WASHINGTON UNIVERSITY
SEVER INSTITUTE OF TECHNOLOGY
DEPARTMENT OF COMPUTER SCIENCE

DESIGN PRINCIPLES AND OPTIMIZATIONS FOR HIGH-PERFORMANCE, REAL-TIME CORBA

by

Aniruddha S. Gokhale, M.S.

Prepared under the direction of Professor Douglas C. Schmidt

A dissertation presented to the Sever Institute of
Washington University in partial fulfillment
of the requirements for the degree of

Doctor of Science

August, 1998

Saint Louis, Missouri

WASHINGTON UNIVERSITY
SEVER INSTITUTE OF TECHNOLOGY
DEPARTMENT OF COMPUTER SCIENCE

ABSTRACT

DESIGN PRINCIPLES AND OPTIMIZATIONS FOR
HIGH-PERFORMANCE, REAL-TIME CORBA

by Aniruddha S. Gokhale

ADVISOR: Professor Douglas C. Schmidt

August, 1998
Saint Louis, Missouri

Applications and services for next-generation distributed systems must be reliable, flexible, reusable, and capable of providing low latency to delay-sensitive applications (such as avionics, and telecommunication systems), and high bandwidth to bandwidth-intensive applications (such as medical imaging, satellite surveillance, and teleconferencing) running over high-speed networks. Requirements for reliability, flexibility, and reusability motivate the use of object-oriented middleware like the Common Object Request Broker Architecture (CORBA).

However, the empirical studies we conducted measuring the performance of CORBA implementations revealed that current CORBA implementations incur a number of overheads stemming from excessive presentation layer conversions, data copying, and inefficient server demultiplexing techniques.

This dissertation describes our work implementing a high performance, real-time ORB called "The ACE ORB" (TAO). This dissertation provides three contributions to building a high performance, real-time ORB. First, we will describe the optimizations we developed to improve the performance of the IIOP interpretive marshaling engine. Second, we will describe the efficient demultiplexing strategies we developed to reduce latency while improving scalability, predictability and consistency of request demultiplexing. Finally, we describe the design and implementation of the OMG IDL compiler we developed that generates efficient stubs and skeletons.

To the Almighty

Contents

List of Tables

List of Figures

Acknowledgments

A number of people have helped me over the past eight years of my graduate student life in the United States of America. Without their help, I would have never fulfilled my ambition of earning a doctorate degree.

First and foremost, the efforts of my parents in raising the funds necessary for my education in USA enabled me to overcome financially adverse conditions during the first couple of years of my education at Arizona State University. A number of my relatives including my grandparents, uncles, and aunts also helped my parents in raising the funds.

At Arizona State University, the faculty (Dr. Lindstrom, Dr. Mary Laner, and others) and staff (Esther Hardesty, Nancy Rittenhouse, and Sandra Ballistreri) in the Department of Sociology were very kind enough to provide me an on-campus job that enabled me to cover my living expenses. I also thank the Department of Computer Science at Arizona State University for providing me with a teaching assistantship in my last semester at ASU.

I thank Dr. Guru Parulkar, whom I met in Phoenix at a conference that resulted in me getting admission and financial assistance at Washington University for the PhD program.

After a dismal performance during my first year at Washington University that resulted in me losing my assistantship, it was my on-campus job as a programmer at the Electronic Radiology Laboratory (ERL) at the Washington University Medical Campus that ensured my survival. I thank my friend Shreedhar Madhavapeddi for informing me about the job opening at ERL. I thank Dr. G. James Blaine, Steve Moore, and other researchers at ERL for providing me this opportunity and putting faith in me. I also thank the faculty of the Department of Computer Science at Washington University who decided to give me a chance after I passed the PhD Qualifying exams by providing tuition waiver without which I could not have continued my studies.

In the next two years, I worked for Dr. George Varghese and Dr. Ron Cytron. Although at times they were harsh in their judgement about my performance, I now realize the truth in their argument that doctoral research needs commitment, dedication, and hard work. I thank them for their advice and encouragement.

I started working for Dr. Douglas Schmidt since Fall 1995. This was the turning point in my career. I owe my doctorate degree to Dr. Douglas Schmidt. I am honored to be his first doctoral student and I hope to carry on research with the same aggressiveness that I maintained during my doctoral student tenure. I thank the members of Dr. Schmidt's

research group called DOC Group for their support, healthy discussions, their generosity, and the fun-filled atmosphere they maintained in our laboratory.

A large number of friends also provided the necessary support when it was most needed. They include Girish Chandranmenon, Mohan Khadilkar, Niranjan Joshi, Milind Khandekar, Milind Kulkarni, Abhijit Rane, Ajay Apte, Umesh Manathkar, Shreedhar Madhavapeddi, and many others.

I would also like to thank the staff (Jean Grothe, Myrna Harbison, Peggy Fuller, and Sharon Matlock) of the Department of Computer Science for their help and generosity over the last six years I have been at Washington University.

Finally, I thank my wife, Bharati, who provided the necessary encouragement and kept me moving towards my goal. Our daughter, Tanvi, who is now four months old, relieved all the tension as I prepared for my defense. I was very fortunate to have my parents, sister, grandmother, and in-laws at the time of my defense and graduation.

I have dedicated my dissertation to the Almighty. I firmly believe that during my testing times, it was He who created a path that finally led me to achieve this goal.

It is difficult to repay all these people who helped me. I pray to the Almighty to give me the necessary courage and wisdom so that someday I can help others who are in need.

Aniruddha S. Gokhale

Washington University in Saint Louis
August 1998

Chapter 1

Introduction

1.1 Emerging Trends in Distributed Real-time Systems

Applications and services for next-generation distributed systems must be flexible, reusable, robust, and capable of providing scalable, low-latency quality of service to delay-sensitive applications. In addition, communication software must allow bandwidth-sensitive applications to transfer data efficiently over high-speed networks. Robustness, flexibility, and reusability are essential to respond rapidly to changing application requirements that span an increasingly wide range of media types and access patterns [7]. Distributed and real-time applications, such as video-on-demand, teleconferencing, and avionics, require endsystems that can provide statistical and deterministic quality of service (QoS) guarantees for latency [28], bandwidth, and reliability [42]. The following trends are shaping the evolution of software development techniques for these distributed real-time applications and endsystems:

• **Increased focus on middleware and integration frameworks:** There is a general industry trend away from *programming* real-time applications from scratch to *integrating* applications using reusable components based on object-oriented (OO) middleware [38].

• **Increased focus on QoS-enabled components and open systems:** There is increasing demand for remote method invocation and messaging technology to simplify the collaboration of open distributed application components [8] that possess stringent QoS requirements.

• **Increased focus on standardizing real-time middleware:** Several international efforts are currently addressing QoS for OO middleware. The most prominent is the Object Management Group's (OMG) Common Object Request Broker Architecture (CORBA) standardization effort [51].

Communication middleware based on the Common Object Request Broker Architecture (CORBA) [51] is a promising approach for improving the flexibility, reliability, and

portability of communication software. CORBA is designed to enhance distributed applications by automating common networking tasks such as object registration, location, and activation; request demultiplexing; framing and error-handling; parameter marshaling and demarshaling; and operation dispatching. CORBA is OO middleware that allows clients to invoke operations on objects without concern for where the objects reside, what language the objects are written in, what OS/hardware platform they run on, or what communication protocols and networks are used to interconnect distributed objects [76]. CORBA also provides the basis for defining higher layer distributed services such as naming, events, replication, transactions, and security [48].

There has been recent progress towards standardizing CORBA for real-time [50] and embedded [49] systems. Several OMG groups, most notably the Real-Time Special Interest Group (RT SIG), are actively investigating standard extensions to CORBA to support distributed real-time applications. The intent of the real-time CORBA standardization effort is to enable real-time applications to interwork throughout embedded systems and heterogeneous distributed environments such as the Internet.

Although some operating systems, networks, and protocols now support real-time scheduling, they do not provide integrated end-to-end real-time ORB endsystem solutions [69]. Moreover, relatively little systems research has focused on strategies and tactics for real-time CORBA. In particular, QoS research at the network and OS layers has not addressed key requirements and programming aspects of CORBA middleware. For instance, research on QoS for ATM networks has focused largely on policies for allocating bandwidth on a virtual circuit basis [10]. Likewise, research on real-time operating systems has focused largely on avoiding priority inversions in synchronization and dispatching mechanisms for multi-threaded applications [60].

However, despite dramatic increases in the performance of networks and computers, designing and implementing flexible and efficient communication software remains hard. Substantial time and effort has traditionally been required to develop this type of software; yet all too frequently communication software fails to achieve its performance and functionality requirements.

Furthermore, notwithstanding the significant efforts of the OMG RT SIG, however, developing and standardizing distributed real-time CORBA ORBs remains hard. There are few successful exemplars of standard, commercially available distributed real-time ORB middleware. In particular, conventional CORBA ORBs are not well suited for performance-sensitive, distributed real-time applications due to (1) lack of QoS specification interfaces, (2) lack of QoS enforcement, (3) lack of real-time programming features, and (4) general lack of performance and predictability [67].

The success of CORBA in mission-critical distributed computing environments depends heavily on the ability of the ORBs to provide the necessary quality of service (QoS) to applications. Common application QoS requirements include:

• **High bandwidth** CORBA ORBs must provide high throughput to bandwidth-sensitive applications such as medical imaging, satellite surveillance, or teleconferencing systems;

• **Low latency** CORBA ORBs must support low latency for delay-sensitive applications such as real-time avionics, distributed interactive simulations, and telecom call processing systems;

• **Scalability of endsystems and distributed systems** CORBA ORBs must scale efficiently and predictably as the number of objects in endsystems and distributed systems increases. Scalability is important for large-scale applications that handle large numbers of objects on each network node, as well as a large number of nodes throughout a distributed computing environment.

Experience over the past several years [59] indicates CORBA is well-suited for request/response applications over lower-speed networks (such as Ethernet and Token Ring). However, earlier studies [57, 68], and our results shown in Chapters 2, 3, and 4, demonstrate that conventional implementations of CORBA incur considerable overhead when used for performance-sensitive applications over high-speed networks. As users and organizations migrate to networks with gigabit data rates, the inefficiencies of current communication middleware (like CORBA) will force developers to choose lower-level mechanisms (like sockets) to achieve the necessary transfer rates. The use of low-level mechanisms increases development effort and reduces system reliability, flexibility, and reuse. This is a serious problem for mission/life-critical applications (such as satellite surveillance and medical imaging [3, 11]). Therefore, it is imperative that performance of high-level, but inefficient, communication middleware be improved to match that of low-level, but efficient, tools.

1.2 Research Contributions

Section 1.1 described the emerging trends in developing distributed, real-time systems and emphasized middleware-based solutions to implement these. Figures 1.1 and 1.2 depict some examples of performance-sensitive, large-scale, distributed applications. This research addresses the following questions pertinent to developing such applications:

- Can CORBA be used for performance-sensitive applications (*e.g.,* telecommunications, satellite surveillance, medical imaging, avionics) on high-speed networks?
  - We determine this empirically as described in Chapters 2, 3, and 4.

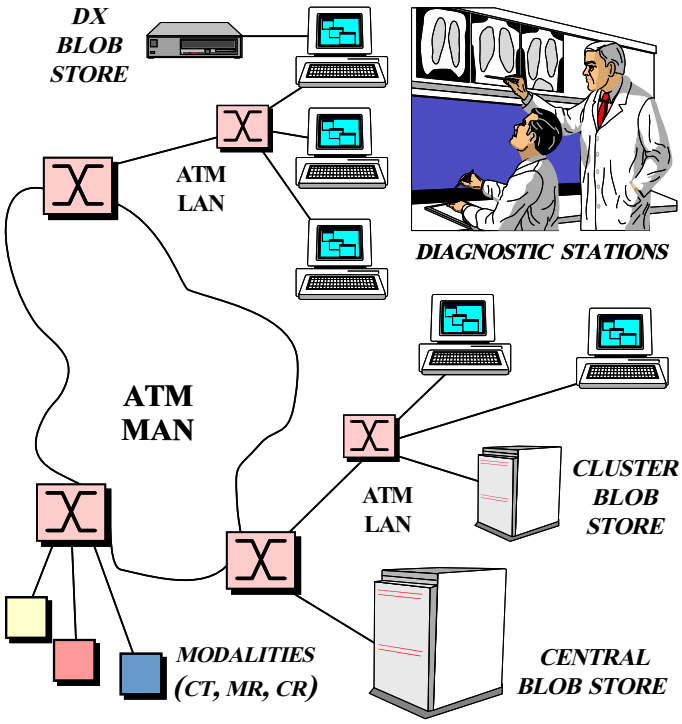


Figure 1.1: Medical Imaging

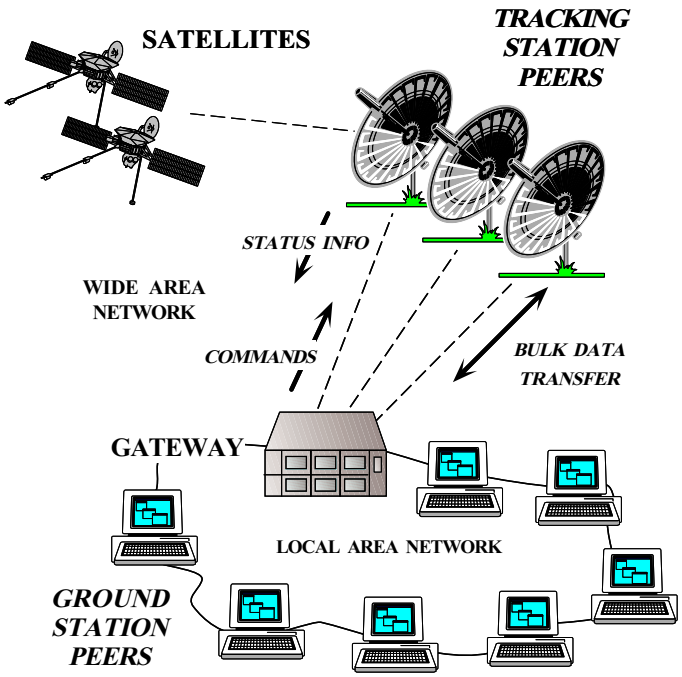


Figure 1.2: Gateway Communication

- Develop optimizations required to build high-performance ORBs with the following constraints:
  - Ability to support very high bandwidth, and low and predictable latency to applications;
  - Maintain strict CORBA compliance.
- Identify features and architectural patterns needed for real-time ORBs, and develop optimized components to support the following:
  - Both hard real-time and statistical real-time applications must be supported.

1.2.1 Limitations of CORBA for Real-time Applications

Prior experience using CORBA on telecommunication [65], avionics [31], and medical imaging projects [58] indicates that it is well-suited for conventional request/response applications with best-effort QoS requirements. However, CORBA is not yet suited for high-performance, real-time applications for the following reasons:

Lack of QoS specification interfaces: The CORBA standard does not provide interfaces to specify end-to-end QoS requirements. For instance, there is no standard way for clients to indicate the relative priorities of their requests to an ORB. Likewise, there is no interface for clients to inform an ORB how frequently to execute operations that have periodic processing deadlines.

The CORBA standard also does not define interfaces that allow applications to specify their admission control policies. For instance, a video server might prefer to use available network bandwidth to serve a limited number of clients and refuse service to additional clients, rather than admit all clients and provide poor video quality. Conversely, a stock quote service might want to admit a large number of clients and distribute all available bandwidth and processing time equally among them.

Lack of QoS enforcement: Conventional ORBs do not provide end-to-end QoS enforcement, *i.e.*, from application-to-application across a network. For instance, most ORBs use a FIFO strategy to transmit, schedule, and dispatch client requests. However, FIFO strategies can yield unbounded priority inversions [60, 40], which occur when a lower priority request blocks the execution of a higher priority request for an indefinite period. Likewise, conventional ORBs do not allow applications to specify the priority of internal threads that process requests.

Standard ORBs also do not control servant execution. For instance, they do not terminate servants that consume excess resources. Thus, most existing ORBs use *ad hoc* resource allocation. Consequently, a single client can consume all available network bandwidth and a misbehaving servant can monopolize a server's CPU.

Lack of real-time programming features: The CORBA specification does not define key features that are necessary to support real-time programming. For instance, the CORBA General Inter-ORB Protocol (GIOP) supports asynchronous messaging. However, no standard programming language mapping yet exists for transmitting client requests asynchronously, though there is one under review by the OMG. Likewise, the CORBA specification does not require an ORB to notify clients when transport layer flow control occurs, nor does it support timed operations [15]. As a result, it is hard to develop portable and efficient real-time applications that behave deterministically when ORB endsystem or network resources are temporarily unavailable.

Lack of performance optimizations: Conventional ORB endsystems incur significant throughput [58] and latency [23] overhead, as well as exhibiting many priority inversions and sources of non-determinism [41], as shown in Figure 1.3.

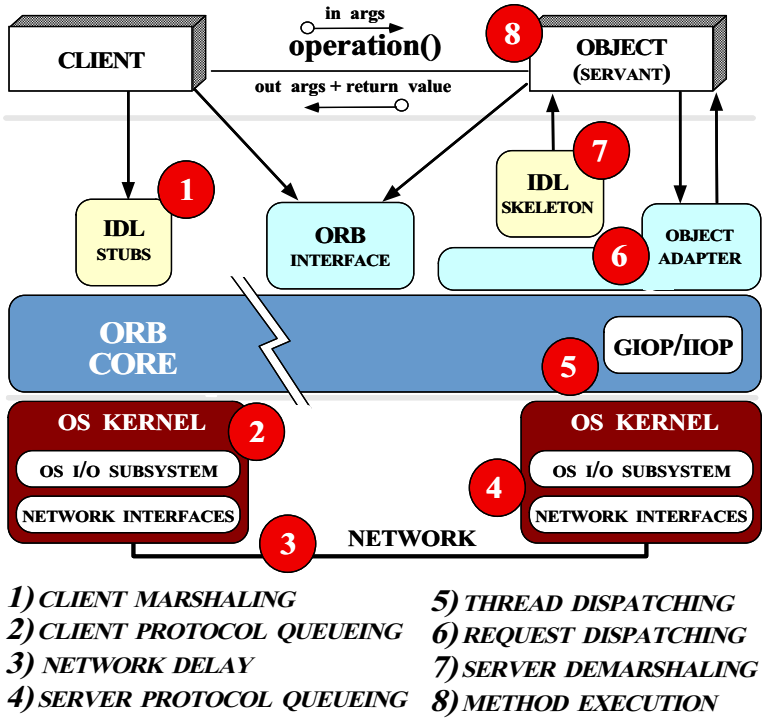


Figure 1.3: Sources of Latency and Priority Inversion in Conventional ORBs

These overheads stem from

1. non-optimized presentation layer conversions and monolithic presentation code that copies data excessively [13] and overflows processor caches [25];

2. internal buffering strategies that produce non-uniform behavior for different message sizes [21];

3. inefficient demultiplexing and dispatching algorithms [24];

4. long chains of intra-ORB virtual method calls [20];

5. lack of integration with underlying real-time OS and network QoS mechanisms [67].

1.2.2 Scope of the Research

The primary contribution of this dissertation has been pinpointing precisely where the key sources of overhead exist in higher-level communication middleware such as CORBA toolkits and developing optimizations to significantly reduce these overheads.

This research employs a measurement-driven approach. The findings from these measurements drive our optimizations. Different components of the CORBA middleware architecture are tested for performance and the sources of overhead are quantified and analyzed. Detailed measurements of throughput, latency, and scalability of existing ORBs appear in our preliminary work [20, 21, 23]. We have used two widely used CORBA implementations - Orbix 2.1 and VisiBroker 2.0 - in our experiments. Our findings indicate that existing ORBs incur significant overhead stemming from a variety of sources including:

1. non-optimized presentation layer conversions, data copying, and memory management;
2. inefficient and inflexible receiver-side demultiplexing and dispatching operations;
3. long chains of intra-ORB function calls;
4. generation of non-word boundary aligned data structures by stub compilers;
5. excessive control information carried in request messages;
6. lack of integration with underlying operating system mechanisms.

Our goal in precisely pinpointing the sources of overhead for communication middleware is to develop scalable and flexible CORBA implementations that can deliver the required QoS to applications [30].

We provide these results and our analysis in Chapters 2, 3, and 4. Based on these analyses, we have developed a number of optimizations to eliminate the different sources of overhead present in existing ORBs [25, 24]. Our optimizations are incorporated in a high-performance, real-time ORB called TAO (The ACE ORB) [69]. The optimizations developed as part of this dissertation and applied to TAO are shown shaded in Figure 1.4.

