocated memory. The default copy constructor does bitwise copying, thus copying the pointer, but does not allocate new memory for the new object, as shown in Figure 15. The two objects are not completely independent, as they were meant to be. To avoid such problems it is a good practice to override the default copy functions.

Figure 15: Default copy constructor copies the pointer, but not the allocated memory

An assignment operator is executed when it is called explicitly by the programmer. A copy constructor can be executed explicitly at an explicit initialization or implicitly, when passing a function argument by value or when returning an object from a function by value. These implicit compiler calls to the copy constructor can have performance impacts the programmer may not be aware of.

5.2.3 Creation/Destruction of Composed Objects

An object can have as member data other user defined objects. In the following example, an object of class B will contain an object of class A.

```
class A                // component class
{
    ...
}

class B                // composed class
{
    ...                // other data members
    A a;               // object data member
    ...
}
```

Creation of an object of a composed class follows the same two steps: memory allocation and memory initialization. Memory initialization is in turn a sequence of two substeps: the initialization of all member objects and then the execution of the constructor body. If a member object has member objects of its own, then the process continues recursively. Only after all the member objects have been initialized, the constructor body of the composed class is executed. Basically, the compiler generates calls to the constructors of the member objects of a composed object, before executing the body of the constructor of the composed class.

The performance overhead for the construction of composed objects can grow up quickly since it includes also the time necessary to create all the member objects.

For example, let us consider the case of an object *Person* that is composed also of two *String* objects:

```

class String
{
private:
    char* data;
public:
    String (const char* value = 0);           // default constructor
    String (const String& x);                 // copy constructor
    ~String ();                             // destructor
    String& operator=(const String&);         // assignment operator
    ... // other member functions
};

class Person
{
private:
    String name, address;                   // member objects
public:
    Person (const String& strName, const String& strAddress)
        : name (strName), address (strAddress)
    {}                                       // constructor
    ~Person() {}                           // destructor
};

```

The time necessary to construct a *Person* object with the following declaration:

```
Person p(String("Smith"), String("Ottawa"));
```

is 732 machine cycles (or 5.12 microseconds). It includes the time necessary to construct the two *String* member objects.

5.2.4 Initialization List vs. Assignment in Constructor Body

Initialization of the member data of a class can be done in a constructor in two different ways either by using an initialization list, or by assignment in the constructor body.

The difference between the two approaches consists in the number of function calls executed for initialization, as well as the type of the function called (copy constructor, or default constructor and assignment operator).

Let's take into consideration the case of a composed object. Even before entering the body of the constructor, the member objects are initialized. If an initialization list is used, then the member objects are created and initialized by a compiler generated call to the copy constructor of each member object, and then the constructor body is executed. If the member data of the composed object are assigned in the constructor body, then even before entering the constructor body, the member data objects are created using compile generated calls to their constructors. In the body of the composed class constructor assignment to member data is done by using the assignment operator.

In the next example, there are presented the two approaches for implementation of the constructor of the *Student* class. (Of course, only one of them should be provided.)

```
class String {...}
```

```
class Student
```

```
{
```

```
  private:
```

```
    String name;
```

```
    int age;
```

```
  public:
```

```

    Student (const String& strName, int iAge); // constructor
    ~Student() {}
};
Student::Student (const String& strName, int iAge)
    :name(strName), age(iAge)
{} // constructor uses initialization list

```

or:

```

Student::Student (const String& strName, int iAge)
{
    name = strName; // constructor uses assignment
    age = iAge;
}

```

The member functions called by the compiler to construct a *Student* object in the first case:

- the *String* copy constructor to initialize the *String* member data of the *Student* object;
- the (empty) body of the *Student* constructor is executed.

The member functions called by the compiler to construct a *Student* object in the second case:

- the *String* default constructor to construct the *String* member data of the *Student* object;
- the body of the *Student* constructor is executed where a default assignment operator is called to assign a value to the *String* object created previously.

The measurements show a better result in the first case -- 190 machine cycles (or 1.33 microseconds) -- compared to 331 machine cycles (or 2.31 microseconds) in the second case.

Performance tip: In case of composed objects prefer an initialization list to assigning members in the body of the constructor by assignment operators. Always use initialization list for const and reference members.

5.2.5 Constructors in a Hierarchy

Inheritance allows the derivation of new classes based on old ones. The new class, called *derived* class, inherits properties from the old class, called *base* class. By inheritance we can add functionality to an existing class, we can add data members or we can modify inherited member functions. A class inherits all the members of its base class, with the exception of constructors and destructors.