The optimizations reported in this dissertation include

1. a high-performance, interpretive Internet Inter-ORB Protocol (IIOP)'s Common Data Representation (CDR) marshaling engine - a feature that reduces the latency (described in Chapter 5)

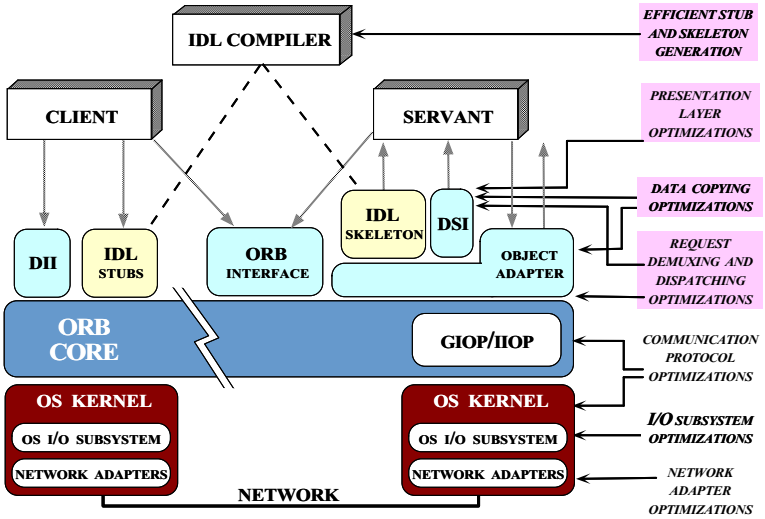


Figure 1.4: Research Contributions

2. a highly scalable request demultiplexing engine whose performance is predictable and consistent - a feature that is highly desirable for large-scale, real-time applications (described in Chapter 6)

3. an OMG IDL compiler back-end that produces efficient stubs and skeletons that use the IIOP CDR marshaling engine (described in Chapter 7)

1.2.3 Impact of the Research

Our research has had a significant impact on the ORB vendors, the industry, and the research community. Some of our findings and solutions have been incorporated in the responses to the Request for Proposals (RFP) [50] submitted by the Object Management Group's (OMG) special interest group on real-time CORBA. In addition, our performance results and the analysis has influenced the ORB vendors to improve the performance of their ORBs in their later releases.

The results of our performance analysis with TAO indicate that the performance of the stubs and skeletons using the interpretive marshaling engine is almost comparable to that of the stubs and skeletons of commercial ORBs. At the same time, the footprint of the stubs and skeletons using interpretive marshaling is significantly smaller compared to that using compiled marshaling as shown in Chapter 7.

1.3 Overview of CORBA

CORBA Object Request Brokers (ORBs) [76] allow clients to invoke operations on distributed objects without concern for:

Object location: CORBA objects can be collocated with the client or distributed on a remote server, without affecting their implementation or use.

Programming language: The languages supported by CORBA include C, C++, Java, Ada95, COBOL, and Smalltalk, among others.

OS platform: CORBA runs on many OS platforms, including Win32, UNIX, MVS, and real-time embedded systems like VxWorks, Chorus, and LynxOS.

Communication protocols and interconnects: The communication protocols and interconnects that CORBA can run on include TCP/IP, IPX/SPX, FDDI, ATM, Ethernet, Fast Ethernet, embedded system backplanes, and shared memory.

Hardware: CORBA shields applications from differences in hardware such as RISC vs. CISC instruction sets.

Figure 1.5 illustrates the components in the CORBA reference model, all of which collaborate to provide the portability, interoperability, and transparency outlined above.

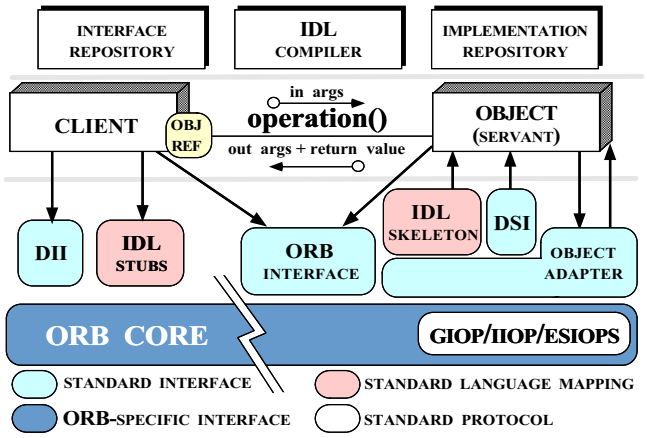


Figure 1.5: Components in the CORBA Reference Model

Each component in the CORBA reference model is outlined below:

Servant: This component implements the operations defined by an OMG Interface Definition Language (IDL) interface. In languages like C++ and Java that support object-oriented (OO) programming, servants are implemented using one or more objects. In non-OO languages like C, servants are typically implemented using functions and `structs`. A servant is identified by its *object reference*, which uniquely identifies the servant in a server process.

Client: This program entity performs application tasks by obtaining object references to servants and invoking operations on the servants. Servants can be remote or collocated relative to the client. Ideally, accessing a remote servant should be as simple as calling an operation on a local object, *i.e.*, `object→operation(args)`. Figure 1.5 shows the

components that ORBs use to transmit requests transparently from client to servant for remote operation invocations.

ORB Core: When a client invokes an operation on a servant, the ORB Core is responsible for delivering the request to the servant and returning a response, if any, to the client. For servants executing remotely, a CORBA-compliant [51] ORB Core communicates via the Internet Inter-ORB Protocol (IIOP), which is a version of the General Inter-ORB Protocol (GIOP) that runs atop the TCP transport protocol. An ORB Core is typically implemented as a run-time library linked into client and server applications.

ORB Interface: An ORB is a logical entity that may be implemented in various ways, *e.g.*, one or more processes or a set of libraries. To decouple applications from implementation details, the CORBA specification defines an abstract interface for an ORB. This ORB interface provides standard operations that (1) initialize and shutdown the ORB, (2) convert object references to strings and back, and (3) creates argument lists for requests made through the Dynamic Invocation Interface (DII) described below.

OMG IDL Stubs and Skeletons: IDL stubs and skeletons serve as a "glue" between the client and servants, respectively, and the ORB. Stubs provide a strongly-typed, *static* invocation interface (SII) that marshals application data into a common packet-level representation. Conversely, skeletons demarshal the packet-level representation back into typed data that is meaningful to an application.

IDL Compiler: An IDL compiler automatically transforms OMG IDL definitions into an application programming language like C++ or Java. In addition to providing language transparency, IDL compilers eliminate common sources of network programming errors and provide opportunities for automated compiler optimizations [13].

Dynamic Invocation Interface (DII): The DII allows clients to generate requests at run-time. This flexibility is useful when an application has no compile-time knowledge of the interface it is accessing. The DII also allows clients to make *deferred synchronous* calls, which decouple the request and response portions of two-way operations to avoid blocking the client until the servant responds. In contrast, SII stubs support only *two-way*, *i.e.*, request/response, and *oneway*, *i.e.*, request only operations.[1]

Dynamic Skeleton Interface (DSI): The DSI is the server's analogue to the client's DII. The DSI allows an ORB to deliver requests to a servant that has no compile-time knowledge of the IDL interface it is implementing. Clients making requests need not know whether the server ORB uses static skeletons or dynamic skeletons. Likewise, servers need not know if clients use the DII or SII to invoke requests.

[1]The OMG has recently standardized an asynchronous method invocation interface, as well.

Object Adapter: An Object Adapter associates a servant with an ORB, demultiplexes incoming requests to the servant, and dispatches the appropriate operation upcall on that servant. Recent CORBA portability enhancements [51] define the Portable Object Adapter (POA), which supports multiple nested POAs per ORB. Object Adapters make it possible for an ORB to support various types of servants that possess similar requirements. This architecture results in a small and simple ORB that can still support a wide range of object granularities, lifetimes, policies, implementation styles, and other properties.

Interface Repository: The Interface Repository provides run-time information about IDL interfaces. Using this information, it is possible for a program to encounter an object whose interface was not known when the program was compiled, yet, be able to determine what operations are valid on the object and make invocations on it. In addition, the Interface Repository provides a common location to store additional information associated with interfaces ORB objects, such as stub/skeleton type libraries.

Implementation Repository: The Implementation Repository contains information that allows the ORB to locate and activate servants. Most of the information in the Implementation Repository is specific to an ORB or operating environment. In addition, the Implementation Repository provides a common location to store information associated with servants, such as administrative control, resource allocation, and security.

The use of CORBA as communication middleware enhances application flexibility and portability by automating common network programming tasks such as object location, object activation, and parameter marshaling. CORBA is an improvement over conventional procedural RPC middleware like OSF DCE since it supports object-oriented language features and more flexible communication mechanisms beyond standard request/response RPC.

Object-oriented language features supported by CORBA include encapsulation, interface inheritance, parameterized types, and exception handling. The flexible communication mechanisms supported by CORBA include its dynamic invocation capabilities and object reference parameters that support distributed callbacks and peer-to-peer communication. These features enable complex distributed and concurrent applications to be developed more rapidly and correctly.

The principal drawback to using middleware like CORBA is its potential for lower throughput, higher latency, and lack of scalability over high-speed networks. Conventional CORBA ORBs have not been optimized significantly. In general, ORB performance has not generally been an issue on low-speed networks, where middleware overhead is often masked by slow link speeds. On high-speed networks, however, this overhead becomes a significant factor that limits communication performance and ultimately limits adoption of CORBA by developers.

1.4 Overview of TAO

We believe that developing real-time OO middleware requires a systematic, measurement-driven methodology to identify and alleviate sources of ORB endsystem overhead, priority inversion, and non-determinism. The ORB software architectures presented in this thesis are based on our experience developing, profiling, and optimizing next-generation avionics [31] and telecommunications [65] systems using OO middleware such as ACE (Adaptive Communications Framework) [64] and TAO (The ACE ORB) [69].

ACE is an OO framework that implements core concurrency and distribution patterns [18] for communication software. It provides reusable C++ wrapper facades and framework components that support high-performance, real-time applications. ACE runs on a wide range of OS platforms, including Win32, most versions of UNIX, and real-time operating systems like VxWorks, Sun ClassiX, pSoS, and LynxOS.

TAO is a highly extensible ORB endsystem written using ACE. It is targeted for applications with deterministic and statistical QoS requirements, as well as best effort requirements. TAO is fully compliant with the latest OMG CORBA specifications [51] and is the first standard CORBA ORB endsystem that can support end-to-end QoS guarantees over ATM networks.

The TAO project focuses on the following topics related to real-time CORBA and ORB endsystems:

- Identifying enhancements to standard ORB specifications, particularly OMG CORBA, that will enable applications to specify their QoS requirements concisely and precisely to ORB endsystems [22].
- Empirically determining the features required to build real-time ORB endsystems that can enforce deterministic and statistical end-to-end application QoS guarantees [69].
- Integrating the strategies for I/O subsystem architectures and optimizations [41] with ORB middleware to provide end-to-end bandwidth, latency, and reliability guarantees to distributed applications.
- Capturing and documenting the key design patterns [66] necessary to develop, maintain, configure, and extend real-time ORB endsystems.

This thesis contributes towards bullets 2 and 3 shown above.

1.5 CORBA Testbed Environment

In this section, we describe details of the testbed environment we used to perform all the experiments as well as the profiling tools we used. The information provided here is

common to all the tests we performed. Individual chapters provide additional details of the experiment performed.

1.5.1 Hardware and Software Platforms

Our experimental ATM/CORBA testbed is depicted in Figure 1.6.

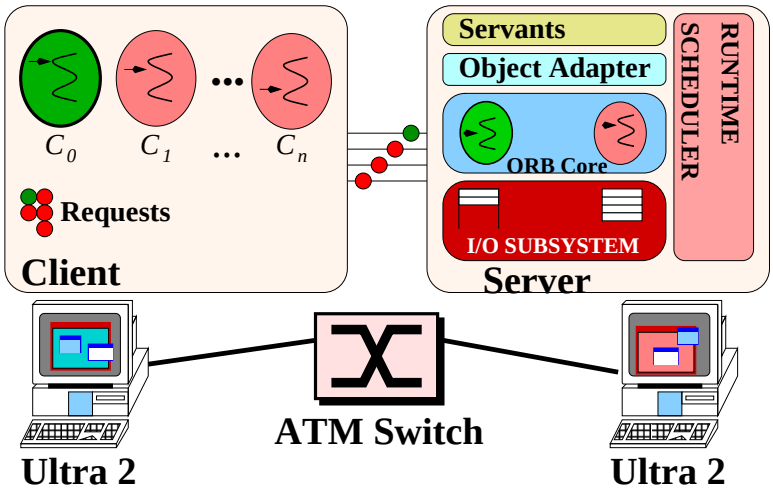


Figure 1.6: ATM Testbed for ORB Endsystem Performance Experiments

The experiments for this research were conducted using Bay Networks LattisCell 10114 and FORE systems ASX-1000 ATM switches connected to two dual-processor SPARCstation 20 Model 712s running SunOS 5.4 and UltraSPARC-2s running SunOS 5.5.1, respectively. The former configuration was used for our preliminary work on identifying the sources of overhead in CORBA as described in Chapters 2, 3, and 4. The latter configuration was used to test the results of our optimizations described in Chapters 5, 6, and 7.

The LattisCell 10114 is a 16 Port, OC-3 155 Mbs/port switch. Each SPARCstation 20 contains two 70 MHz Super SPARC CPUs with a 1 Megabyte cache per-CPU. The SunOS 5.4 TCP/IP protocol stack is implemented using the `STREAMS` communication framework [62]. Each SPARCstation has 128 Mbytes of RAM and an ENI-155s-MF ATM adaptor card, which supports 155 Megabits per-sec (Mbps) SONET multimode fiber. The Maximum Transmission Unit (MTU) on the ENI ATM adaptor is 9,180 bytes. Each ENI card has 512 Kbytes of on-board memory. A maximum of 32 Kbytes is allotted per ATM virtual circuit connection for receiving and transmitting frames (for a total of 64 K). This allows up to eight switched virtual connections per card.

The ASX-1000 is a 96 Port, OC-12 622 Mbps/port switch. Each UltraSparc-2 contains two 168 MHz Super SPARC CPUs with a 1 Megabyte cache per-CPU. The SunOS 5.5.1

TCP/IP protocol stack is also implemented using the STREAMS communication framework. Each UltraSPARC-2 has 256 Mbytes of RAM and an ENI-155s-MF ATM adaptor card, which supports 155 Megabits per-sec (Mbps) SONET multimode fiber. The MTU and on-board memory on the ENI card is the same as before.

1.5.2 Profiling Tools

Detailed timing measurements used to compute latency were made with the `gethrtime` system call available on SunOS 5.5. This system call uses the SunOS 5.5 high-resolution timer, which expresses time in nanoseconds from an arbitrary time in the past. The time returned by `gethrtime` is very accurate since it does not drift.

The profile information for the empirical analysis was obtained using the `Quantify` [35] performance measurement tool. `Quantify` analyzes performance bottlenecks and identifies sections of code that dominate execution time. Unlike traditional sampling-based profilers (such as the UNIX `gprof` tool), `Quantify` reports results without including its own overhead. In addition, `Quantify` measures the overhead of system calls and third-party libraries without requiring access to source code. All data is recorded in terms of machine instruction cycles and converted to elapsed times according to the clock rate of the machine. The collected data reflect the cost of the original program's instructions and automatically exclude any `Quantify` counting overhead.

Additional information on the run-time behavior of the code such as system calls made, their return values, signals, number of bytes written to the network interface, and number of bytes read from the network interface are obtained using the UNIX `truss` utility, which traces the system calls made by an application. `truss` was used to observe the return values of system calls such as `write` and `read`, which indicates the number of times that buffers were written and read from the network.

1.6 Organization

The dissertation is organized as follows: Chapters 2 and 3 evaluate the throughput performance of conventional CORBA implementations using the static (SII) and dynamic (DII) invocation interfaces, respectively. Chapter 4 evaluates the performance of CORBA ORBs in terms of their end-to-end latency and scalability; Chapter 5 describes the optimizations we developed to significantly reduce the IIOP CDR marshaling engine overhead; Chapter 6 describes optimizations we developed to eliminate the unpredictability and inconsistency in the performance of request demultiplexing and dispatching; Chapter 7 describes the design and implementation of our CORBA IDL compiler that generates efficient stubs and skeletons; Chapter 8 describes related work; and Chapter 9 presents concluding remarks and future work.

Chapter 2

Throughput Performance of CORBA's Static Invocation Interface

2.1 Introduction

Applications using CORBA's Static Invocation Interface (SII) invoke operations on objects passing it parameters. These parameters are then marshaled and bundled into the outgoing bytestream by the stubs generated by the IDL compiler. Similarly, on the receiving end, the IDL compiler-generated skeletons unmarshal the parameters before making the upcall. Just like a Remote Procedure Call (RPC), applications are unaware of the stubs and skeletons.

This chapter quantifies the performance of two widely used existing CORBA implementations (Orbix 2.0 and VisiBroker 2.0) in terms of their support for the *static invocation interface* (SII). We measure the end to end throughput observed by bandwidth-intensive CORBA applications that transfer typed as well as untyped data. [1] Our aim in performing these experiments is to identify key sources of overheads in the IDL compiler-generated stubs and skeletons that marshal and unmarshal parameters.

The chapter is organized as follows: Section 2.2 describes our experimental testbed environment; Section 2.3 demonstrates the key sources of overhead in conventional CORBA implementations over ATM; and Section 2.4 presents a summary and research contributions.

[1] These results have appeared in the ACM SIGCOMM 96 conference proceedings.

2.2 Additional Features of the CORBA/ATM Testbed Environment

This section describes features of our testbed environment that are specific for the experiments described in this chapter. Features common to all the tests are described in Section 1.5.

2.2.1 Hardware and Software Platforms

To approximate the performance of communication middleware for channel speeds greater than our available ATM network, we also duplicated our experiments in a loopback mode using the I/O backplane of a dual-CPU SPARCstation 20s as a high-speed "network." The user-level memory-to-memory bandwidth of our SPARCstation 20 model 712s was measured at 1.4 Gbps, which is roughly comparable to an OC-24 gigabit ATM network [55].

2.2.2 Traffic Generators

Earlier studies [68, 57] tested the performance of "flooding models" that transferred untyped bytestream data between hosts using several implementations of CORBA and other lower-level mechanisms like sockets. Untyped bytestream traffic is representative of applications like bulk file transfer and videoconferencing. Note, however, that bytestream traffic does not adequately test the overhead of presentation layer conversions since untyped data need not be marshalled or demarshalled. Ironically, the implementations of CORBA used in our tests perform marshaling and demarshaling even for untyped `octet` data [68], which is further evidence that CORBA implementations have not been optimized for high-speed networks.

The experiments described in this chapter extend our earlier studies [68] by measuring the performance of sockets, ACE C++ wrappers for sockets [64], standard- and hand-optimized version of Sun's Transport Independent RPC (TI-RPC) [72], and two widely used implementations of CORBA (Orbix 2.0 and VisiBroker 2.0) to transfer both bytestream and typed data between remote hosts over a high-speed ATM network. The use of typed data is representative of applications like electronic medical imaging [11, 3] and high-speed distributed databases (such as global change repositories [56]). In addition, measuring typed data transfer reveals the overhead of presentation layer conversions and data copying for the various communication middleware mechanisms we measured.

Traffic for the experiments was generated and consumed by an extended version of the widely available TTCP [74] protocol benchmarking tool. We extended TTCP for use with C sockets, C++ socket wrappers, TI-RPC, Orbix, and VisiBroker. Our TTCP tool measures end-to-end data transfer throughput in Mbps from a transmitter process to a remote receiver process across an ATM network or host loopback. The flow of user data

for each version of TTCP is uni-directional, with the transmitter flooding the receiver with a user-specified number of data buffers. Various sender and receiver parameters may be selected at run-time. These parameters include the size of the socket transmit and receive queues, the number of data buffers transmitted, the size of data buffers, and the type of data in the buffers.

The following data types were used for all the tests: scalars (`short`, `char`, `long`, `octet`, `double`) and a C++ `struct`, called `BinStruct`, composed of all the scalars. The CORBA implementation transferred the data types using IDL `sequences`, which are dynamically sized arrays. To compare CORBA with C and C++, we defined `structs` in the same manner that the CORBA IDL compiler generated `sequences`. Likewise, to compare CORBA with TI-RPC, we generated `structs` using unbounded arrays defined in the RPC language (RPCL). These definitions are shown in Appendix A.

The C and C++ versions of TTCP were written using the standard Internet family of macros that convert values between host and network byte order. These macros are implemented as "no-ops" because the sender and receiver processes both ran on SPARCs, which use big-endian network byte order. Therefore, the C/C++ versions do not actually perform any presentation layer conversions on the data. The CORBA and the RPC versions of TTCP also omit these conversions since they use the byte order macros, as well. However, the CORBA and RPC implementations do *not* omit the overhead of the no-op function calls, which has a non-trivial overhead (shown in Section 2.3.3).

2.2.3 TTCP Parameter Settings

Existing studies [11, 45, 9, 68, 57] of transport protocol performance over ATM demonstrate the impact of parameters such as socket queue sizes and data buffer on performance. Therefore, our TTCP benchmarks varied these two parameters for each type of data as follows:

• **Socket queue size:** The sender and receiver socket queue sizes used were 8 K and 64 K bytes (on SunOS 5.4 these are the default and maximum, respectively). These parameters influence the size of the TCP segment window, which has been shown to significantly impact CORBA-level and TCP-level performance on high-speed networks [45, 68]. Since the performance of the 8 K socket queues was consistently one-half to two-thirds slower than using the 64 K queues, we omitted the 8 K results from the figures below.

• **Data buffer size:** Sender buffers were incremented by powers of two, ranging from 1 Kbytes to 128 Kbytes. The experiment was carried out ten times for each buffer size to account for variations in ATM network traffic (which was insignificant since the network was otherwise unused). The average of the throughput results is reported in the figures below.

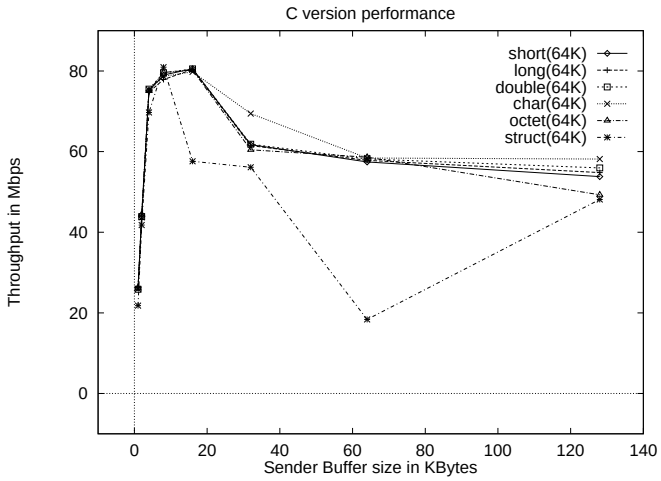


Figure 2.1: Performance of the C Version of TTCP

In the tests involving 64 K receiver socket queue, the sender socket queue was also set to 64 K and the `TCP_NODELAY` flag was set. The TCP_NODELAY flag was used to send small packets (resulting due to fragmentation) as soon as possible. The receiver in the C and C++ versions set a user-level `read` buffer of size 64 K to read incoming data.

2.3 Throughput Results

The user-level, end-to-end throughput measurements for each of the six versions of TTCP are presented first. Detailed profiling measurements of presentation layer, data copying, and memory management overhead are presented in Section 2.3.3. The profile data was obtained using the `Quantify` performance measurement tool. `Quantify` analyzes performance bottlenecks and identifies sections of code that dominate execution time. An important feature of `Quantify` is its ability to report results without including its own overhead, unlike traditional sampling-based profilers like the UNIX `gprof` tool.

2.3.1 Remote Transfer Results:

Figures 2.1, 2.2, 2.5, 2.6, 2.7 and 2.8 depict the throughput obtained for sending 64 MB data of various data types for each TTCP implementation over ATM. These figures present the observed user-level throughput at the sender for buffer sizes of 1 K, 2 K, 4 K, 8 K, 16 K, 32 K, 64 K and 128 K bytes using 64 KB sender and receiver socket queues (the maximum possible on SunOS 5.4).[2] This section analyzes the overall trends of the throughput for each communication middleware mechanism. Section 2.3.3 uses profiling output from `Quantify` to explain why performance differences occur.

[2]Our tests revealed that the receiver-side throughput was approximately the same as the sender-side. Therefore, we only show sender-side throughput results.

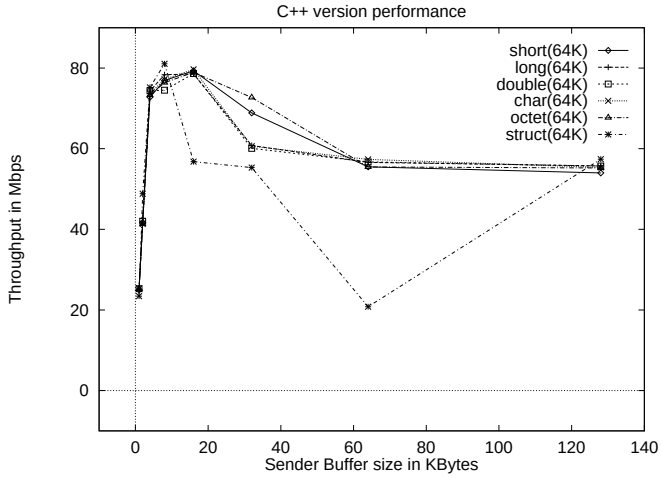


Figure 2.2: Performance of the C++ Wrappers Version of TTCP

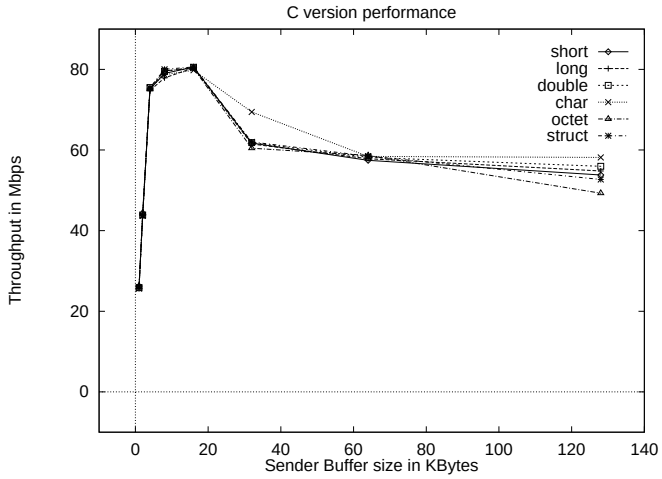


Figure 2.3: Performance of the Modified C Version of TTCP

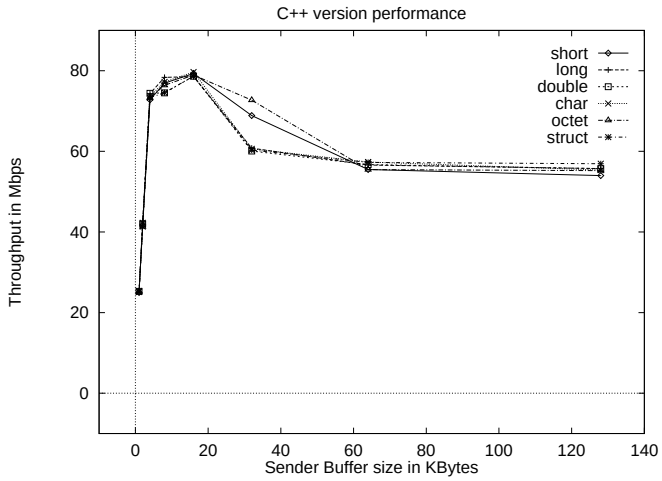


Figure 2.4: Performance of the Modified C++ Version of TTCP

• **C and C++ versions of TTCP:** Figures 2.1 and 2.2 indicate that the C and C++ versions both achieved a maximum of 80 Mbps throughput for sender buffer sizes of 8 K and 16 K bytes. The similarity between these indicates that the performance penalty for using the higher-level C++ wrappers is insignificant, compared with using C socket library function calls directly.

As shown in the figures, the throughput increases steadily from 1 K to 8 K buffer sizes. The reason for this is that as the sender buffer size increases, the sender requires fewer `writes` to transmit 64 MB of data. The throughput peaks between the 8 K and 16 K buffer sizes and then gradually decreases – leveling off at around 60 Mbps. This drop off between 8 K and 16 K arises from the 9,180 MTU of the ATM network. When sender buffer sizes exceed this amount, fragmentation at the IP and ATM driver layers degrades performance. As sender buffer sizes increase, fragmentation becomes a dominant factor, yielding the performance curves shown in Figures 2.1 and 2.2.

Close scrutiny of Figures 2.1 and 2.2 illustrate unusual behavior for `BinStructs` when the sender buffers are 16 K and 64 K. In these cases throughput drops sharply. Analysis of `Quantify's` profile information for 64 K sender buffers revealed that the `writev` system call is called 1,025 times, accounting for 28,031 msec of the total execution time. In contrast, in the best case (sending `longs`) the 1,025 calls to `writev` accounted for only 9,087 msec of the total execution time.

This aberrant behavior occurs since 64 K is not an integral multiple of the size of the C and C++ `BinStruct` data type (which is 24 bytes). Therefore, the sender buffers were slightly less than 64 K when written with the `writev` function. This minor difference apparently triggered interactions between the SunOS 5.4 internal STREAMS buffering strategy and the TCP sliding window protocol, which yielded extremely low throughput. To work around this problem, we defined a C/C++ `union` that ensures the size of the transmitted data is rounded up to the next power of 2 (in this case 32 bytes). This enabled TTCP to send 64 K bytes in a single `writev` call and obtain throughput comparable to the other data types. These new results are shown in Figures 2.3 and 2.4.

• **RPC version of TTCP:** Figures 2.5 and 2.6 show the performance of the original and hand-optimized RPC versions of TTCP. The original stubs generated automatically by RPCGEN attained extremely low throughout (peaking at 29 Mbps for `doubles`, which is only 35% of the throughput attained by the C and C++ versions). `Quantify` analysis reveals that this poor performance was due to excessive data copying and presentation layer conversions performed by XDR (explained in Section 2.3.3).

To make the implementation comparable to the C/C++ TTCP implementations, we hand-optimized the RPC generated code for TTCP. For all the data types, the `xdr_bytes` function generated by RPCGEN was used to send/receive data. This avoided the overhead of converting between the native and XDR formats. This optimization was valid because

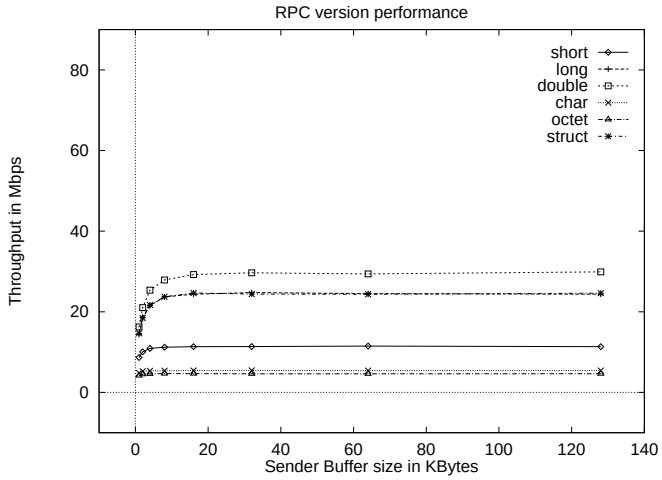


Figure 2.5: Performance of the Standard RPC Version of TTCP

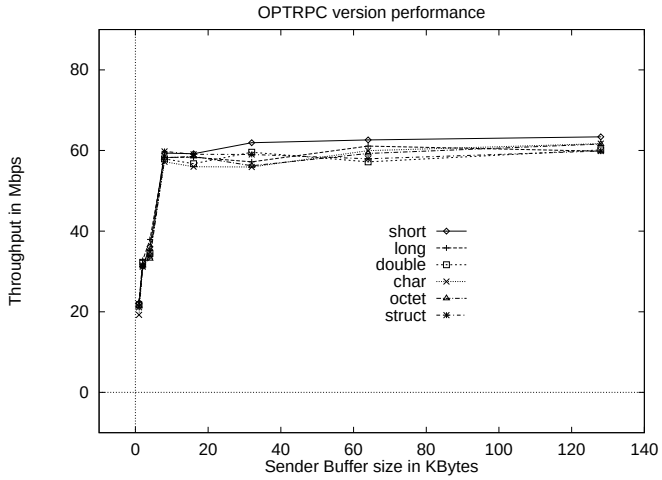


Figure 2.6: Performance of the Optimized RPC Version of TTCP

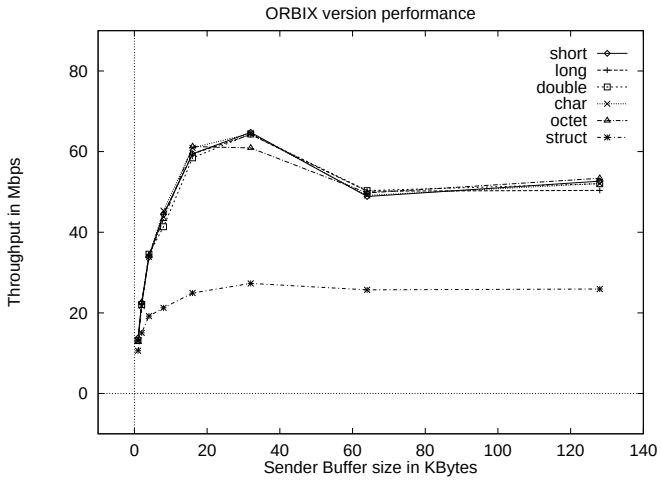


Figure 2.7: Performance of the Orbix Version of TTCP

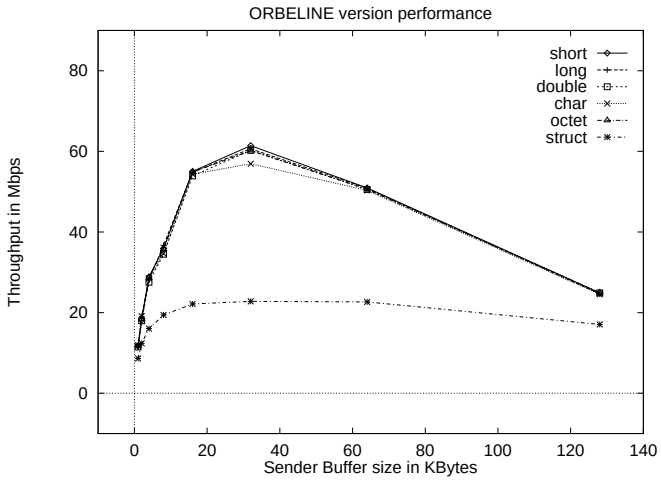


Figure 2.8: Performance of the VisiBroker Version of TTCP

the data was transferred between big-endian SPARCstations with the same alignment and word length.

The hand-optimized code improved the performance significantly. Figure 2.6 illustrates the results are 79% of the C and C++ performance. These results indicate that for sender buffer sizes from 8 K to 128 K, the measured throughput was roughly 59-63 Mbps. The throughput steadily increases until the sender buffer size reaches 8 K. Beyond this point there was only a marginal improvement in the throughput.

The shape of the optimized RPC performance curves result from the 9,000 bytes data buffer sizes sent by the generated RPC stubs. Analysis from `Quantify` and the SunOS5.4 system-call tracing command (`truss`) reveals that the RPC sender-side stubs use 9,000 byte internal buffers to make the `writes`. As a result, the performance attained for sender buffer sizes from 8 K to 128 K show only a marginal improvement, which is attributed to the use of 64 K socket queue sizes at the sender and receiver.

• **CORBA versions of TTCP:** Figures 2.7 and 2.8 illustrate the throughput obtained for both CORBA implementations. These figures indicate that the throughput steadily increases until the sender buffers reach 32 K, at which point it peaks at 65 Mbps for Orbix and 60 Mbps for VisiBroker for sending scalars. Beyond this point, performance gradually decreases. This behavior differs from the C and C++ versions, which peak at 8 K and 16 K. VisiBroker performance falls off much more quickly than Orbix performance. This effect is noticeable for sender buffer size of 128 K in Figure 2.8.

Analysis using `truss` for 128 K sender buffer size revealed that both the Orbix and the VisiBroker versions try to write the entire 128 K bytes plus some control information (56 bytes for Orbix and 64 bytes for VisiBroker). The Orbix version uses the `write` system call, whereas the VisiBroker version uses the `writev` system call.

Analysis using `Quantify` indicated that to send 64 MB user data using 128 K user buffer size, the Orbix version attempted a total of 538 `write`s, which required 9,638 msec. In contrast, the VisiBroker version made a total of 512 `writev`s, which required 20,319 msec to complete. This explains the lower throughput for the VisiBroker client. The receiver performance in both cases is comparable. The `truss` output for the Orbix and the VisiBroker receiver shows that the time spent by VisiBroker in `read`s is marginally smaller than that of the Orbix version, but this is offset by the 4,252 `poll` system calls made by VisiBroker compared to only 539 made by the Orbix receiver. For the 32 K data buffers, the performance of Orbix and VisiBroker is comparable with the 65-70 Mbps attained by the C/C++ versions. Likewise, the optimized RPC version achieved roughly the same throughput as the CORBA implementations. However, both CORBA implementations achieved approximately half the throughput for `structs`. As shown in Section 2.3.3, this performance reduction occurs from the high amount of presentation layer conversions and data copying in Orbix and VisiBroker. In addition, `truss` revealed that both the CORBA implementations write buffers containing only 8 K when sending `structs`. In contrast, for 32 K data buffers, they sent scalars in buffers containing all 32 K data plus additional control information, as described above. This behavior adds to the overhead imposed by data copying and presentation layer conversions and greatly reduces throughput.

2.3.2 Loopback Results:

Figures 2.9, 2.10, 2.11, 2.12, 2.13 and 2.14 depict the throughput obtained by replicating the TTCP tests described above through the SPARCstation loopback device. Measuring loopback behavior approximates the performance of communication middleware for channel speeds greater than our 155 Mbps ATM network.