A derived class can be in turn a base class for another derived class. The levels of an inheritance hierarchy are limited only by memory constraints. The derived class is a *specialization* of the base class. An inheritance relationship is also called a *generalization-specialization* relationship.

An *abstract base class* is a class that defines some virtual functions which remain undefined in the base class, and will be defined, possibly differently, in different derived classes. There can't be created any instances of an abstract class.

As it was shown in Section 5.2, step 2.a of the memory initialization step is executed only in the case of inheritance. In order to create an object in an inheritance hierarchy, the compiler has to call first the constructor of the base class to initialize the base class object. This is a recursive process if the base class in turn is derived from another class. Only then the constructor body is executed. Thus each constructor that is called adds its own overhead to the total execution time of the constructor for the derived class.

Rumbaugh suggests in [RBPEL91] that a hierarchy with 2-3 levels is acceptable; one with 10 levels is probably too deep; one with 5-6 levels may or may not be good.

Let us consider the case of an inheritance hierarchy with n levels where each class A_n has no member data, but only constructor and destructor member functions with empty body. To create an A_n object all the constructors for the objects A_{n-1} to A_0 have to be executed before the body of the constructor for A_n can be executed. In this case the time necessary to execute the A_0 abstract base class constructor is 8 machine cycles (or 0.06 microseconds) while all the other constructors are 13 machine cycles (or 0.09 microseconds). The time to execute A_n constructor = time to execute abstract base class A_0 constructor + n * time to execute each of the other A_1 to A_n constructors. In the case that $n=10$, then the time to create an A_{10} object is 138 machine cycles (or 0.96503 microseconds).

The process of destructing an object from an inheritance hierarchy is based on the model presented in this section. After executing the constructor body, the base class object

is deinitialized. This process is recursive if the base class object is in turn derived from another base class object.

The measurements for destructors in the previous example show the following results: the time necessary to execute an A_n to A_1 destructor is 21 machine cycles; the time necessary to execute the A_0 abstract base class destructor is 17 machine cycles. Thus the time necessary to destroy an A_{10} object is 227 machine cycles (or 1.59 microseconds).

Because the call to an object destructor is inserted automatically by the compiler, it is important not to forget to take into consideration the overhead for destructing the object, as well as the fact that it incorporates also the overhead for the destructor of its base class object.

Performance tip: Do not use class hierarchies with too many inheritance levels because the execution cost of the constructors (destructors) grows approximately linearly with the number of levels.

5.3 Member Function Calls

The overhead of member function invocation depends on the number and especially the type of parameters and returned results, as shown below.

5.3.1 Call by Value or by Reference

Any C++ function (member function or free standing function) can accept objects as parameters and/or return an object. Sending an object to a function or returning an object from a function can be done by value or by reference.

It is preferable, whenever it is possible, to send and return an object by reference. By sending to or returning an object by value from a function, the overhead is considerably larger because of the implicit calls to the copy constructor. By sending or returning an object by reference no calls to the copy constructor have to be executed, thus no supplementary overhead is added. In the case of complex objects (for example, composed objects or derived objects) the performance cost of executing supplementary copy constructors can be high, as shown in the previous sections.

Let us consider a short example with all the possible ways in which an object can be sent to a function and returned:

```
class A
{
public:
    A() {}           // constructor
    A (const A & x) {} // copy constructor
    ~A() {}         // destructor
};

// functions that accept and return objects by value or by reference

A callByValReturnByVal (A x)
{
```

```

    return x;
}

A& callByRefReturnByRef (A& x)
{
    return x;
}

A callByRefReturnByVal (A& x)
{
    return x;
}

int main()
{
    A a;                // create an A object
    callByValReturnByVal (a);
    callByRefReturnByRef (a);
    callByRefReturnByVal (a);
    return 0;
}

```

In a *callByVal* a copy constructor is called to copy the actual parameter into the formal parameter, before the function is entered. In the function, a destructor is called to destruct the object sent as parameter. A destructor is very expensive. In a *returnByVal* the object returned by the function is going to be destroyed by an implicit call to the destructor in *main*.