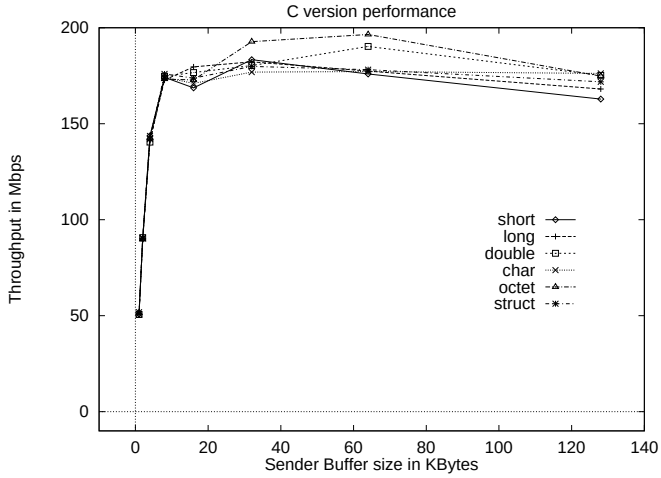


Figure 2.9: Performance of the C Loopback Version of TTCP

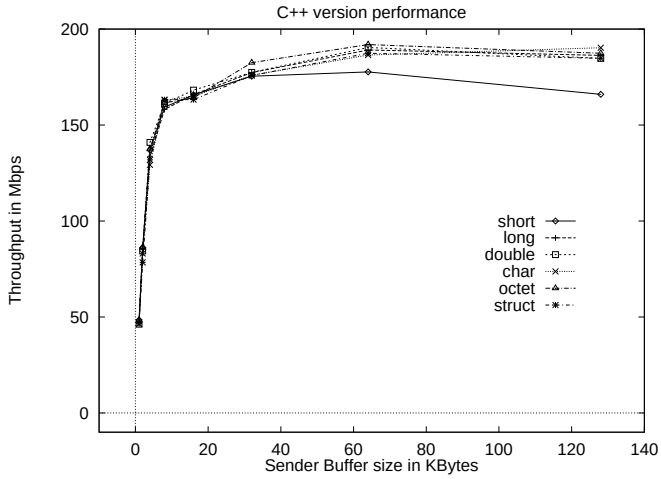


Figure 2.10: Performance of the C++ Wrappers Loopback Version of TTCP

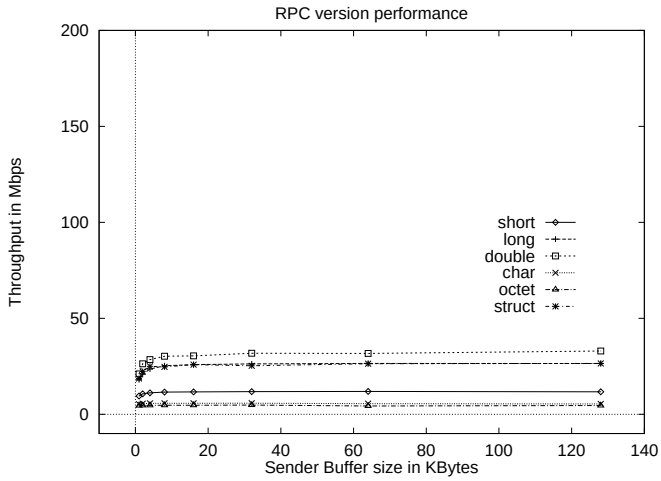


Figure 2.11: Performance of the Standard RPC Loopback Version of TTCP

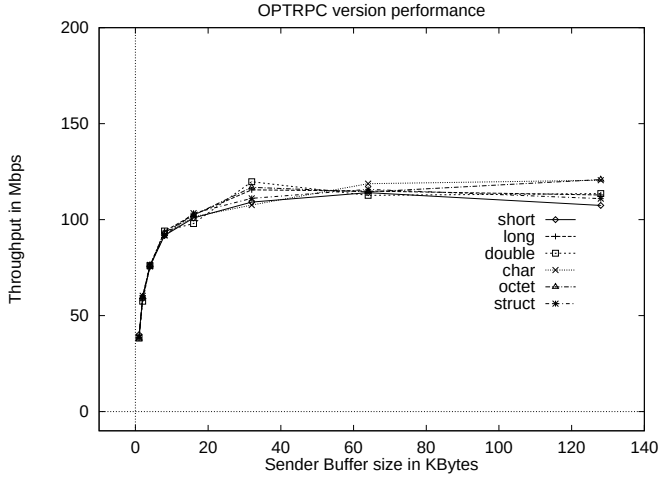


Figure 2.12: Performance of the Optimized RPC Loopback Version of TTCP

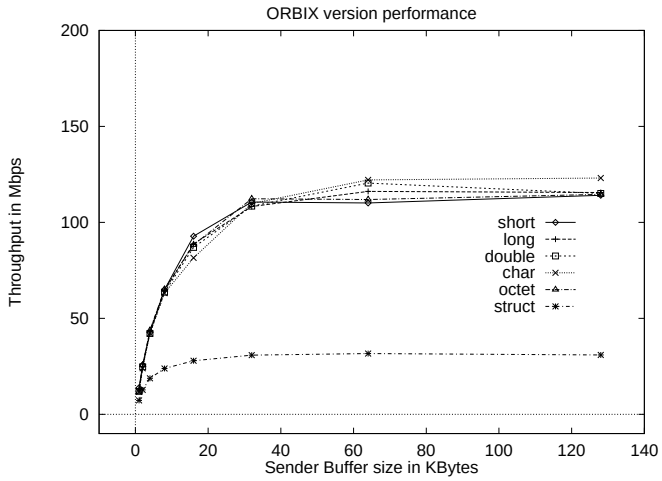


Figure 2.13: Performance of the Orbix Loopback Version of TTCP

• **C/C++ Results:** The results indicate that for C/C++ versions of TTCP, the throughput starts leveling off around 8 K sender buffer size at roughly 190 Mbps. Due to the implementation of the loopback device in SunOS 5.4, throughput was not affected as significantly by fragmentation overhead compared with the ATM results shown in Figures 2.1 and 2.2.

• **RPC Results:** The original RPC version did not show any significant change over the remote transfer results. The optimized RPC version of TTCP exhibited behavior similar to C/C++ over the loopback, leveling off at around 110 Mbps. This behavior is attributed to the smaller internal buffer size [3] RPC uses to `write` data on the sender-side and `read` on the receiver-side. This smaller size increases the number of times these functions are invoked.

[3] As explained earlier, the RPC version used an internal buffer of roughly 9,000 bytes for writing and reading. In contrast, the C/C++ versions used a 64 KB read buffer and sends are done according to the size of buffers passed by the client.

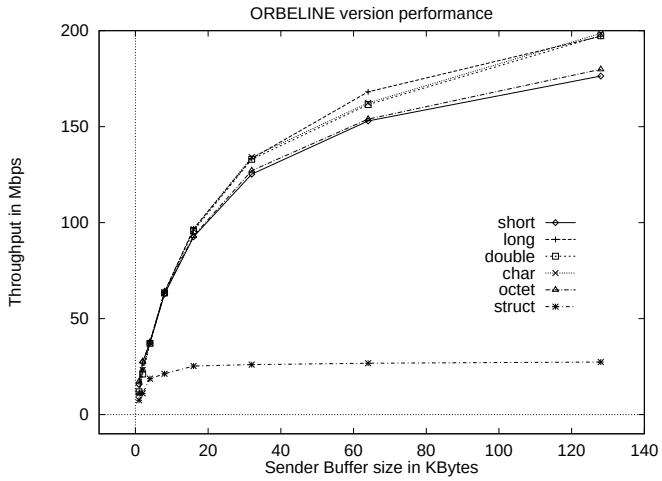


Figure 2.14: Performance of the VisiBroker Loopback Version of TTCP

• **CORBA Results:** The Orbix version of TTCP behaves like the optimized RPC for all scalar data types. The VisiBroker version shows a gradual increase in throughput, which peaks at around 197 Mbps for a sender buffer size of 128 K which is close to the C/C++ version performance for the loopback case. Analysis using `Quantify` for 128 K sender buffers reveals that both Orbix and VisiBroker senders and receivers spend approximately equal amount of time in `write`s and `read`s. However, the Orbix version spends around 896 msec in `memcpy` on both the sender and receiver side compared to only 1.51 msec for VisiBroker sender and 15.51 msec for VisiBroker receiver.

This explains why the VisiBroker throughput for loopback is higher than the Orbix throughput with increasing buffer size. However, both the Orbix and VisiBroker still perform poorly for `structs` because they spend a significant amount of time performing presentation layer conversions and data copying. Although VisiBroker provides an option of using shared memory, we did not use this option since our goal was to measure throughput over the "network" and not the memory speed.

In general, the highest throughput for Orbix is approximately 75-80% that of the C/C++ versions for remote transfers and around 68% for the loopback test. The difference in the throughputs is most apparent for `structs`. In this case, the throughput for both the Orbix and VisiBroker versions is roughly 33% of the C/C++ version for remote transfers and 16% for the loopback tests.

These findings illustrate that as channel speeds increase, the performance of the CORBA implementations become worse relative to that achieved by low-level communication middleware since the overhead of presentation layer conversions and data copying become increasingly dominant. Thus, it is imperative to eliminate this overhead so that CORBA can be used effectively to build flexible and reliable middleware capable of delivering very high data rates to applications.

Table 2.1: Summary of Observed Throughput for Remote and Loopback Tests in Mbps

TTCP version	**Remote Transfer**				**Loopback**			
	Scalars		**Struct**		**Scalars**		**Struct**	
	Hi	**Lo**	**Hi**	**Lo**	**Hi**	**Lo**	**Hi**	**Lo**
C/C++	80	25	80	25	197	47	190	47
Orbix	65	15	27	11	123	14	32	10
VisiBroker	61	12	23	9	197	11	27	7
RPC	30	4	25	14	33	5	27	18
optRPC	63	20	63	20	121	38	116	38

The results for the remote and loopback tests for all versions of TTCP are summarized in Table 2.1. This table depicts the highest and the lowest throughput attained by each version of TTCP for all the scalars and `structs`. In addition, we combine results for the C and C++ versions of TTCP since their performance is similar. All entries are in Mbps rounded up to the nearest integer.

2.3.3 Presentation Layer and Data Copying Overhead

Section 2.3 presented the "blackbox" performance results. This section presents the "white-box" performance results. Tables 2.2 and 2.3 depict the time spent by the senders and receivers of various versions of TTCP when transferring 64 Mbytes of sequences using 128 K sender and receiver buffers and 64 K socket queues. For each version, an analysis for a specific data type is presented if it resulted in throughput that differs from that of the rest of the data types. Otherwise, an analysis for a representative data type is presented (*e.g.,* `BinStruct`).

For Orbix and VisiBroker, `structs` resulted in throughput that differed significantly from the throughput for the rest of the data types. Therefore, we present analysis for `struct` and `char`, which are representative data types. In the tables, the % column shows the percentage of the total execution time attributed to the corresponding function under the **Method Name** column. The time spent in milliseconds by this method is indicated in the **msec** column. This fine-grained profiling information reveals precisely why the C and C++ implementations outperform the RPC and CORBA implementations.

Sender-side Overhead:

The overhead for the sender-side presentation layer and data copying is presented below for each version of the TTCP benchmarks.

• **C/C++ Overhead:** The C and C++ versions of TTCP spent over 98% of their runtime making `writev` system calls. In this case, there is no presentation layer conversion

Table 2.2: Sender-side Overhead

TTCP Version	Data Type	Analysis		
		Method Name	msec	%
C/C++	struct	writev	9,415	98
RPC	char	write	283,350	89
		xdr_char	17,000	5
	short	write	134,855	90
	long	write	71,600	92
	double	write	37,877	87
		xdr_double	2,348	5
	struct	write	80,517	92
optRPC	struct	write	4,262	80
		memcpy	896	17
Orbix	char	write	9,638	89
		memcpy	895	8
	struct	write	26,366	68
		NullCoder::codeLongArray	1,162	3
		BinStruct::encodeOp	952	2
		CHECK	932	2
		Request::encodeLongArray	812	2
		Request::insertOctet	782	2
		Request::op<<(double&)	838	2
		Request::op<<(short&)	782	2
		Request::op<<(long&)	782	2
		Request::op<<(char&)	782	2
VisiBroker	char	writev	20,319	99
	struct	writev	82,794	84
		op<<(NCostream&,BinStruct&)	3,831	4
		memcpy	3,594	4
		PMCIIOPStream::put	951	1
		PMCIIOPStream::op<<(double)	978	1
		PMCIIOPStream::op<<(long)	950	1

overhead. As explained earlier, the standard Internet family of macros that convert values between host and network byte order are implemented as "no-ops."

• **RPC Overhead:** The RPC version of TTCP spends different amounts of time writing various data types. For instance, to write `chars`, the RPC version takes 283,330 msec compared to 71,600 msec for writing `longs`. The reason for this behavior is due to the RPC XDR mapping, which converts a single byte `char` into a four byte data representation before it is sent over the network. The hand-optimized version of RPC considers all data types as `opaque`, which avoids the XDR mapping for each data type. The hand-optimized RPC version of TTCP is also largely `write` bound, though it spends about 17% of its time performing data copies with `memcpy`. The significant amount of `memcpys` is due to the XDR routine `xdrrec_putbytes` being called many times on the sender-side. The user buffer is copied into an internal buffer, which is then sent over the network.

• **CORBA Overhead:** The sender-side of the Orbix version of TTCP that transmitted `BinStructs` spent a significant amount of time marshalling the `BinStruct` fields. For 64 MB of data and a sender buffer of 128 KB, the client invokes the `sendBinStruct` method 512 times. This method invokes the IDL compiler generated `_IDL_SEQUENCE _BinStruct`

`::encodeOp` method. Since a `BinStruct` is 32 bytes, each sender buffer of size 128 KB can accommodate 4,096 `structs`. For each `struct`, the Orbix version marshalled each field using `CORBA::Request` methods like `::encodeLongArray` and `::operator<< (const long&)`. Each of these marshalling routines was invoked for the 4,096 structs in a single buffer for 512 iterations, yielding a total of 2,097,152 invocations! Moreover, each of these calls are C++ virtual function, which incur still more levels of indirection.

Analysis using `Quantify` revealed that for `BinStructs`, the Orbix sender spent around 68% of its time (26,366 msec) in `writes`, 1.71% (671 msec) doing `memcpys` and over 18% time marshalling the structure. Likewise, the VisiBroker sender spent around 84% of the time (82,724 msec) in `writevs`, with approximately 4% in `memcpy` and 10% marshalling the structure.

Receiver-side Overhead:

Our benchmarks found the receiver-side tests performed similar to the sender-side tests.

• **C/C++ Overhead:** The C and C++ versions spent the bulk of their time in `read` and `readv`. The C and C++ versions on the receiver side used `readv` to read the `length`, `type` and `buffer` fields of the structures, thereby avoiding an intermediate copy. If the buffer is not completely received by `readv`, subsequent `reads` fill in the rest of the buffer.

• **RPC Overhead:** The receiver side analysis for the RPC version of TTCP shows that the RPC code spent a significant amount of time demarshalling various data types from the XDR network representation to the native host format. For instance, to demarshal the `chars` using `xdr_char` takes 30,422 msec. In contrast, to demarshall `longs` takes only 4,697 msec. As explained earlier, the hand-optimized RPC version eliminates this marshalling overhead by treating the data types as `opaque`.

The hand-optimized RPC version spent a significant amount of time doing `getmsg`, which stems from the use of the System V `STREAMS` in Sun's TI-RPC. Similar to the sender-side results in Table 2.2, the receiver-side RPC implementation spends about one-third of its time performing data copying. The time spent in `memcpy` is due to a large number of calls to an internal function called `get_input_bytes`, which in turn is invoked by `xdrrec_getbytes`. The buffer received through calls to `t_rcv` is copied into another buffer, which is subsequently passed to the user application. The contribution of these functions to the total execution time is insignificant, so their results are omitted from the table.

• **CORBA Overhead:** The results for Orbix indicate that a considerable amount of time was spent demarshaling each field of the structs that were received. This task was performed by a number of overloaded `operator>>` methods of the `CORBA::Request` class *e.g.,* `CORBA::Request::operator>>(double &)` to demarshall `double` types. The `Quantify` analysis of VisiBroker's server-side to receive `structs` reveals that around 19% of the time

Table 2.3: Receiver-side Overhead

TTCP Version	Data Type	Analysis		
		Method Name	msec	%
C/C++	struct	read	7,199	75
		readv	2,374	24
RPC	char	xdr_char	30,422	44
		xdrrec_getlong	16,998	24
		xdr_array	14,317	20
		getmsg	5,977	8
	short	xdr_short	11,184	36
		xdrrec_getlong	8,499	27
		xdr_array	7,158	23
		getmsg	2,969	9
	long	xdr_long	4,697	31
		xdrrec_getlong	4,250	28
		xdr_array	3,579	23
		getmsg	1,639	10
	double	xdr_double	3,467	29
		xdrrec_getlong	4,250	35
		xdr_array	1,790	15
		getmsg	1,562	13
	struct	xdrrec_getlong	4,250	26
		xdr_BinStruct	2,684	16
		getmsg	1,518	9
		xdr_char	1,267	7
		xdr_uchar	1,267	7
		xdr_double	1,155	7
optRPC	struct	getmsg	2,229	67
		memcpy	897	27
Orbix	char	read	7,915	85
		memcpy	896	9
	struct	read	4,280	26
		NullCoder::codeLongArray	1,314	8
		CHECK	923	5
		BinStruct::decodeOp	923	5
		Request::extractOctet	699	4
		Request::op>>(double&)	699	4
		Request::op>>(short&)	699	4
		Request::op>>(long&)	699	4
		Request::op>>(char&)	699	4
		memcpy	672	4
VisiBroker	char	read	3,041	85
	struct	memcpy	3,581	19
		read	3,533	18
		op>>(NCistream&,BinStruct&)	3,495	18
		PMCIIOPStream::get	1,121	5
		PMCIIOPStream::op>>(double)	1,118	5
		PMCIIOPStream::op>>(long)	1,118	5

was spent in `memcpy`, 18% in `reads`, and a large percent of its time in demarshalling the `BinStructs`.

The analysis of the performance of the CORBA versions suggests that presentation layer conversions and data copying are the primary areas that must be optimized to achieve higher throughputs.

2.4 Summary

In general, the CORBA implementations measured in this chapter attain lower throughput than the C, C++ wrapper, and hand-optimized RPC versions of TTCP over ATM. On average, the CORBA performance averaged 75% the level of the C/C++ versions for remote transfers of scalars and averaged 33% for `structs` containing binary data. For the loopback tests, the VisiBroker version performed as well as the C/C++ versions for scalar data types at higher sender buffer sizes (*e.g.,* for 128 K sender buffer, the VisiBroker throughput for sending `doubles` was around 196 Mbps which is comparable to the throughput obtained for the C/C++ versions). The Orbix version did not perform as well as the VisiBroker version for transferring scalars (*e.g.,* the Orbix version performed roughly 65-68% as well as the C/C++ versions). Both CORBA implementations performed poorly compared to the C/C++ versions when transferring `structs` containing binary fields. For this type of data Orbix and VisiBroker performed roughly 16% as well as the C/C++ versions.

The CORBA implementations performed worst when sending complex typed data (`structs`) because of excessive copying and marshalling/demarshalling overhead and excessive writes resulting from small size write-buffers. The loopback tests provide a means for testing the performance of CORBA and low-level implementations at higher network speeds. From the loopback results, we conclude that with increasing network speeds, the performance of the CORBA implementations actually becomes worse compared with low-level communication middleware like sockets when marshalling of data is involved. The results in this paper indicate that efficient optimizations need to be applied to the CORBA client-side stubs and server-side skeletons to reduce the marshalling, data copying and request demultiplexing overhead.

We contend that advances in communication middleware like CORBA can be achieved only by simultaneously integrating techniques and tools that simplify application development, optimize application performance, and systematically measure application behavior in order to pinpoint and alleviate performance bottlenecks. Our work is motivated by an increasing demand for efficient and flexible communication software to support next-generation multimedia applications and to leverage emerging high-speed networking technology.

Chapter 3

Throughput Performance of CORBA's Dynamic Invocation and Dynamic Skeleton Interface

3.1 Introduction

This chapter quantifies the performance of two widely used existing CORBA implementations (Orbix 2.0 and VisiBroker 2.0) in terms of their support for the *dynamic invocation interface* (DII) and the *dynamic skeleton interface* (DSI), which are described below.[1]

• **The Dynamic Invocation Interface:** CORBA provides two different interfaces for clients to communicate with servers:

- *Static Invocation Interface (SII)* – is provided by static stubs generated by a CORBA IDL compiler and is useful when client applications know the interface offered by the server at compile-time. The performance evaluation of SII is reported in Chapter 2.
- *Dynamic Invocation Interface (DII)* – is provided by an ORB's dynamic messaging mechanism and is useful when client applications do not have compile-time knowledge of the interfaces offered by servers.

Many distributed applications can be written using CORBA's SII. However, there is an important class of applications (such as network management MIB browsers, configuration management tools and distributed visualization tools and debuggers) that are easier to develop using CORBA's DII. The DII enables applications to construct and invoke CORBA requests at run-time by querying the *Interface Repository* of a server.

[1]The results described in this chapter have appeared in the IEEE Globecom 96 Conference Proceedings.

In addition, the DII is required for applications that use CORBA's *deferred synchronous* model of operation invocation. Although all operation invocations in CORBA are synchronous, there are three models of invocation:

- *Twoway synchronous* – which is the typical request/response model associated with RPC toolkits, where the client blocks after sending the request until the response arrives from the server;
- *Oneway synchronous* – which is a "request-only" messaging model where the client does not receive a response;
- *Deferred synchronous* – this model of two-way call decouples the request from the response so that other client processing can occur until the server sends the response.

The DII supports all three forms of invocation semantics, whereas the SII stubs only support two-way and oneway synchronous calls. Note that neither oneway nor deferred synchronous CORBA calls are asynchronous since the CORBA specification permits the ORB to block while sending a oneway call (*e.g.,* if the underlying transport layer connection is flow controlled).

The first set of experiments in this chapter pinpoint precisely where the key sources of overhead exist in CORBA using the DII. These experiments reveal that the CORBA implementations spent considerable amounts of time in presentation layer conversions for various data types. The experimental results indicate that throughput for different data types is significantly different. This is due to the differences in the amount of overhead in presentation conversion for different data types.