The steps executed when calling each of the three functions from *main* are shown below:

1) callByValReturnByVal (a);

- a) before executing the function, the copy constructor is called by the compiler to copy the actual parameter *a* into the formal parameter *x*;
- b) control is given to the function *A callByValReturnByVal (A x)*;
- c) in the function the copy constructor is called to copy the value of *x* into a temporary object that is going to be returned by the function;
- d) the destructor is invoked automatically by the compiler to destruct *x*;
- e) the object returned by the function is going to be destroyed by an implicit call to the destructor before *main* ends;

2) callByRefReturnByRef (a);

- a) no constructors or destructors are called before entering the function, or in it;

3) callByRefReturnByVal (a);

- a) control is given to the function *A callByRefReturnByVal (A& x)*;
- b) in the function the copy constructor is called to copy the value of *x* into a temporary object that is going to be returned by the function;
- c) the object returned by the function is going to be destroyed by an implicit call to the destructor before *main* ends;

The measurements show that each constructor and copy constructor take 2 machine cycles (or 0.01 microseconds). The destructor takes 11 machine cycles (or 0.08 microseconds). The cost of calling and executing each of the three functions is 36, 9, and

15 machine cycles respectively (or 0.25, 0.06, and 0.10 microseconds respectively). These times do not include the time necessary to destruct the object returned from the function in the cases where return by value has been used. As expected, the best result has been obtained for sending and returning an object by reference (9 machine cycles), while the worst result was obtained for sending and returning an object by value (36 machine cycles).

As the complexity of the objects increases the differences between the best and worst case increase as well. For example, let us take into consideration the next example:

```
class String
{ ... }
class Person
{
    private:
        String name, address;
    public:
        Person();           // parameters omitted for simplicity
        ~Person();
        ...
}
class Employee:public Person
{
    private:
        String employerName, employerAddress;
    public:
        Employee();        // parameters omitted for simplicity
        ~Employee();
        ...
}
```

An *Employee* object is composed of 2 *String* objects. It is derived from a *Person* object, which in turn is also a composed object. The overhead to send and return by reference an *Employee* object is 9 machine cycles (or 0.06 microseconds) because no objects are created or destroyed. The overhead in the case of sending and returning by value an *Employee* object is 1273 machine cycles (or 8.9 microseconds), 141 times larger! The reason is that, in the case of sending and returning by value an *Employee* (composed and derived) object, 12 constructors and 12 destructors are called implicitly by the compiler. The programmer might not even be aware of all these calls.

Performance tip: *It is a good practice to try to avoid whenever possible to send and return objects by value.*

There are some cases when an object has to be returned by value, for example when an object is created within a function, a reference to it cannot be returned by the function because the object is going to be deleted automatically when the function returns and thus the reference will be a name for some non-existent object. For a more detailed explanation see [Mey92]. If a function returns an object passed to it via a reference or a pointer, the object should be returned by reference.

5.3.2 Virtual Functions and Dynamic Binding

A member function in C++ can be invoked in 3 different ways:

- through an object;

- through a class pointer or reference;
- through a base class pointer or reference that refers to the derived class.

By default, in C++ the type of a pointer or reference is used to decide which function to use. The type of the object pointed to or referred to is ignored.

By *static binding* the compiler matches at compile-time a function call to a particular function definition based on the type of pointer or reference.

By *dynamic binding* the program has the responsibility to decide at run-time which function definition to use based on the type of the object to which the reference or pointer refers to.

The mechanism of dynamic binding is implemented by virtual member functions. A *virtual function* is preceded by the keyword *virtual* in the base class declaration. This member function can be redefined differently in different derived classes. All definitions will have the same name and signature as in the base class, but without the keyword *virtual*. (This represents the implementation of polymorphism in C++). The decision regarding the function definition to be used when a function call through a pointer or reference takes place will be taken at run-time based on the type of object pointed to or referred to. Dynamic binding never occurs for a function call through an object.

A *pure virtual function* is a virtual function with a prototype but no definition.

Example: `virtual void display() const = 0;`

If a class declaration contains a pure virtual function, it is called an *abstract base class*. No instances of that class can be created. Only instances of its concrete derived classes can exist.

The virtual functions have some costs. In terms of absolute space costs, every class in the hierarchy will have a table of addresses of virtual functions; every object will have an additional member data -- a pointer to the virtual table. The virtual function table and the pointer to it are created automatically by the compiler. The relative run-time overhead of a virtual function call compared to a statically bound function call is looking up the address of the function to execute in the virtual function table.

Despite the fact that non-virtual functions are slightly more efficient than virtual functions, virtual functions should be used whenever a run-time decision has to be made. The virtual function calling overhead for short functions with only a few statements is negligible in comparison to the function execution time [Pap95a] because the implementation of virtual functions is highly optimized with as many decisions as possible taken at compile-time and registered in the virtual function tables.