3.2 Additional Features of the CORBA/ATM Testbed

3.2.1 Traffic Generators

Earlier studies [68, 58] tested the performance of transferring untyped bytestream data between hosts using several implementations of CORBA and other lower-level mechanisms like sockets. However, there were several limitations with these experiments:

- *Lack of presentation layer measurements* – typed data reveals the overhead imposed by the presentation layer conversions since it must be marshalled and demarshalled. However, earlier studies only examined untyped data.
- *Lack of DII measurements* – Earlier studies used the CORBA IDL compiler generated SII stubs that are unable to test important CORBA DII based applications such as network management and distributed visualization/debugging tools.

- *Lack of DSI measurements* – Earlier studies used the CORBA IDL compiler generated skeletons for the server side. No measurements were performed to evaluate the performance of the DSI.

In contrast, the experiments described below used richly typed data to identify overheads imposed by the DII and DSI. The experiments conducted for this chapter extend earlier studies by measuring the performance of the Orbix 2.0 and VisiBroker 2.0 implementations of CORBA for two features of CORBA:

• **The Dynamic Invocation Interface:** Both CORBA implementations of the CORBA-TTCP [58] client send 64 MB of richly-typed and untyped data as a `sequence` data type using the DII. The client tests the `invoke` (two-way communication) and `send_oneway` (oneway communication) methods supported by the CORBA `Request` interface. The server uses the skeletons generated by the IDL compiler. To use the DII, a client must first create a request using the CORBA `create_request` method. This request is then populated by the appropriate method name and its parameters in the correct order. In our experiment, we create a request and populate it with the method name and a data buffer of the desired size. We sent the same request repeatedly, only varying the data buffer, until 64 MB of data is sent to the server.

Since the request object (*i.e.,* the operation name and associated internal state) does not change in our tests, it is desirable to reuse the request object on every invocation to amortize the cost of creating the object. However, the CORBA specification [51] does not specify whether a request created using the `create_request` method can be reused or whether it must be created new and populated by the parameters, and released after every invocation.

Orbix 2.0 did not allow reuse of the request. Therefore, we had to create a new request and populate it repeatedly. This had an adverse effect on the performance since the client spends a significant amount of time in the request creation, population, and release. Conversely, VisiBroker 2.0 did allow reuse of the request object, so we could just replace the data buffer. We performed two sets of experiments using the VisiBroker CORBA-TTCP client - both with and without request reuse. As described in Section 3.3, VisiBroker with request reuse twice as well compared with the versions not reusing requests.

• **The Dynamic Skeleton Interface:** The same VisiBroker[2] client in the DII experiment was used to send requests to the VisiBroker CORBA-TTCP server, which uses DSI.

The DII client tests the `invoke` (two-way communication) method[3] supported by the CORBA `Request` interface. We reused the requests in these tests, as described above.

[2]Orbix 2.0 does not yet support DSI, so we could not perform the DSI tests for Orbix.

[3]We observed many requests were not reaching the VisiBroker DSI server since the requests were `oneway`. Hence we did not report results for this tests.

Traffic charactersitics and parameter settings for the experiments were similar to those described in Section 2.2.

3.3 Performance Results

The throughput measurements for the DII and DSI features of CORBA using remote transfers are presented in this section. Detailed profiling measurements of presentation layer, data copying, and memory management overhead are also presented. The profile information was obtained using the `Quantify` performance measurement tool [35].

3.3.1 Dynamic Invocation Interface Measurements

The figures presented in this section depict the observed user-level throughput at the sender for buffer sizes of 1 K, 2 K, 4 K, 8 K, 16 K, 32 K, 64 K, and 128 K bytes using 64 KB sender and receiver socket queues (the maximum possible on SunOS 5.5).

Remote Results:

Figures 3.1 and 3.2 depict the throughput obtained for sending 64 MB of data of different data types for the Orbix and VisiBroker implementations of CORBA-TTCP over ATM using oneway[4] communication without `request` reuse, respectively. Figure 3.3 depicts the throughput for the VisiBroker CORBA-TTCP implementation that reuses the request.

The shape of the CORBA DII performance curves indicate that the observed throughputs differ significantly for different types of data. The observed throughput for a given data type depends on how the CORBA implementation performs presentation layer conversions, data copying and memory management. The throughputs initially increase as the sender buffer size increases due to a decrease in the number of `writes`. However, this throughput increase begins to level off once network-layer fragmentation occurs. For the ATM network we used, the maximum transfer unit (MTU) was 9,180 bytes. Thus, fragmentation starts once the write buffers become greater than the MTU size. Although the MTU is 9,180 bytes, we do not observe the highest throughput for sender buffer sizes of 8 K, since for these buffer sizes, the number of `writes` is large and the presentation layer overhead plays a major role in decreasing the throughput.

The throughput for the oneway transfer case was consistently higher than that of the two-way case. In the oneway transfer, the client does not block for a response from the server after it has sent a request. This allows the client to send the next request immediately. This behavior differs from our CORBA two-way communication tests, where the clients are blocked until they receive an acknowledgement from the server (although no return values

[4]The two-way results are consistently lower than the oneway results since the client in the two-way case blocks for an acknowledgement. Therefore, we omitted the results.

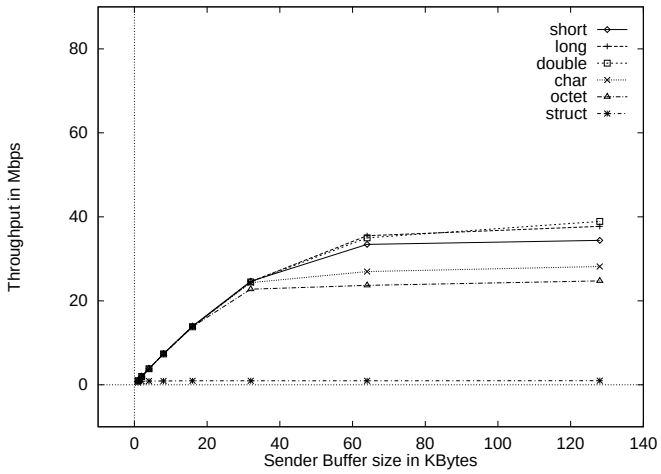


Figure 3.1: Orbix: Client-side Throughput for Oneway Communication using DII without Request Reuse.

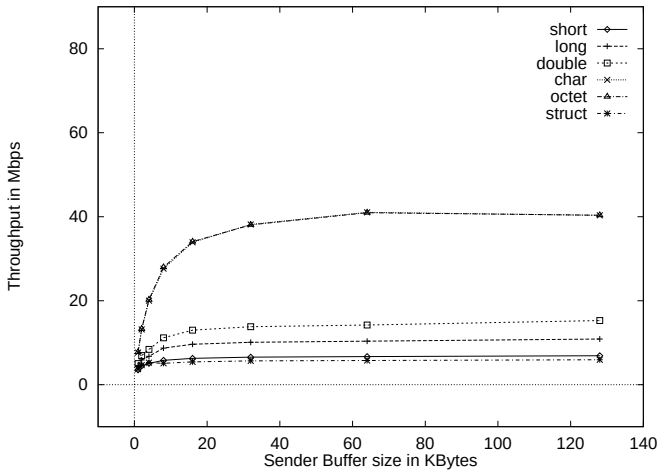


Figure 3.2: VisiBroker: Client-side Throughput for Oneway Communication using DII without Request Reuse.

are sent back). The additional latency associated with this blocking reduced the throughput by roughly 20% for the two-way case. This latency may be hard to avoid, however, if the client requires confirmation that the server has completed the request.

A detailed analysis of the overhead incurred on the client and the server side in sending the `octet` and `struct` (`BinStruct`) data types for the oneway case with 128 K sender buffer sizes is presented in Tables 3.1 and 3.2, respectively. [5] In Tables 3.1 and 3.2, the `Time in msec` column indicates the total time spent by the corresponding operation indicated under the `category` column. The `Percent execution` column indicates the percentage of the total execution time taken up by the method. Table 3.1 reveals that for

[5]The analysis for the two-way case was similar to the oneway case.

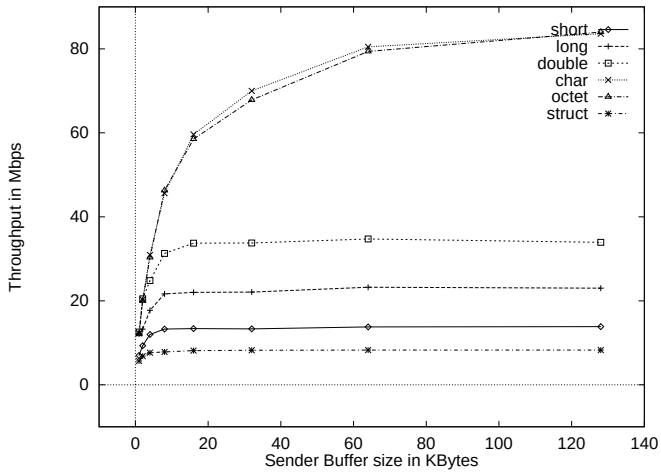


Figure 3.3: VisiBroker: Client-side Throughput for Oneway Communication using DII with Request Reuse.

sending `octet/char`, the Orbix and both versions of VisiBroker spent most of their time in `write` and populating the request.

In contrast, for each of the other data types (only `structs` are shown in the table), the implementations spent a significant amount of time creating a request, populating it with parameters and marshalling the data. For example, to send 64 MB of `structs`, the Orbix version spent 96% of its time in marshalling its parameter and packetizing the data. The throughput for the VisiBroker version without request reuse is comparable to that of Orbix. The VisiBroker version with request reuse performs much better than the VisiBroker version without request reuse and the Orbix version. This is due to the less amount of time spent in marshalling (14,168 msec) and memcpy (896 msec) as opposed to the time taken by the VisiBroker version that does not reuse requests (22,215 msec for marshalling and 13,428 msec for `memcpy`).

In addition, the tables reveal that the throughput for `shorts` is lower than that for `longs` because the CORBA DII implementation spent a larger amount of time doing marshalling and data copying for `shorts` than for `longs`. The lowest throughputs were observed for sending `structs`. The analysis of overheads for the server-side is similar to that of the client-side. The receiver side for data types other than `char/octet` spends a significant amount of time performing `reads` and demarshalling the data.

Loopback Results:

Figures 3.4 and 3.5 depict the throughput obtained for sending 64MB data of various data types for the VisiBroker implementation[6] of CORBA-TTCP over ATM. The loopback results provide an insight into the performance that can be expected of the CORBA DII

[6]Orbix DII did not operate correctly.

Table 3.1: Analysis of Client-side Overheads for CORBA DII

Version	**Data Type**	**Analysis**		
		Category	**Time (msec)**	**Percent Execution**
Orbix	octet	write	26,728	73.52
		Populating and marshalling	7,160	19.70
		Allocate memory	1,164	3.15
		memcpy	895	2.46
	struct	Marshalling and packetizing	397,724	96.6
		write	5,016	1.23
VisiBroker (without request reuse)	octet	writev	4,623	61.14
		memcpy	1,791	23.69
		Allocate memory	940	12.43
	struct	writev	77,605	55.72
		Populating and marshalling	22,215	15.95
		memcpy	13,428	9.64
VisiBroker (with request reuse)	octet	writev	5,794	85.83
		memcpy	898	13.31
	struct	writev	76,575	64.89
		Populating and marshalling	14,168	12.01

Table 3.2: Analysis of Server-side Overheads for CORBA DII

Version	Data Type	Analysis		
		Category	Time (msec)	Percent Execution
Orbix	octet	typecode assertion	3,361	50.00
		read	2,140	32.00
		memcpy	896	13.39
	struct	demarshalling	11,849	49.00
		typecode assertion	6,665	27.57
		read	4,985	20.62
VisiBroker (without request reuse)	octet	read	3,177	90.48
	struct	demarshalling	10,552	58.52
		memcpy	3,878	21.5
		read	3,318	18.4
VisiBroker (with request reuse)	octet	read	3,238	90.60
	struct	demarshalling	10,552	58.52
		memcpy	3,878	21.41
		read	3,395	18.74

implementations for network channel speeds greater than the 155 Mbps ATM network available for our research.

The figures indicate that for sending `octet/chars`, the observed throughput increases as the sender buffer sizes increase. This is due to the lesser number of writes needed as sender buffer size increases. Also, due to the loopback mode, the maximum transfer unit (MTU) of the ATM network (9,180 bytes) does not cause any fragmentation and hence the throughput improves with increasing buffer sizes. The figures also indicate that for the rest of the data types, the presentation layer conversions and data copying operations severely restrict the throughput. A comparison between the `Quantify` analysis for sending `char` and `double` for 64 K sender buffer size reveals that `chars` spend 1,207 msec in `memcpy` as opposed to 7,208 msec taken up by `doubles`. In addition, the marshalling of `chars` take up 0.54 msec as opposed to 4,434 msec for `doubles`. This explains why the throughput for sending `char/octet` is much higher than that for the rest of the data types.

3.3.2 Dynamic Skeleton Interface Measurements

Figure 3.6 depicts the client-side throughput for an VisiBroker client using DII and an VisiBroker server using DSI for two-way transfer.[7]. We did not conduct the experiment for

[7]Oneway transfer results have not been presented since the VisiBroker 2.0 frequently drops `oneway` operations at the server. Although this behavior is allowed by the "best effort" semantics of CORBA `oneway` operations, it appears to be a bug in the VisiBroker implementation.

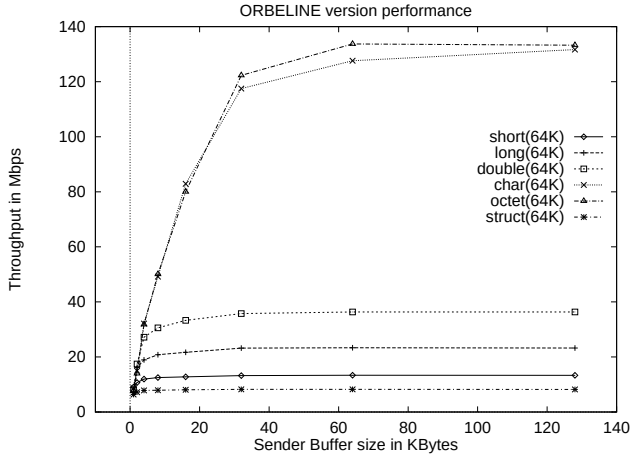


Figure 3.4: Client-side Throughput for Oneway Communication using DII in Loopback Mode.

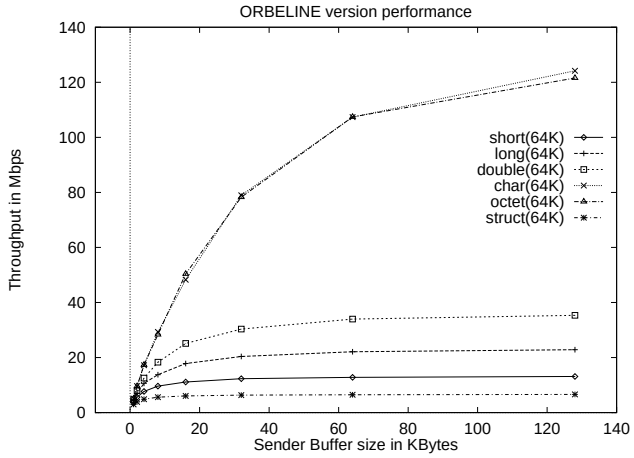


Figure 3.5: Client-side Throughput for Twoway Communication using DII in Loopback Mode.

Orbix since the version we tested did not support DSI. The shape of the CORBA DSI performance curves indicate that the observed throughputs differ significantly for different types of data. The throughput for a given data type depends on how the CORBA implementation performs presentation layer conversions, data copying and memory management.

Of all the CORBA data types we tested, the `chars` and `octets` performed the best. This is not surprising, and results from the fact that presentation layer conversions and data copying can be minimized for this "untyped" data (though both ORBs do perform multiple data copies for `chars` and `octets`). Throughput initially increases as the sender buffer size increases due to a decrease in the number of `writes` and `reads`. However, the increased throughput begins to level off once network-layer fragmentation occurs. For our ATM network, the maximum transfer unit (MTU) was 9,180 bytes. Thus, fragmentation occurs once the write buffers become greater than the MTU size. Although the MTU is

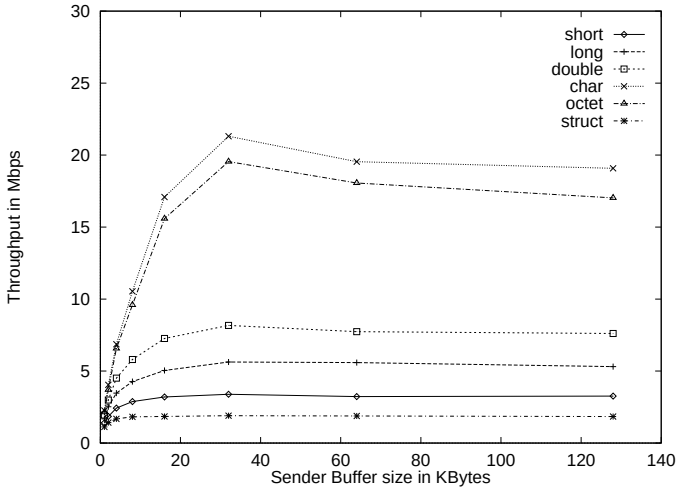


Figure 3.6: Observed Client Throughput using VisiBroker DSI

9,180 bytes, we do not observe the highest throughput for sender buffer sizes of 8K. For these buffer sizes, the number of `writes` is large and the presentation layer overhead plays a major role in decreasing the throughput.

Table 3.3 depicts the time spent by the sender and receiver of the VisiBroker version of CORBA-TTCP using DSI when transferring 64 Mbytes of sequences using 128 K sender and receiver buffers and 64 K socket queues. An analysis for `octets` (best throughput) and `structs` (worst throughput) is presented. These figures precisely pinpoint the time spent in marshalling, data copying, and reading and writing of buffers. The excessive amount of time taken for `writes` suggests that the ORBs use non-optimal buffer management and flow control methods.

The analysis of the performance of the CORBA implementations suggests that optimized read/writes, presentation layer conversions and data copying are the primary areas that must be optimized to achieve higher throughputs.

3.4 Summary

This chapter illustrates how existing CORBA implementations incur considerable overhead when application use the dynamic invocation interface (DII) and the dynamic skeleton interface (DSI) over high-speed ATM networks. Our DII and DSI results indicate that both Orbix 2.0 and VisiBroker 2.0 incur different levels of presentation conversion overhead for different CORBA data types. Moreover, non-optimal internal buffer management strategies lead to large amounts of time spent in network writes and reads.

Finally, the experiments indicate that for DII, it is desirable to permit reuse of requests if the operations are `oneway` and the parameter values do not change. The current

Table 3.3: Analysis of Overhead using Twoway Client-side DII and Server-Side DSI

Agent	**Data Type**	**Analysis**		
		Category	**Time (msec)**	**Percent Execution**
Client	octet	read	3,911	47.95
		writev	2,923	35.84
		memcpy	912	11.19
	struct	writev	505,914	91.34
		marshalling and populating	13,341	2.41
		memcpy	7,198	1.30
		read	5,062	0.91
Server	octet	writev	23,572	82.00
		memcpy	2,701	9.40
		read	1,806	6.29
	struct	writev	1,250,130	92.41
		demarshalling	39,373	2.89
		memcpy	19,693	1.46

CORBA 2.0 specification does not define whether this is legal or not, so different implementations interpret the specification differently. Not only does this impede portability across ORB implementations, but it also permits non-optimal performance if requests are not reused.

Chapter 4

CORBA Latency and Scalability Over High-speed Networks

4.1 Introduction

The success of CORBA in mission-critical distributed computing environments depends heavily on the ability of Object Request Brokers (ORBs) to provide the necessary quality of service (QoS) to applications.

Our earlier work [20, 21, 58, 25] has focused on measuring and optimizing the *throughput* of CORBA ORBs. This chapter extends our prior work by measuring the *latency* and *scalability* of two widely used CORBA ORBs, Orbix 2.1 and VisiBroker 2.0, precisely pinpointing their key sources of overhead, and describing how to systematically remove these sources of overhead by applying optimization principles.[1]

The contributions of this chapter include the following:

• **CORBA Latency** Section 4.3 measures the oneway and two-way latency of Orbix and VisiBroker. Our findings indicate that the latency overhead in these ORBs stem from (1) long chains of intra-ORB function calls, (2) excessive presentation layer conversions and data copying, and (3) non-optimized buffering algorithms used for network reads and writes. Chapter 5 describes optimizations we have developed to reduce these common sources of ORB latency.

• **CORBA Scalability** Section 4.3 also measures the scalability of Orbix and VisiBroker to determine how the number of objects supported by a server affects an ORB's ability to process client requests efficiently and predictably. Our findings indicate that scalability impediments are due largely to (1) inefficient server demultiplexing techniques and (2)

[1]The results described in this chapter have appeared in the IEEE ICDCS 97 Conference Proceedings and the IEEE Computer Society's Transactions on Computers, April 98.

lack of integration with OS and network features. Chapter 6 describes demultiplexing optimizations we have developed to increase ORB scalability.

4.2 Experimental Setup and Testbed Environment

4.2.1 Traffic Generators

Our earlier studies [20, 21, 58] tested bulk data performance using "flooding models" that transferred untyped bytestream data, as well as richly typed data between hosts using several CORBA ORBs and lower-level mechanisms like sockets. On the client-side, these experiments measured the static invocation interface (SII) and the dynamic invocation interface (DII) provided by the CORBA ORBs.

The SII allows a client to invoke a remote operation via static stubs generated by a OMG IDL compiler. The SII is useful when client applications know the interface offered by the server at compile-time. In contrast, the DII allows a client to access the underlying request mechanisms provided by an ORB directly. The DII is useful when the applications do not know the interface offered by the server until run-time.

The experiments conducted for this chapter extend our earlier throughput studies by measuring end-to-end latency incurred when invoking operations with a range of data types and sizes on remote servant(s). In addition, we measure CORBA scalability by determining the demultiplexing overhead incurred when increasing the number of servants in an endsystem server process.

Traffic for the latency experiment was generated and consumed by an enhanced version of TTCP [74]. We measured round-trip latency using the two-way CORBA operations. We first measured operations that did not use any parameters to determine the "best case" operation latency. In addition, we measured operation transfers using following data types: primitive types (`short`, `char`, `long`, `octet`, `double`) and a C++ `struct` composed of all the primitives (`BinStruct`). The CORBA ORBs transferred the data types using IDL `sequences`, which are dynamically-sized arrays. The following OMG IDL interface was used by the CORBA ORBs for the latency tests reported in this chapter:

```
struct BinStruct{ short s; char c; long  l;
                  octet o; double d; };
interface ttcp_sequence
{
  typedef sequence<BinStruct> StructSeq;
  typedef sequence<octet> OctetSeq;

  // Routines to send sequences of various data types
  void sendStructSeq_2way (in  StructSeq ttcp_seq);
  void sendOctetSeq_2way (in  OctetSeq ttcp_seq);
  void sendNoParams_2way();
  void sendNoParams_1way();
};
```

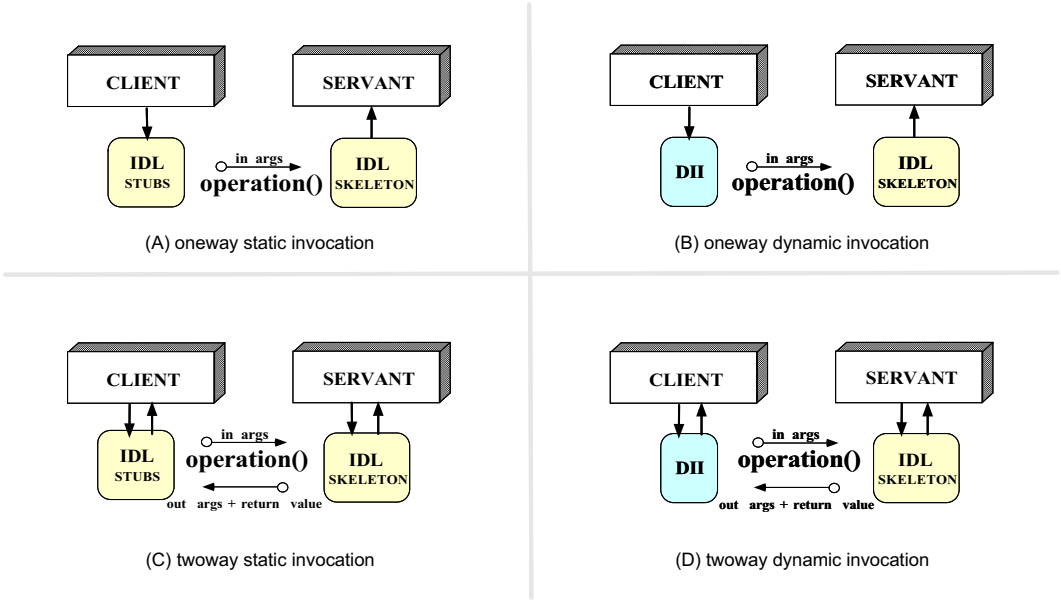


Figure 4.1: CORBA Operation Invocation Strategies

4.2.2 Additional TTCP Parameter Settings

• **Data buffer size** For the latency measurements, the sender transmits parameter units of a specific data type incremented in powers of two, ranging from 1 to 1,024. Thus, for `shorts` (which are two bytes on SPARCs), the sender buffers ranged from 2 bytes to 2,048 bytes. In addition, we measured the latency of remote operation invocations that had no parameters.

• **Number of servants** Increasing the number of servants on the server increases the demultiplexing effort required to dispatch the incoming request to the appropriate servant. To pinpoint the latency overhead in this demultiplexing process, and to evaluate the scalability of CORBA implementations, our experiments used a range of servants (1, 100, 200, 300, 400, and 500) on the server.

4.2.3 Operation Invocation Strategies

One source of latency incurred by CORBA ORBs in high-speed networks involves the *operation invocation strategy*. This strategy determines whether the requests are invoked via the static or dynamic interfaces and whether the client expects a response from the server. In our experiments, we measured the following operation invocation strategies defined by the CORBA specification, which are shown in Figure 4.1.

• **Oneway static invocation** The client uses the SII stubs generated by the OMG IDL compiler for the oneway operations defined in the IDL interface, as shown in Figure 4.1 (A).

• **Oneway dynamic invocation** The client uses the DII to build a request at run-time and uses the CORBA `Request` class to make the requests, as shown in Figure 4.1 (B).

• **Twoway static invocation** The client uses the static invocation interface (SII) stubs for two-way operations defined in IDL interfaces, as shown in Figure 4.1 (C).

• **Twoway dynamic invocation** The client uses the dynamic invocation interface (DII) to make the requests, but blocks until the call returns from the server, as shown in Figure 4.1 (D).

We measured the average latency for 100 client requests for groups of 1, 100, 200, 300, 400, and 500 servants on the server. In every request, we invoked the same operation. We restricted the number of requests per servant to 100 since neither Orbix nor VisiBroker could handle a larger numbers of requests without crashing, as described in Section 4.3.4.

4.2.4 Servant Demultiplexing Strategies

Another source of overhead incurred by CORBA ORBs involves the time the Object Adapter spends demultiplexing requests to servants. The type of demultiplexing strategy used by an ORB significantly affects its scalability. Scalability is important for applications ranging from enterprise-wide network management systems, with agents containing a potentially large number of servants on each ORB endsystem, to real-time avionics mission computers, which must support real-time scheduling and dispatching of periodic processing operations.

A standard GIOP-compliant client request contains the identity of its remote object and remote operation. A remote object is represented by an Object Key `octet sequence` and a remote operation is represented as a `string`. Conventional ORBs demultiplex client requests to the appropriate operation of the servant implementation using *layered demultiplexing* explained in Section 6.1.

However, layered demultiplexing is generally inappropriate for high-performance and real-time applications for the following reasons [73]:

Decreased efficiency: Layered demultiplexing reduces performance by increasing the number of internal tables that must be searched as incoming client requests ascend through the processing layers in an ORB endsystem. Demultiplexing client requests through all these layers is expensive, particularly when a large number of operations appear in an IDL interface and/or a large number of servants are managed by an Object Adapter.

Increased priority inversion and non-determinism: Layered demultiplexing can cause priority inversions because servant-level QoS information is inaccessible to the lowest-level device drivers and protocol stacks in the I/O subsystem of an ORB endsystem. Therefore, an Object Adapter may demultiplex packets according to their FIFO order of arrival.

FIFO demultiplexing can cause higher priority packets to wait for an indeterminate period of time while lower priority packets are demultiplexed and dispatched.

Conventional implementations of CORBA incur significant demultiplexing overhead. For instance, [20, 23] show that conventional ORBs spend ∼17% of the total server time processing demultiplexing requests. Unless this overhead is reduced and demultiplexing is performed predictably, ORBs cannot provide uniform, scalable QoS guarantees to real-time applications.

Prior work [24] analyzed the impact of various IDL skeleton demultiplexing techniques such as linear search and direct demultiplexing. However, in many applications the number of operations defined per-IDL interface is relatively small and static, compared to the number of potential servants, which can be quite large and dynamic.

To evaluate the scalability of the CORBA ORBs in this chapter, we varied the number of servants residing in the server process from 1 to 500, by increments of 100. The server used the *shared* activation mode, where all servants on the server are managed by the same process.

4.2.5 Request Invocation Algorithms

The experiments conducted for this chapter use four different request invocation algorithms on the client-side. Each invocation algorithm evaluates the merits of the server-side Object Adapter's strategy for demultiplexing incoming client requests. The experiments conducted for the Orbix 2.1 and VisiBroker 2.0 ORBs use the *request train* and *round robin* invocation algorithms described below.

Chapter 6 describes how we applied active demultiplexing and perfect hashing to optimized demultiplexing in our high-performance, real-time ORB called TAO [69, 24]. To test these strategies, we developed two additional request invocation algorithms called *random invocation* and *worst-case invocation*, respectively. These algorithms are used to evaluate the predictability, consistency, and scalability properties of TAO's demultiplexing strategies, and to compare their performance with the worst-case performance of linear-search demultiplexing. All the four invocation algorithms are described below.

The Request Train Invocation Algorithm

One way to optimize demultiplexing overhead is to have the Object Adapter cache recently accessed servants. Caching is particularly useful if client operations arrive in "request trains," where a server receives a series of requests for the same servant. By caching information about a servant, the server can reduce the overhead of locating the servant for every incoming request.

To determine if caching was used, and to measure its effectiveness, we devised the following request invocation algorithm:

```
const int MAXITER = 100;

long sum = 0;
Profile_Timer timer; // Begin timing.

for (int j = 0; j < num_servants; j++){
  for (int i = 0; i < MAXITER; i++) {
     // Use one of the 2 invocation strategies
     // to call the send() operation on servant_#j
     // at the server...
     sum += timer.current_time ();
  }
}
avg_latency = sum / (MAXITER * num_servants);
```

This algorithm does not change the destination servant until `MAXITER` requests are performed. If a server is caching information about recently accessed servants, the request train algorithm should elicit different performance characteristics than the round robin algorithm described next.

The Round Robin Invocation Algorithm

In this scheme, the client invokes the `send` operation `MAXITER` times on a different object reference. This algorithm is used to evaluate how predictable, consistent, and scalable is the demultiplexing technique used by the Object Adapter. The round robin algorithm is defined as follows:

```
const int MAXITER = 100;

long sum = 0;
Profile_Timer timer; // Begin timing.

for (int i = 0; i < MAXITER; i++){
  for (int j = 0; j < num_servants; j++) {
     // Use one of the 2 invocation strategies
     // to call the send() operation on servant_#j
     // at the server...
     sum += timer.current_time ();
  }
}
avg_latency = sum / (MAXITER * num_servants);
```

Random Invocation Algorithm

The random invocation algorithm is different than the round robin algorithm since requests are made on a randomly chosen object reference for a randomly chosen operation as shown below:

```
const int MAXITER = 100;
```

```
long sum = 0;
Profile_Timer timer; // Begin timing.

for (int i = 0; i < num_servants; i++) {
  for (int j = 0; j < NUM_OPERATIONS; j++) {
     // choose a servant at random from
     //    the set [0, NUM_SERVANTS - 1];
     // choose an operation at random from
     //    the set [0, NUM_OPERATIONS - 1];
     // invoke the operation on that servant;
  }
}
avg_latency = sum / (MAXITER * num_servants);
```

The pattern of requests generated by this scheme is different from the well-defined pattern of requests made by the round robin algorithm. The random invocation algorithm is thus better suited to test the predictability, scalability, and consistency properties of the demultiplexing techniques used than the round robin Invocation algorithm.

Worst-case Invocation Algorithm

In this scheme, we choose the last operation of the last servant. The algorithm for sending the worse-case client requests is shown below:

```
const int MAXITER = 100;

long sum = 0;
Profile_Timer timer; // Begin timing.

for (int i = 0; i < num_servant; i++) {
  for (int j = 0; j < NUM_OPERATIONS; j++) {
     // invoke the last operation on the
     //   last servant
  }
}
avg_latency = sum / (MAXITER * num_servants);
```

The purpose of this scheme is to compare the performance of different demultiplexing schemes with that of the worst-case behavior depicted by a linear-search based scheme.

4.3 Performance Results for CORBA Latency and Scalability over ATM

This section presents the performance results from our latency and scalability experiments. Sections 4.3.1 and 4.3.2 describe the blackbox experiments that measure end-to-end communication delay from client requester to a range of servants using a variety of types and sizes of data. Section 4.3.3 describes a whitebox empirical analysis using `Quantify` to precisely pinpoint the overheads that yield these results. Our measurements include the overhead imposed by all the layers shown in Figure 4.2.

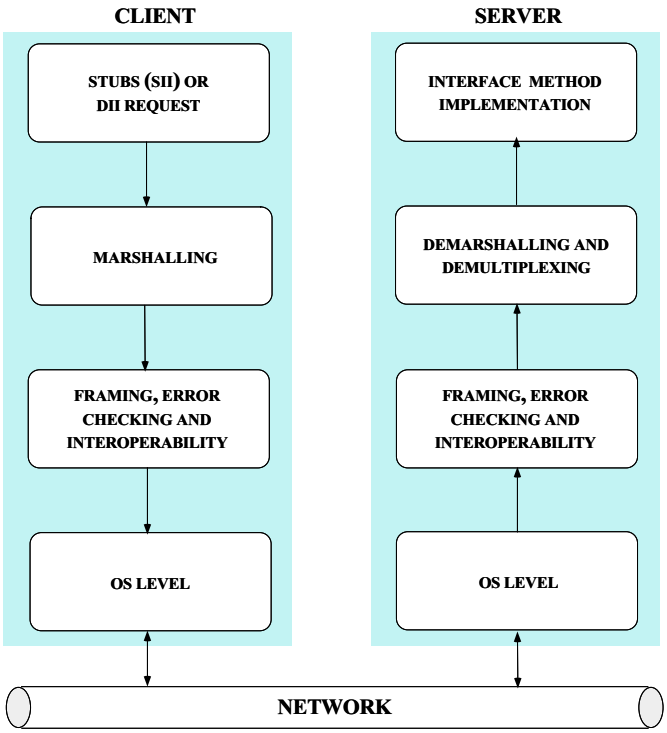


Figure 4.2: General Path of CORBA Requests

4.3.1 Blackbox Results for Parameterless Operations

In this section, we describe the results for invoking parameterless operations using the round robin and request train invocation strategies.

Latency and Scalability of Parameterless Operations

Figures 4.3 and 4.4 depict the average latency for the parameterless operations using the request train variation of our request invocation algorithm. Likewise, Figures 4.5 and 4.6 depict the average latency for invoking parameterless operations using the round robin algorithm.

These figures reveal that the results for the request train experiment and the round robin experiment are essentially identical. Thus, it appears that neither ORB supports caching of servants. As a result, the remainder of our tests just use the round robin algorithm.

Twoway latency The figures illustrate that the performance of VisiBroker was relatively constant for two-way latency. In contrast, Orbix's latency grew as the number of servants increased. The rate of increase was approximately 1.12 times for every 100 additional servants on the server.

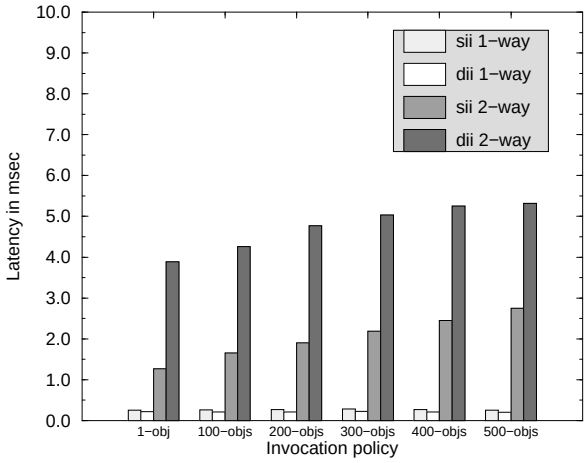


Figure 4.3: Orbix: Latency for Sending Parameterless Operation using request train Requests

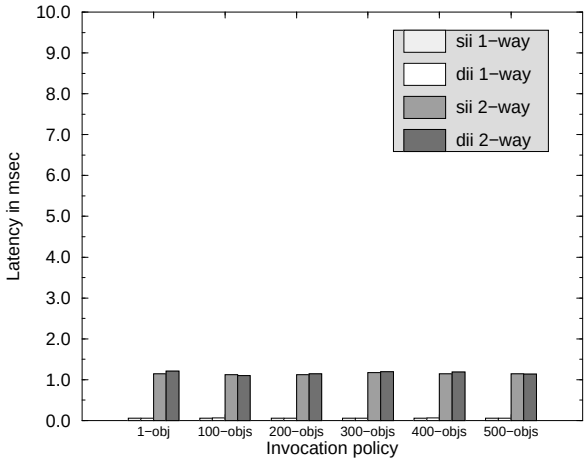


Figure 4.4: VisiBroker: Latency for Sending Parameterless Operation using request train Requests

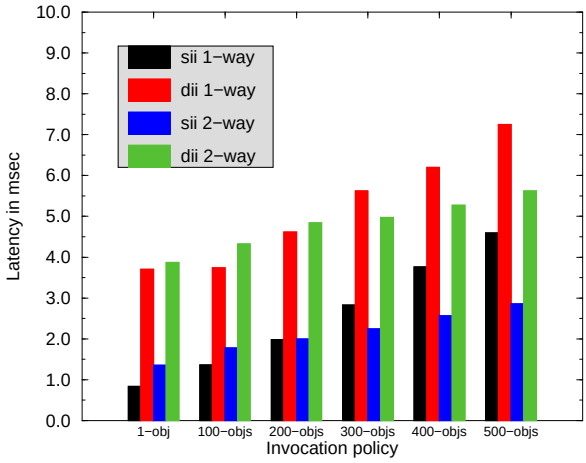


Figure 4.5: Orbix: Latency for Sending Parameterless Operation using Round Robin Requests

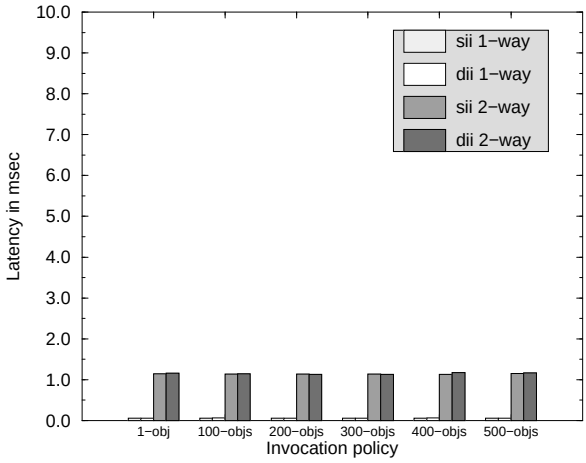


Figure 4.6: VisiBroker: Latency for Sending Parameterless Operation using round robin Requests

We identified the problem with Orbix by using `truss`, which is a Solaris tool for tracing the system calls at run-time. When Orbix 2.1 is run over ATM networks, it opens a new TCP connection (and thus a new socket descriptor) for every object reference.[2] This behavior has the following consequences:

- *Increased demultiplexing overhead* – Opening a new socket connection for each object reference degrades latency significantly since the OS kernel must search the socket endpoint table to determine which descriptor should receive the data.
- *Limited scalability* – As the number of servants grew, all the available descriptors on the client and server were exhausted. We used the UNIX `ulimit` command to increase the number of descriptors to 1,024, which is the maximum supported per-process on SunOS 5.5 without reconfiguring the kernel. Thus, we were limited to approximately 1,000 object references per-server process on Orbix over ATM.