As an example, let us consider the following hierarchy:

```
class a0                                // abstract base class
{
    public:
        virtual void display () = 0;    // pure virtual function
        a0() {}                          // constructor
        virtual ~a0() {}                // destructor
};
```



```
class a1 : public a0           // derived class
{
    public:
        void display ();       // virtual function
        a1() {}
        ~a1() {}
};
```

```
class a2 : public a1           // derived class
{
    public:
        void display ();
        a2() {}
        ~a2() {}
};
```

// implementation of the member functions

```
void a1::display ()
{ ... // specific implementation
}
```

```
void a2::display ()
{ ... // specific implementation
}
```

The function `void a1::display ()` can be called in 3 different ways:

- 1) through an `a1` object:

```
a1 o1;           // declare an a1 object

o1.display();
```

- 2) through a pointer to the `a1` derived class:

```
a1 *p11 = &o1;    // declare a pointer to the a1 class
```

```
p11->display ();
```

3) through a pointer to the *a0* base class:

```
a0 *p01 = &o1;    // declare a pointer to the a0 base class and
```

```
// initialize it with the address of an a1 object
```

```
p01->display ();
```

The decision as to which virtual function *display()* to execute when a call to *display()* is encountered can be solved at compile-time for the first two cases. A call to a function through an object is always statically bound. A function called through a pointer to a class is also solved at compile-time as long as the class is not an abstract one. A pointer to an abstract class can point to any object derived from the base class (for example, it could point to an *a2* object). A decision concerning which function definition to use in order to execute the call can be taken only at run-time.

The calling overhead is 3 machine cycles (or 0.0209 microseconds) in the first case and 15 machine cycles (or 0.1049 microseconds) in the last 2 cases. The overhead of a function call is shorter when it is done through an object than through a pointer. It is interesting to note that both calls through pointers have the same overhead although binding has been done at compile-time for case 2 and at run-time for case 3.

5.4 Implementation of an Object Model in C++

An object model describes graphically the static structure of a system. It depicts the classes with their attributes and operations and their relations with other classes.

5.4.1 Attribute and Association

Attributes and associations are fundamental object model concepts.

Attributes are characteristics of object instances. Each object has individual values for its attributes. The attributes of an objects are usually implemented as member data.

An *association* describes a group of connections with common structure and common semantics between objects [RBPEL91]. An association can have some properties, called also link attributes.

In the process of modeling, the decision to view a particular characteristic as an attribute or as an association is based always on context.

Each association has two multiplicities, one for each end of the association. *Multiplicity* of an association specifies how many instances of a class participate in an association with a single instance of the class at the other end of the association. Associations can represent one-to-one, one-to-many, many-to-one and many-to-many relationships.

5.4.2 Aggregation

Aggregation [RBPEL91], known also as *containment* [Sch92] or *whole-part* relationship [CNM95], is a relationship in which an object representing the assembly (or

the whole) is associated with some number of objects representing its components (or parts). It can be implemented by including instances or pointers to instances of a component class as data members of the assembly class. Aggregation is a special type of association, and it is implemented in the same way. Aggregation should be used when the two objects are inter-related and participate in a whole-part relationship, whereas association should be used for independent objects that may be linked together (meaning that one invokes the other's member functions).

5.4.3 Implementation of Association/Aggregation

There are several ways of implementing associations (aggregations) in C++ programs, each offering a different compromise between two contradictory requirements: the generality/reusability of the code and the speed of execution.

Various approaches related to implementing of association/aggregation can be found in [Pap95b], [CN93] or [RBPEL91]. The most general approach we found in [Lin96].

We will compare here different implementation alternatives for two types of associations: one-to-one and one-to-many.

a) One-to-one associations

The simplest way to implement a one-to-one association between two objects is by using pointers. If the association is unidirectional, then only one pointer is necessary to point from one object to the other one. Only the first object is able to locate the other one. If the association is bidirectional then two pointers are necessary, one in each object, to

traverse the association forward and backward. Programmers face challenges when implementing bidirectional associations because of the pointer management problems.

Linenbach offers in [Lin96] a very general solution for a bidirectional one-to-one association. He considers the example of an one-to-one bidirectional association between a company and its board of directors that has performed its duties for a particular period. The one-to-one association called "has" is presented in Figure 16.

Figure 16: One-to-one relationship

The implementation of the association is presented in Figure 17. The left-hand-side (LHS) object in an association is the point of reference; the right-hand-side (RHS) object is what the LHS object sees. Each class participating in the one-to-one association has its own *ASToOneProp* association object that contains:

- a pointer to the LHS object;

Figure 17: Instance diagram of the association implementation

- a pointer to the association object corresponding to the RHS object;
- a pointer to the association properties.

This implementation adds 3 classes to the design: 2 association classes and one association properties class. Implementation is based on a template class for *ASToOneProp*, which accepts 3 parameters: the two classes participating in the association and the class with the properties of the association. The *ASToOneProp* class is derived from *ASToOneBase* template class, which in turn is derived from *ASAssn* base class.

Subclassing can be used to provide reusability of a class that should be reused without a particular association. The class without any associations should be implemented as a stand-alone class, and then a subclass that has as data members association objects should be created.

Analyzing now the execution time necessary to travel such an association when, for example, a Company object wants to call a member function in the BoardOfDirectors object we realize that the implementation from Figure 17 pays a higher performance cost than a simpler implementation where the two objects point directly to each other.

b) One-to-many associations

The programmer faces more difficulties when implementing one-to-many associations. Linenbach offers a very general solution to the problem. He gives as an example the one-to-many relationship between a car and its accessories, presented in Figure 18.

Figure 18: One-to-many relationship

The proposed association implementation is presented in Figure 19. Figure 19 is overlapped with the case study presented in the next section. The car object has its own association object, called *ASToManyProp*, which contains:

- a pointer to the LHS object (the car);
- a list of objects, each of which contains a pointer to the accessory's association object and a pointer to the association properties.

Figure 19: Instance diagram of the association implementation

The accessory object has its own association object, called *ASToOneProp*, which contains:

- a pointer to the LHS object (the accessory);
- a pointer to the association object corresponding to the RHS object (the car);
- a pointer to the association properties.

The proposed implementation is very general because the accessory objects are not “aware” that they are part of a list. These objects do not contain any data for list links. The list is implemented as a separate entity. In such cases list and iterator classes must be implemented first. This approach allows the accessory object to participate in several associations implemented by lists without changing its class definitions. An example of such an implementation where the object does not contain any data for storing the list links is presented in the case study in the next section. If we are more concerned about performance than reusability and generality of the code, than we can provide an implementation where information about association implementation (i.e. list) is combined with private data of objects. An even less general approach is presented in the case study where an attribute has been used instead of an association.

5.4.4 Case Study

This section presents performance measurements on an C++ program implementing a point-of-sale system. The detailed description of the system can be found in [CN93].

Figure 20: Object model of the point-of-sale application

The object model corresponding to the point-of-sale application is presented in Figure 20. The application consists of 24 classes with 302 functions. Only the most important classes have been represented in the figure. Small classes like *Date*, *Time*, *String*, *Boolean*,

2 OF/DE 2

PM-1 3 1/2" x 4" PHOTOGRAPHIC MICROCOPY TARGET
NBS 1010a ANSI/ISO #2 EQUIVALENT

PRECISION^{SM} RESOLUTION TARGETS

SalesDomainClassMap or additional classes as *OrderedList*, *OrderedListElement* and *ObjectTableObjectIterator* used for implementation of aggregation association have been omitted in the conceptual object model.

Figure 21: Conceptual class model

Figure 20 illustrates a detail from the object model presented in Figure 19. A Customer knows a number of Transactions, which is represented in the figure by a one-to-many association. A Transaction contains some number of LineItems which is represented in the figure by an aggregation. Both relationships are implemented as ordered lists of pointers to Transaction objects, and LineItem objects respectively. This way of implementing the relationships introduces two additional objects in both cases: an *OrderedList<Transaction*>* and an *OrderedListElement<Transaction>* between the Customer and the Transaction objects, and an *OrderedList<LineItem*>* and an

Figure 22: Detailed class model

`OrderedList<LineItem>` between the `Transaction` and the `LineItem` objects. This more detailed model is presented in Figure 21. `OrderedList` and `OrderedListElement` are class templates that can be instantiated at compilation time for different types of parameters.