In contrast, VisiBroker did not create socket descriptors for every object reference. Instead, a single connection and socket descriptor were shared by all object references in the client. Likewise, a single connection and socket descriptor were shared by all servant implementations in the server. This, combined with its hashing-based demultiplexing scheme for locating servants and operations, significantly reduces latency. In addition, we were able to obtain object references for more than 1,000 servants.

Oneway latency The figures illustrate that for VisiBroker, the oneway latency remains nearly constant as the number of servants on the server increase. In contrast, Orbix's latency grows as the number of servants increase.

Figures 4.3 and 4.5 reveal an interesting case with Orbix's oneway latency. The oneway SII and DII latencies remain slightly less than their corresponding two-way latencies until 200 servants on the server. Beyond this, the oneway latencies exceed their corresponding two-way latencies.

The reason for Orbix's behavior is that it opens a new TCP connection (and allocate a new socket descriptor) for every object reference. Since the oneway calls do not involve any server response to the client, the client can send requests without blocking. As explained in Section 4.3.3, the receiver is unable to keep pace with the sender due to the large number of open TCP connections and inefficient demultiplexing strategies. Consequently, the underlying transport protocol, TCP in this case, must invoke flow control techniques to slow down the sender. As the number of servants increase, this flow control overhead becomes dominant, which increases oneway latency.

In contrast, the two-way latency does not incur this flow control overhead since the sender blocks for a response after every request.

[2]Interestingly, when the Orbix client is run over Ethernet it only uses a single socket on the client, regardless of the number of servants in the server process.

Figure 4.7 compares the two-way latencies obtained for sending parameterless operations for Orbix and VisiBroker with that of a low-level C++ implementation that uses sockets directly. The two-way latency comparison reveals that the VisiBroker and Orbix

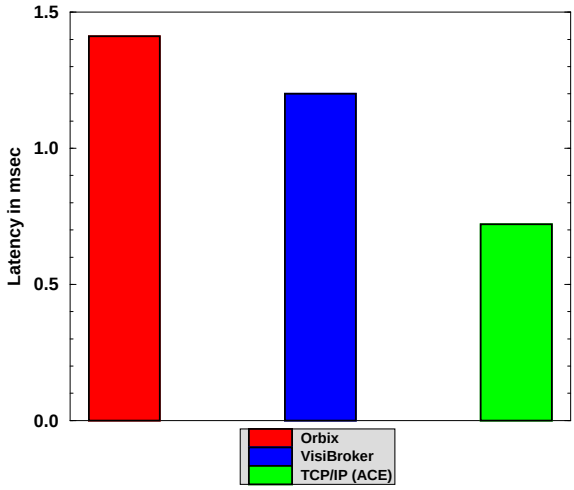


Figure 4.7: Comparison of Twoway Latencies

versions perform only 50% and 46% as well as the C++ version, respectively.

Summary of Results for Parameterless Operations

- Neither Orbix nor VisiBroker caches recently accessed object references in the Object Adapter. As a result, the latency for the request train and round robin cases are nearly equivalent.
- Oneway latencies for VisiBroker remained relatively constant as the number of servants increase on the server. However, the oneway latency for Orbix increases linearly as the number of servants grow.
- Oneway latencies for Orbix exceed their corresponding two-way latencies beyond 200 servants on the server. This is due to the flow control mechanism used by the underlying TCP transport protocol to throttle the fast sender.
- Twoway latency for VisiBroker remains relatively constant as the number of servants increases. This is due to the efficient demultiplexing based on hashing used by VisiBroker. Moreover, unlike Orbix, VisiBroker does not open a new connection for every object reference.
- Twoway latency for Orbix increases linearly at a rate of ∼1.12 per 100 servant increment. As explained earlier, this stems from Orbix's inefficient demultiplexing strategy and the fact that it opens TCP connection per object reference.

- Twoway DII latency in VisiBroker is comparable to its two-way SII latency. This is due to its reuse of DII requests, thereby only creating the request once.
- Twoway DII latency in Orbix is ∼2.6 times that of its two-way SII latency. In Orbix DII, a new request must be created per invocation.[3]
- Twoway DII latency for Orbix is always greater than its two-way SII latency, whereas for VisiBroker they are comparable. The reasons is that Orbix creates a new request for every DII invocation. In contrast, VisiBroker recycles the request.

4.3.2 Blackbox Results for Parameter Passing Operations

In this section, we describe the results for invoking parameter passing operations using the round robin and request train invocation strategies.

Latency and Scalability of Parameter Passing Operations

Figures 4.8 through 4.15 depict the average latency for sending richly-typed `struct` data and untyped `octet` data using (1) the oneway operation invocation strategies (described in Section 4.2.3) and (2) varying the number of servants (described in Section 4.2.4). Similarly, Figures 4.16 through 4.23 depict the average latency for sending richly-typed `struct` data and untyped `octet` for two-way operations. These figures reveal that as the sender buffer size increases, the marshaling and data copying overhead also grows [20, 21], thereby increasing latency. These results demonstrate the benefit of using more efficient buffer management techniques and highly optimized stubs [13] to reduce the presentation conversion and data copying overhead.

Oneway latency The oneway SII latencies for Orbix and VisiBroker for `octets` are comparable. However, as depicted in Figures 4.9 through 4.13, due to inefficient internal buffering strategies, there is substantial variance in latency. This jitter is generally unacceptable for real-time systems that require predictable behavior [31].

The Orbix DII latency for `octets` is nearly double the latency for VisiBroker. The oneway SII latencies for Orbix and VisiBroker for `BinStructs` are comparable. However, the oneway DII latency for Orbix increases rapidly compared to that of VisiBroker. For 500 servants, the Orbix oneway DII latency for `BinStructs` is ∼5.6 times that of VisiBroker.

Twoway latency Figures 4.16 through 4.23 reveal that the two-way latency for Orbix increases as (1) the number of servants and (2) the sender buffer sizes increase. In contrast, for VisiBroker the latency increases only with the size of sender buffers. Figures 4.20 through 4.23 also reveal that the latency for the Orbix two-way SII case at 1,024 data units of `BinStruct` is almost 1.2 times that for VisiBroker.

[3]The CORBA 2.0 specification does not dictate whether a new DII request should be created for each request, so an ORB may chose to use either approach.

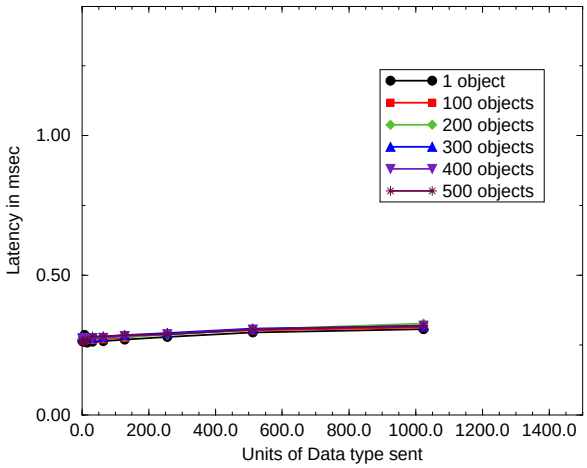


Figure 4.8: Orbix Latency for Sending Octets Using Oneway SII

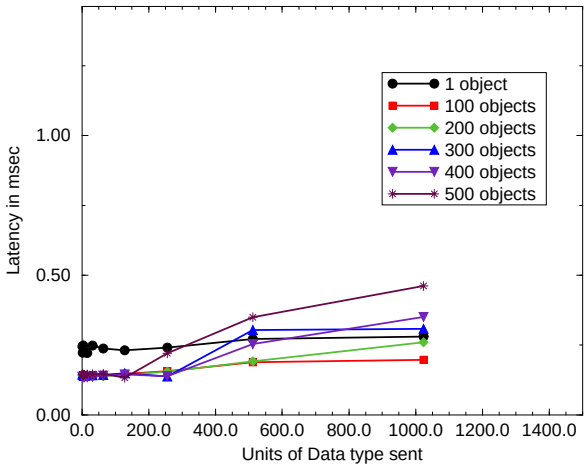


Figure 4.9: VisiBroker Latency for Sending Octets Using Oneway SII

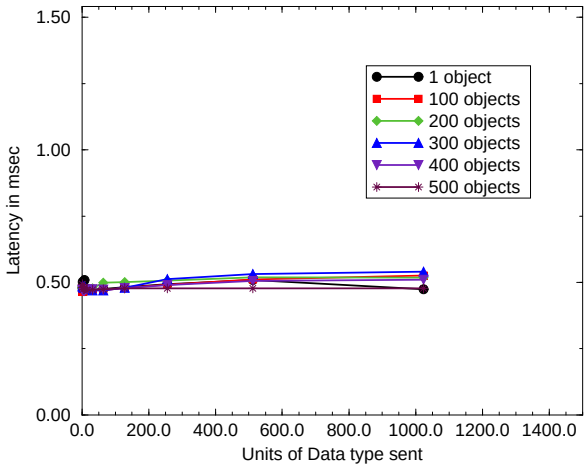


Figure 4.10: Orbix Latency for Sending Octets Using Oneway DII

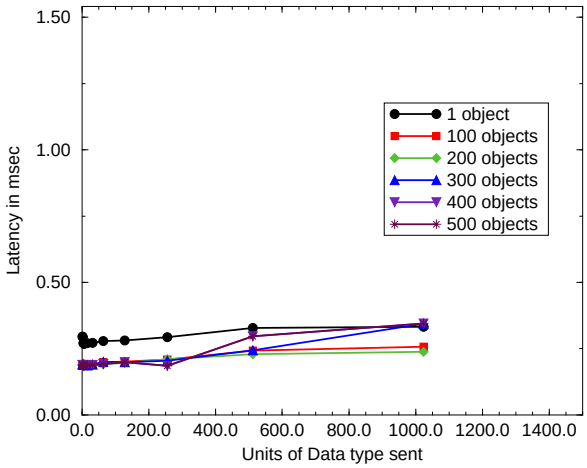


Figure 4.11: VisiBroker Latency for Sending Octets Using Oneway DII

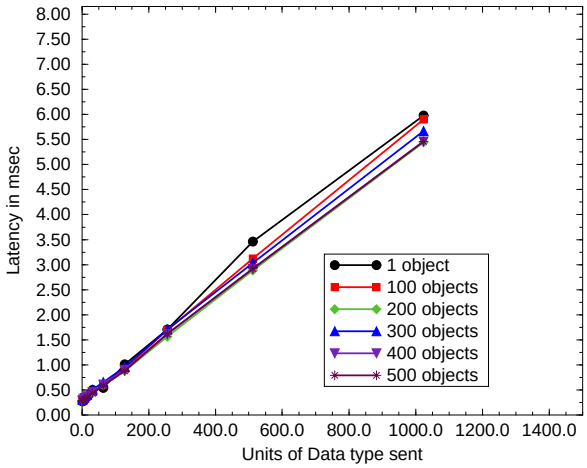


Figure 4.12: Orbix Latency for Sending Structs Using Oneway SII

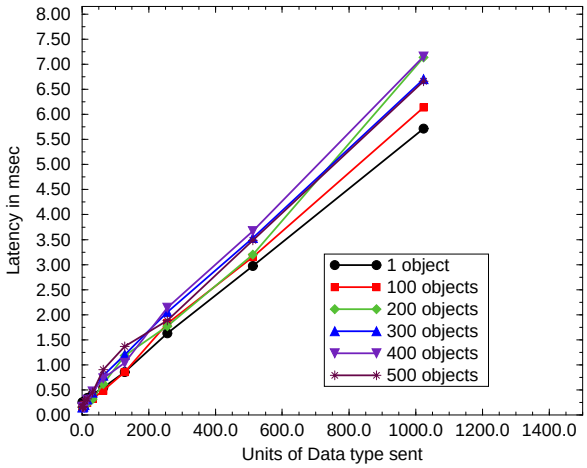


Figure 4.13: VisiBroker Latency for Sending Structs Using Oneway SII

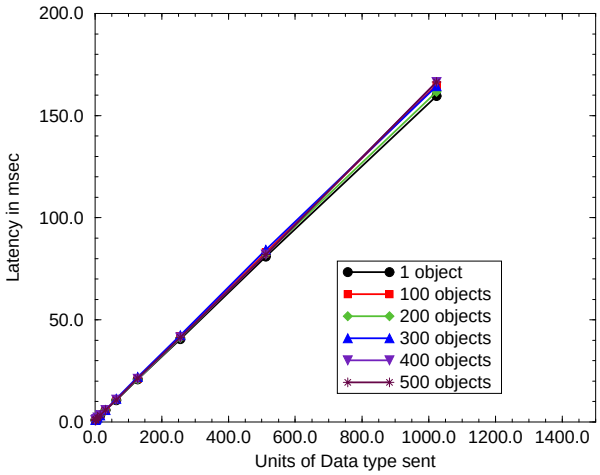


Figure 4.14: Orbix Latency for Sending Structs Using Oneway DII

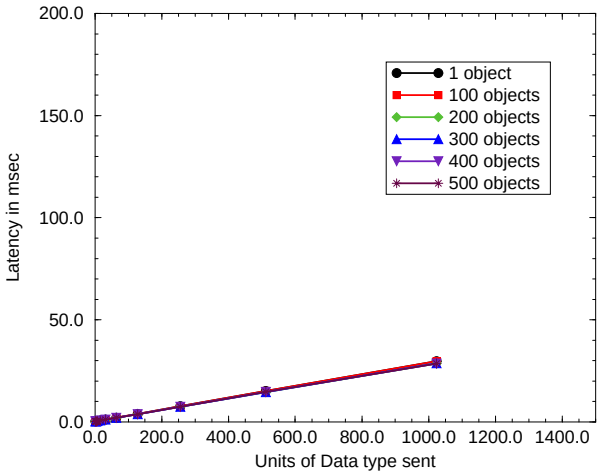


Figure 4.15: VisiBroker Latency for Sending Structs Using Oneway DII

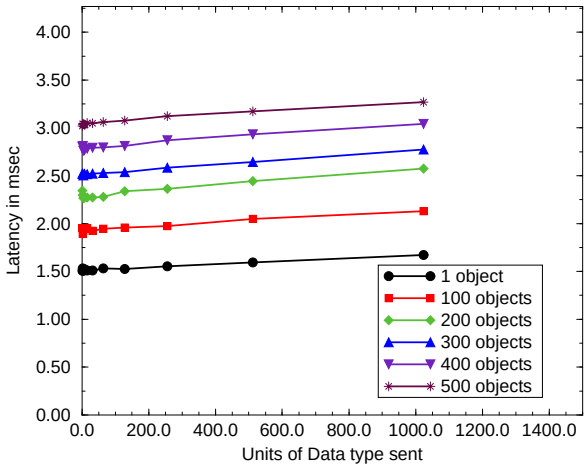


Figure 4.16: Orbix Latency for Sending Octets Using Twoway SII

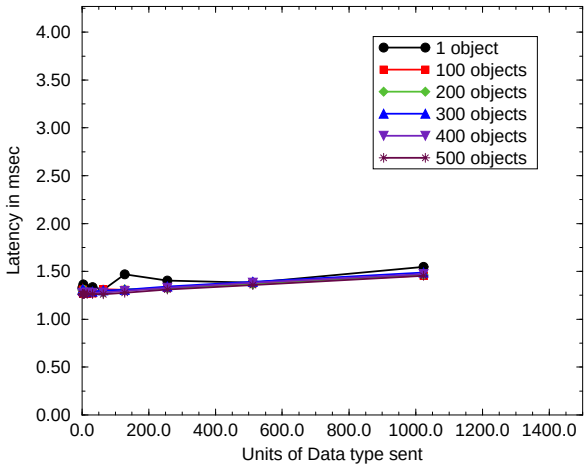


Figure 4.17: VisiBroker Latency for Sending Octets Using Twoway SII

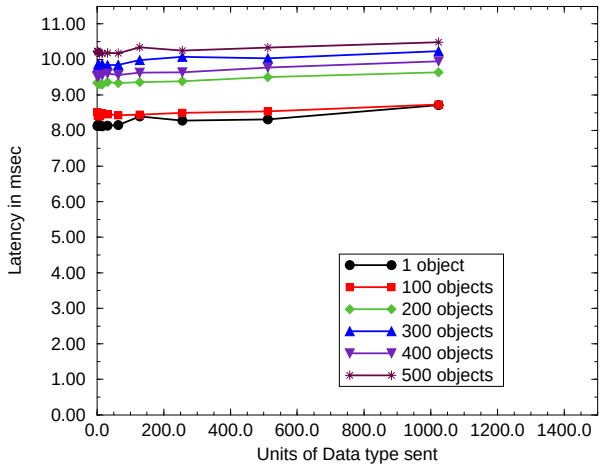


Figure 4.18: Orbix Latency for Sending Octets Using Twoway DII

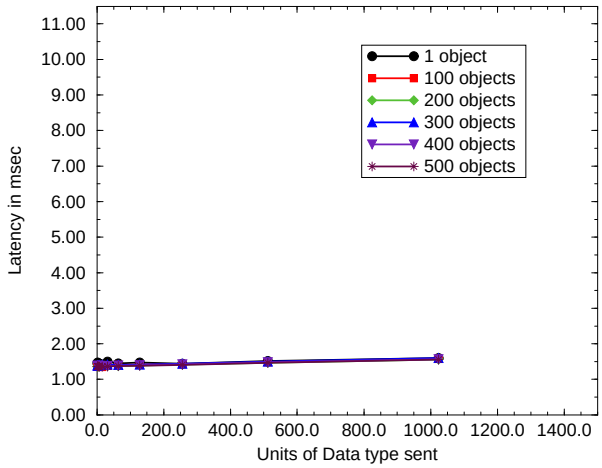


Figure 4.19: VisiBroker Latency for Sending Octets Using Twoway DII

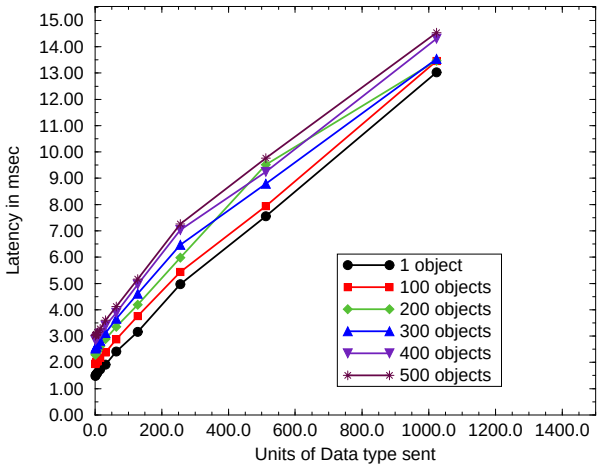


Figure 4.20: Orbix Latency for Sending Structs Using Twoway SII

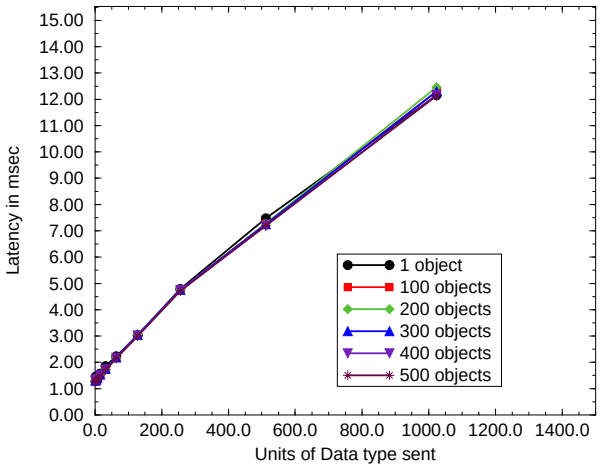


Figure 4.21: VisiBroker Latency for Sending Structs Using Twoway SII

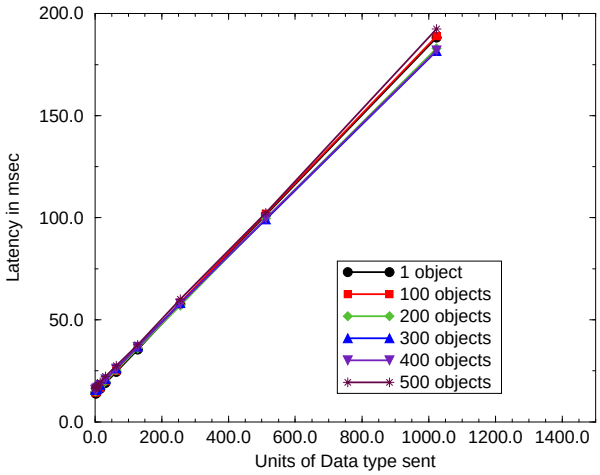


Figure 4.22: Orbix Latency for Sending Structs Using Twoway DII

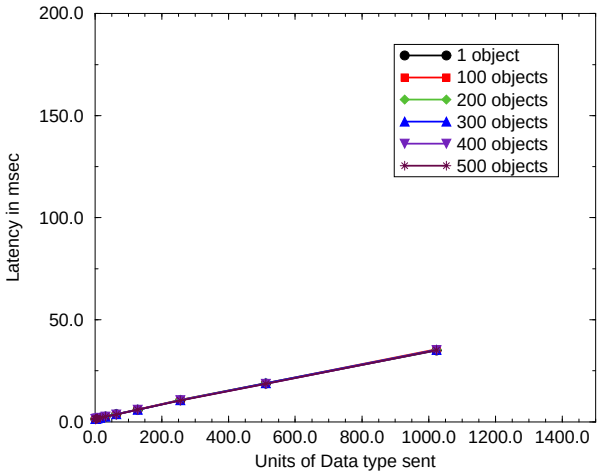


Figure 4.23: VisiBroker Latency for Sending Structs Using Twoway DII

Similarly, the latency for the Orbix two-way DII case at 1,024 data units of `BinStruct` is almost 4.5 times that for VisiBroker. In addition, the figures reveal that for Orbix, the latency increases as the number of servants increase. As shown in 4.3.3, this is due to the inefficient demultiplexing strategy used by Orbix. For VisiBroker, the latency remains unaffected as the number of servants increases.