The relationships between a transaction object containing three `lineItem` objects which refer to two product objects are illustrated in Figure 23. The *transaction* has 3 *lineItems* which are implemented as an ordered list of *lineItemListElements*. Every *lineItemListElement* has a *lineItem* and a pointer to the next element in the list. The *transaction* has a pointer to the first, current and last element in the *lineItemList*. It also keeps track of the number of elements in the list, e.g. 3 in this case. Every *lineItem* is

Figure 23: Relationships between objects

associated to a *product*, which does not have to be different from *lineItem* to *lineItem*. This many-to-one association is implemented by a pointer from a *lineItem* to a *product*.

In order to understand better how the execution time is spent in the point-of-sale application, we have measured the time it takes to create a sales transaction, as well as the time necessary to calculate the total of a sales transaction (which has to travel the aggregation between transaction and lineItems and the association between lineItems and products). From Figure 20 it can be seen that the *SalesTransaction* class is a specialization of the *Transaction* class.

Figure 24: MSC for the creation of a salesTransaction object

Figure 28: MSC for calculate_total of a transaction

Performance tips:

- *Reduce the number of objects in a C++ program because the creation/destruction process is expensive.*
- *Minimize the links representing associations and aggregations; fewer objects on the paths leads to faster execution time.*
- *Compromise between the generality/reusability of the code and the performance costs.*

Figure 28: MSC for calculate_total of a transaction

Performance tips:

- *Reduce the number of objects in a C++ program because the creation/destruction process is expensive.*
- *Minimize the links representing associations and aggregations; fewer objects on the paths leads to faster execution time.*
- *Compromise between the generality/reusability of the code and the performance costs.*

next element in the *lineItem*List. As long as the end of the list has not been reached, the totals of the next *lineItems* in the list are calculated in the same way. The dashed arrows in the figure represent the fact that a friend object accesses the private member data of another object.

The time necessary for a *salesTransaction* to calculate its total is 3.58 microseconds or 512 machine cycles.

If we want to reduce the execution time of this program, we will consider an alternate implementation where some of the additional classes are eliminated.

One way of doing it is to consider the alternative of implementing a relationship as an association or an attribute. Sometimes the distinction between attribute and association is unimportant; other times the distinction is crucial. The decision to implement in a model a particular characteristic as attribute or relationship is based on context.

In the case of the point-of-sale application, one way to reduce the time necessary to perform the operations of creating a *salesTransaction* and to calculate its total is to provide a different implementation for the *Transaction* class by implementing *lineItem* as an attribute of a *transaction* object instead of a different object. In the initial implementation, a *transaction* object has an ordered list of pointers to *lineItem* objects which introduces two additional classes (*LineItem*List and *LineItem*ListElement). A more efficient way to implement a *transaction* is to provide as an attribute in the class instead of a list of pointers to *lineItem* objects, an array of line items. Every element of the array is implemented as a struct and contains all the information that was stored initially in a *lineItem* object. Besides

the array of structs, an index is necessary to keep track of the number of line items in the array. Also a maximum length of the array has to be provided. In this way the need for the *lineItem* object disappears, as well as the two intermediary classes (*LineItemList* and *LineItemListElement*). The new structure of a transaction is presented in Figure 26.

The message sequence chart corresponding to the construction a *transaction* object based on the new implementation is presented in Figure 27. By comparing the MSCs for the two implementations of the *salesTransaction* constructor (presented in Figure 24 and Figure 27) it can be seen that the call to create an ordered list of pointers to *lineItem* objects has been eliminated. Thus the time necessary to construct the *salesTransaction* object based on the new implementation of the *transaction* object has been reduced from 63.02 microseconds (or 9013 machine cycles) to 62.95 microseconds (or 9002 machine cycles).

The advantage of the new implementation is more evident for the calculation of the total of a *salesTransaction* because many calls that introduce their own overheads have been eliminated. The MSC for *calculate_total* based on the new implementation of the *transaction* object is presented in Figure 28. All the calls to the *lineItem* and the intermediary *lineItemList* and *lineItemListElement* have been eliminated. Thus the time necessary to calculate the total of the transaction has been improved from 3.58 microseconds (or 512 machine cycles) to 0.92 microseconds (or 132 machine cycles). The improvement of the performance by almost 4 times is significant.

Figure 26: New implementation of the transaction object

Figure 27: MSC for the new `salesTransaction` constructor

Figure 28: MSC for calculate_total of a transaction

Performance tips:

- *Reduce the number of objects in a C++ program because the creation/destruction process is expensive.*
- *Minimize the links representing associations and aggregations; fewer objects on the paths leads to faster execution time.*
- *Compromise between the generality/reusability of the code and the performance costs.*

Chapter 6

Conclusions

The recent OO literature recognizes that the software development process would benefit considerably from the ability to evaluate and analyze the performance of the system from early stages throughout the whole life-cycle.

This thesis illustrates how an analytical performance model can help the developers to evaluate and study the performance characteristics of an OO system. The OO system from the telecommunication domain presented as a case study has been modeled using a performance model called LQN. The thesis shows how to build and parametrize an LQN model of an OO system based on the basic object model of the system and on the resource requirements of individual objects. It then shows how to analyze the system performance, and how to identify software and/or bottlenecks and how to devise improvements.

The measurements presented in chapter 5 illustrate where and how the time is spent in OO programs.

Based on the measurements and the analysis done, there can be derived some general advice for programmers interested in speeding up their C++ programs:

- Reduce the number of objects as much as possible because the object creation/destruction process is expensive;

- Compromise the generality and reusability of the code in favor of shorter execution times by implementing faster, less general associations aggregations

Future work

The work done in this thesis can be extended in the following directions:

- Include the important issues of concurrency and distribution in the measurements of OO overheads. Consider the additional overheads due to context switching and intercommunication between objects allocated to different processes and processors.
- Study more thoroughly the trade off between code generality-reusability and efficiency in C++ programs.
- Apply the LQN model to a distributed OO application and validate the model predictions against measurements on a real implementation.
- Investigate how performance information can be attached to reusable classes to facilitate the construction of performance models during the development of the system.

References

- [BC95] Buhr, R.J.A. & Casselman, R. (1995). *Use-Case Maps for Object-Oriented Systems*. Prentice Hall.
- [BD93] Burns, A. & Davies, G. (1993). *Concurrent Programming*. Addison-Wesley, Reading, MA.
- [Boo91] Booch, G. (1991). *Object-Oriented Design with Applications*. Benjamin/Cummings.
- [Bri91] Briot, J.-P. (1991). *Object-Oriented Concurrent Programming*. Litp. Universite Paris VI.
- [CN93] Coad, P. & Nicola, J. (1993). *Object-Oriented Programming*. Prentice Hall.
- [CNM95] Coad, P., North, D. & Mayfield, M. (1995). *Object Models. Strategies, Patterns. and Applications*. Prentice Hall.
- [CY91] Coad, P. & Yourdon, E. (1991). *Object-Oriented Analysis*, 2nd edition. Prentice Hall.
- [DH95] Decker, R. & Hirshfield, S. (1995). *The Object Concept*. PWS Publishing Company.
- [FHMN95] Franks, G., Hubard, A., Majumdar, S., Neilson, J., Petriu, D.C., Rolia, J. & Woodside, C.M. (1995). *A Toolset for Performance Engineering and Software Design of Client-Server Systems*.

- Performance Evaluation (special issue on Performance Tools), vol. 24, 117-136.
- [HOOD89a] HOOD *User Manual*. Issue 3.0. WME/89-353/JB. HOOD Working Group. European Space Agency, December.
- [HOOD89b] HOOD *Reference Manual*. Issue 3.0. WME/89-173/JB. HOOD Working Group. European Space Agency, September.
- [Fir93] Firesmith, D.G. (1993). *Object-Oriented Requirements. Analysis and Logical Design: A Software Engineering Approach*. Addison Wesley.
- [JCJO92] Jacobson, I., Christerson, M., Jonsson, P. & Overgaard, G. (1992). *Object-Oriented Software Engineering: A Use Case Driven Approach*. Addison-Wesley.
- [Lav89] Lavenberg, S. (1989). A Perspective on Queueing Models of Computer Performance. *Performance Evaluation*, Vol.10, 53-76.
- [Lin96] Linenbach, T. (1996). Reusable Binary Associations in C++. *Dr. Dobbs's Journal*, Vol. 21, Issue 6, 34-41.
- [Liu94] Liu, J. (1994). *Performance Analysis of Concurrent Object-Oriented Systems*, Masters Thesis, SCE-94-28, Carleton University, Ottawa, Canada.
- [LPC95] Liu, J., Petriu, D.C. & Corriveau, J.P. (1995, August). Performance Modeling of Concurrent OO Systems. *Proceedings of 17th*

International Conference on Technology of Object-Oriented Languages and Systems (TOOLS USA '95), 241-253, Santa Barbara.

- [LZGS84] Lazowska, E.D., Zaborjan, J., Graham, G. S. & Sevcik, K. C.(1984). *Quantitative System Performance*. Prentice Hall.
- [Mey92] Meyers, S. (1992). *Effective C++. 50 Specific Ways to Improve Your Programs and Designs*. Addison-Wesley.
- [NWPM95] Neilson, J., Woodside, C.M., Petriu, D.C. & Majumdar, S. (1995, Sept.). Software Bottlenecking in Client-Server Systems and Rendezvous Networks. *IEEE Trans. on Software Eng.*, Vol.21, No.9, 776-782.
- [OMG92] Object Management Group (1992). *Common Object Request Broker: Architecture and Specification*. John Wiley, New York.
- [Pan95] Pancake, C.M. (1995, October). The Promise and the Cost of Object Technology: A Five Year Forecast. *Communication of the A.C.M.*, Vol.38, No.10, 33-49.
- [Pap95a] Papurt, D.M. (1995). *Inside the Object Model*. SIGS Books.
- [Pap95b] Papurt, D.M. (1995). Automating Association Implementation in C++. *Dr. Dobb's Journal*, Vol. 20, Issue 10, 18-25.
- [PCML95] Petriu, D.C., Corriveau, J.-P., Metea, L.S., Liu, J. (1995). *Analytic Performance Models of Concurrent Object-Oriented Software*. SCE-95-26, Carleton University, Ottawa, Canada. (submitted for publication in the "Theory and Practice of Object Systems" journal).

- [Pet94] Petriu, D.C. (1994, May). Approximate Mean Value Analysis of Client-Server Systems with Multi-Class Requests. Proc. of the 1994 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems, *Performance Evaluation Review*, Vol.22, nb.1, 77-86.
- [PW91] Petriu, D.C. & C.M. Woodside (1991). Approximate MVA from Markov Model of Software Client--Server Systems, *Proc. of The Third IEEE Symposium on Parallel and Distributed Processing*, Dallas, Texas, 322-329.
- [Qua] Pure Software (1995). *Quantify User's Guide*. Pure Software Inc.
- [RBPEL91] Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F. & Lorensen, W. (1991). *Object-Oriented Modeling and Design*. Prentice Hall, Eaglewood Cliffs, NJ.
- [Rol92] Rolia, J.A (1992,January). *Software Performance Modeling*. Ph.D. Dissertation, CSRI Technical Report 260, University of Toronto, Canada.
- [RS95] Rolia, J.A. & Sevcik, K.C. (1995, August). The Method of Layers. *IEEE Trans. on Software Eng.*, Vol.21, No.8, 689-700.
- [Sch92] Schammas, N.C. (1992). *Advanced C++*. Prentice Hall.
- [Smi90] Smith, C.U.(1990). *Performance Engineering of Software Systems*, Addison-Wesley.

- [SW93] Smith, C.U. & Williams, L.G. (1993, July). Software Performance Engineering: A Case Study Including Performance Comparison with Design Alternatives, *IEEE Trans. on Software Eng.*, Vol.19, No.7, 730-741.
- [Tan92] Tanenbaum, A.S. (1992). *Modern Operating Systems*. Prentice Hall.
- [Vet95] Vetland, V. (1995, March). *Advice on instrumentation of small code chunks*. Seminar Notes, Carleton University.
- [Wil93] Wilkie, G. (1993). *Object-oriented software engineering: The professional developer's guide*, Addison-Wesley.
- [WWW90] Wirfs-Brock, R.J., Wilkerson, B. & Wiener, L. (1990). *Designing Object-Oriented Software*. Prentice-Hall, Eaglewood Cliffs, NJ.
- [WNPM95] Woodside, C.M., Neilson, J.E., Petriu, D.C. & Majumdar, S. (1995, January). The Stochastic Rendezvous Network Model for Performance of Synchronous Client-Server-like Distributed Software. *IEEE Trans. on Computers*, Vol.44, No.1, 20-34.
- [Woo89] Woodside, C.M. (1989). Throughput calculation for basic Stochastic Rendezvous Networks. *Performance Evaluation*, Vol.9, 143-160.
- [Woo95] Woodside, C.M. (1995, Sept.). A Three-View Model for Performance Engineering of Concurrent Software. *IEEE Trans. on Software Eng.*, Vol.21, No.9, 754-767

Appendix A

OMT Notation (Rumbaugh).

Class:

Association:

Multiplicity of Associations:

Generalization (Inheritance):

Aggregation:

END

1 810 619 7

FIN