Orbix incurs higher latencies than VisiBroker due to (1) the additional overhead stemming from the inability of Orbix DII to reuse requests and (2) the presentation layer overhead of marshaling and demarshaling the `BinStructs`. These sources of overhead reduce the receiver's performance, thereby triggering the flow control mechanisms of the transport protocol, which impede the sender's progress.

[20, 21] precisely pinpoint the marshaling and data copying overheads when transferring richly-typed data using SII and DII. The latency for sending `octets` is much less than that for `BinStructs` due to significantly lower overhead of presentation layer conversions. Section 4.3.3 presents our analysis of the `Quantify` results for sources of overhead that increase the latency of client request processing.

Summary of Latency and Scalability Results for CORBA Parameter Passing Operations

The following summarizes the latency results for parameter passing operations described above:

- Latency for Orbix and VisiBroker increases linearly with the size of the request. This is due to the increased parameter marshaling overhead.
- VisiBroker exhibits relatively low, constant latency as the number of servants increase, due to its use of hashing-based demultiplexing for servants and IDL skeletons at the receiver. In contrast, Orbix exhibits linear increases in latency based on the number of servants and the number of operations in an IDL interface. This behavior stems from Orbix's use of linear search at the TCP/socket layer since it opens a connection per object reference. In addition, it also uses linear search to locate servant operations in its IDL skeletons.
- The DII performs consistently worse than SII. For two-way Orbix operations the DII performs 3 times worse for `octets` and 14 times worse for `BinStructs`. For VisiBroker the DII performs comparable for `octets` and ∼4 times worse for `BinStructs`).

  Since Orbix does not reuse the DII requests, the DII latency for Orbix incurs additional overhead. However, both Orbix and VisiBroker must populate the request with parameters. This involves marshaling and demarshaling the parameters. The marshaling overhead for `BinStructs` is more significant than that for `octets`. As a result, the DII latency for `BinStructs` is worse compared to that of `octets`.

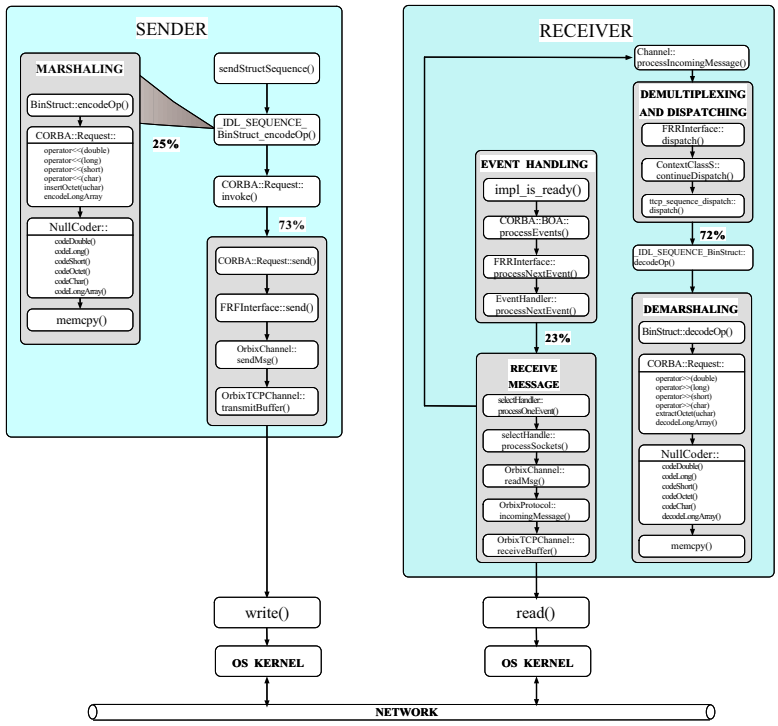


Figure 4.24: Request Path Through Orbix Sender and Receiver for SII

4.3.3 Whitebox Analysis of Latency and Scalability Overhead

Sections 4.3.1 and 4.3.2 presented the results of our blackbox performance experiments. These results depict *how* the two ORBs perform. However, the blackbox tests do not explain *why* there are differences in performance, *i.e.*, they do not reveal the *source* of the latency and scalability overheads.

This section presents the results of whitebox profiling that illustrate why the two ORBs performed differently. We analyze the `Quantify` results on sources of latency and scalability overhead in the two ORBs to help explain the variation reported in Section 4.3. The performance results reported in this section motivated the latency and scalability optimizations applied to our TAO ORB in Chapter 6.

Figures 4.24 and 4.25 show how Orbix and VisiBroker implement the generic SII request path shown in Figure 4.2. Percentages at the side of each figure indicate the contribution to the total processing time for a call to the `sendStructSeq` method, which was used to perform the operation for sending sequences of `BinStructs`.[4] The DII request path is similar to the SII path, except that clients create requests at run-time rather than using the stubs generated by the IDL compiler.

[4]The percentages in the figures do not add up to 100 since the overhead of the OS and network devices are omitted.

Orbix SII Request Flow Overhead

In Figure 4.24, the Orbix sender invokes the stub for the `ttcp_sequence::sendStructSeq` method. The request traverses through the `invoke` and the `send` routines of the CORBA `Request` class and ends up in the `OrbixChannel` class. At this point, it is handled by a specialized class, `OrbixTCPChannel`, which uses the TCP/IP protocol for communication.

On the receiver, the request travels through a series of dispatcher classes that locate the intended servant implementation and its associated IDL skeleton. Finally, the `ttcp_sequence_dispatch` class demultiplexes the incoming request to the appropriate servant and dispatches its `sendStructSeq` method with the demarshaled parameters.

On the sender, most of the overhead is attributed to the OS. The Orbix version uses the `write` system call which accounts for 73% of the processing time, due primarily to protocol processing in the SunOS 5.5 kernel. The remaining overhead can be attributed to marshaling and data copying, which accounts for ∼25% of the processing time. On the receiver, the demarshaling layer accounts for almost 72% of the overhead, due largely to the presentation layer conversion overhead incurred while demarshaling incoming parameters.

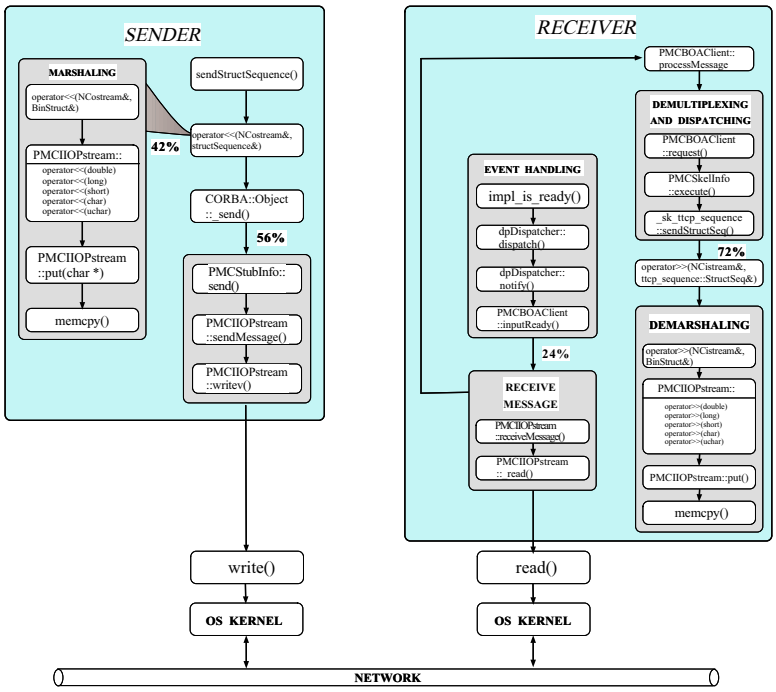


Figure 4.25: Request Path Through VisiBroker Sender and Receiver for SII

VisiBroker SII Request Flow Overhead

In Figure 4.25, the VisiBroker sender invokes the stub for the `sendStructSeq` method defined by the `ttcp_sequence` class. The request passes through the `send` methods of the

`CORBA::Object` and the `PMCStubInfo` classes. Finally, the request passes through the methods of the `PMCIIOPStream` class. This class implements the Internet Inter-ORB Protocol (IIOP), which specifies a standard communication protocol between servants on different heterogeneous hosts. The IIOP implementation writes to the underlying socket descriptor.

On the receiver, the IIOP implementation reads the packet using methods of the `PMCIIOPStream` class, which passes the request to the Object Adapter. The Object Adapter demultiplexes the incoming request by identifying the skeleton (`_sk_ttcp _seq ::skel`). The skeleton identifies the servant and makes an upcall to the `sendStructSeq` method of the `ttcp_sequence_i` implementation class.

On the sender, 56% of the overhead is attributed to the OS and networking level. The rest of the overhead stems from marshaling and data copying, which accounts for ∼42% of the processing time. On the receiver, the demarshaling and demultiplexing layer accounts for almost 72% of the processing time. VisiBroker spends most of its receiver processing demarshaling the parameters. In addition, the incoming parameters must travel through long chain of function calls (shown in Figure 4.25), which increases the overhead.

The request flow path traced by Orbix and VisiBroker is very similar. VisiBroker uses the standard IIOP common data representation (CDR) encoding as its native protocol for representing data types internally. In contrast Orbix uses a proprietary communication protocol based on ONC's XDR.

Servant Demultiplexing Overhead

To evaluate how the CORBA ORBs scale for endsystem servers, we instantiated 1, 100, 200, 300, 400, and 500 servants on the server. The following discussion analyzes the server-side overhead for demultiplexing client requests to servants. We analyze the performance of the `sendNoParams_1way` method for 500 servants on the server and 10 iterations. The `sendNoParams_1way` method is chosen so that the demultiplexing overhead can be analyzed without being affected by the demarshaling overhead involved with sending richly-typed data as shown in Sections 4.3.3 and 4.3.3.

Sources of Orbix demultiplexing overhead Table 4.1 depicts the latency and scalability impact of instantiating 500 servants and invoking 10 requests of the `sendNoParams_1way` method per servant using Orbix. `Quantify` analysis reveals that the performance of both the round robin and the request train case is similar. In both cases, the client spends most of its time performing network writes.

The server spends ∼22% of its time doing `strcmps` used for linearly searching the operation table to lookup the right operation, ∼16% of its time searching the hash table to locate the right servant and its skeleton, ∼10% of its time in `writes`, and ∼7% of its time in `select`. Orbix opens a new socket descriptor for every object reference obtained by the client. Hence, to demultiplex incoming requests, Orbix must use `select` to determine

Table 4.1: Analysis of Servant Demultiplexing Overhead for Orbix

Comm. Entity	Request Train	Analysis		
		Method Name	msec	%
Client	No	write	1,225	75.60
	Yes	write	1,229	73.88
Server	No	strcmp	3,010	21.97
		hashTable:: lookup	2,178	15.90
		write	1,392	10.16
		select	902	6.58
		hashTable:: hash	704.52	5.14
		Selecthandler:: processSockets	476	3.45
		read	346	2.55
	Yes	strcmp	2,979	21.06
		hashTable:: lookup	2,178	15.40
		write	1,504	10.63
		select	902	6.38
		hashTable:: hash	712	5.04
		Selecthandler:: processSockets	479	3.39
		read	372	2.63

which socket descriptor is ready for reading. The `writes` are used for flushing of buffers invoked by the underlying flow-control mechanism of the transport protocol.

Sources of VisiBroker demultiplexing overhead Table 4.2 depicts the affect on latency and scalability of instantiating 500 servants and invoking 10 iterations of the `sendNoParams_1way` method per servant using VisiBroker. The table reveals no significant difference between the round robin and request train case. The `Quantify` analysis for the VisiBroker version reveals that the server spends ∼15-20% of its time in network writes, ∼5% in `reads`, and ∼22% time demultiplexing requests.

VisiBroker's Object Adapter manages the internal tables ∼`NCTrans` and `NCOutTbl`. These tables use a hash-based table lookup strategy to demultiplex and dispatch incoming requests to their intended servants.

Comparison of Orbix and VisiBroker demultiplexing overhead

- VisiBroker opens one socket descriptor for all object references in the same server process. In contrast, Orbix opens a new socket descriptor for every object reference over networks (described in Section 4.3.1).
- VisiBroker uses a hashing-based scheme to demultiplex incoming requests to their servant. In contrast, although Orbix also uses hashing to identify the servant, a

Table 4.2: Analysis of Servant Demultiplexing Overhead for VisiBroker

Comm. Entity	Request Train	Analysis		
		Method Name	msec	%
Client	No	write	10,895	99.00
	Yes	write	10,992	99.00
Server	No	write	393	20.84
		~NCTransDict	138	7.31
		~NCClassInfoDict	138	7.31
		read	83	4.40
		NCOutTbl	73	3.84
		NCClassInfoDict	71	3.75
	Yes	write	275	15.32
		~NCTransDict	138	7.67
		~NCClassInfoDict	138	7.67
		read	83	4.61
		NCOutTbl	73	4.03
		NCClassInfoDict	71	3.93

different socket is used for each servant. Therefore, the OS kernel must search the list of open socket descriptors to identify which one is enabled for reading.

4.3.4 Additional Impediments to CORBA Scalability

In addition to servant demultiplexing overhead, both versions of CORBA used in our experiments possessed other impediments to scalability. In particular, neither worked correctly when clients invoked a large number of operations on a large number of servants accessed via object references.

We were not able to measure latency for more than ~1,000 servants since both ORBs crashed when we performed a large number of requests on ~1,000 servants. As discussed in Section 4.3.1, Orbix was unable to support more than ~1,000 servants since it opened a separate TCP connection and allocated a new socket for each servant in the server process. Moreover, even though VisiBroker supported ~1,000 servants, it could not support more than 80 requests per servant without crashing when the server had 1,000 servants *i.e.*, no more than a total of 80,000 requests could be handled by VisiBroker (this appears to be caused by a memory leak). Clearly, these limitations are not acceptable for mission-critical ORBS.

4.3.5 Summary of Performance Experiments

The following summarizes the results of our findings of conventional ORB latency and scalability over high-speed networks:

• **Sender-side overhead** Much of the sender-side overhead resides in OS calls that send requests. Removing this overhead requires the use of optimal buffer manager and tuning different parameters (such as socket queue lengths and flow control strategies) of the underlying transport protocol.

• **Receiver-side overhead** Much of the receiver-side overhead occurs from inefficient demultiplexing and presentation layer conversions (particularly for passing richly-typed data like `structs`). Eliminating the demultiplexing overhead requires de-layered strategies and fast, flexible message demultiplexing [14, 24]. Eliminating the presentation layer overhead requires optimized stub generators [52, 13] for richly-typed data.

• **Demultiplexing overhead** The Orbix demultiplexing performs worse than VisiBroker demultiplexing since Orbix uses a linear search strategy based on `string` comparisons for operation demultiplexing. In addition, due to an open TCP connection for every object reference, Orbix must use the UNIX event demultiplexing call `select` to determine which socket descriptors are ready for reading.

• **Intra-ORB function calls** Conventional ORBs suffer from excessive intra-ORB function calls, as shown in Section 4.3.3. Minimizing intra-ORB function calls requires sophisticated compiler optimizations such as integrated layer processing [7].

• **Dynamic invocation overhead** DII performance drops as the size of requests increases. To minimize the dynamic invocation overhead, ORBs should reuse DII requests and minimize the marshaling and data copying required to populate the requests with their parameters.

Chapter 5

Optimizing the Performance of the IIOP CDR Marshaling Engine

5.1 Introduction

References [20, 21, 23] show that the performance of conventional CORBA middleware implementations is relatively poor compared to lower-level implementations using sockets and C/C++ since the ORBs incur a significant amount of data copying, marshaling, demarshaling, and demultiplexing overhead. These results, however, focused entirely on the communication performance between *homogeneous* ORBs. They do not measure the runtime costs of interoperability between heterogeneous ORBs. In addition, earlier work on measuring CORBA performance did not present the results of optimizations to reduce key sources of ORB overhead.

In this chapter, we measure the performance of SunSoft IIOP using a CORBA/ATM testbed environment similar to [20, 21, 23]. SunSoft IIOP is a freely available implementation of the IIOP protocol. We measure the performance of SunSoft IIOP and precisely pinpoint its performance overheads. In addition, we describe the results of systematically applying seven **principle-driven optimizations** [75] that substantially improve the performance of SunSoft IIOP. These optimizations include: *optimizing for the common case; eliminating gratuitous waste; replacing general purpose methods with specialized, efficient ones; precomputing values, if possible; storing redundant state to speed up expensive operations; passing information between layers; and optimizing for the processor cache.*[1]

The results of applying these optimization principles to SunSoft IIOP improved its performance 1.9 times for `doubles`, 3.3 times for `longs`, 4 times for `shorts`, 5 times for `chars/octets`, and 6.7 times for richly-typed `structs` over ATM networks. Our optimized

[1]The optimizations and results described in this chapter have appeared in the IEEE Hawaii International Conference on System Sciences, 97.

version of SunSoft IIOP is now comparable to existing commercial ORBs [20, 21, 23] using the static invocation interface (SII) and around 2 to 4.5 times (depending on the data type) faster than commercial ORBs using the dynamic skeleton interface (DSI).

The optimizations and the resulting speedups reported in this chapter are essential for CORBA to be adopted as the standard for implementing high-bandwidth, low-latency distributed applications. The protocol optimizations described in this chapter are based on a set of principles that have been used to improve the performance of communication protocols described in Chapter 8. These principles can be applied to improve the performance of other ORB implementations and distributed object computing middleware.

Improving IIOP performance is not our only requirement, however, since optimizations must not come at the expense of interoperability. Therefore, we illustrate that our optimized implementation of SunSoft IIOP interoperates seamlessly with Visigenic's *VisiBroker for C++* ORB.

5.2 Overview of GIOP and SunSoft's IIOP Architecture

5.2.1 Overview of CORBA GIOP and IIOP

The CORBA General Inter-ORB Protocol (GIOP) defines an interoperability protocol between potentially heterogeneous ORBs. The GIOP protocol provides an abstract protocol specification that can be mapped onto conventional connection-oriented transport protocols. An ORB is GIOP-compatible if it can send and receive all valid GIOP messages.

The GIOP specification consists of the following elements:

A Common Data Representation (CDR) definition: The GIOP specification defines the CDR transfer syntax. CDR maps OMG IDL types from the native host format into a low-level *bi-canonical* representation, which supports both little-endian and big-endian formats. CDR encoded messages are used to transmit CORBA requests and server responses across a network. All OMG IDL data types are marshaled using the CDR syntax into an *encapsulation*, which is an octet stream that holds marshaled data.

GIOP message formats: The GIOP specification defines seven types of messages that send requests, receive replies, locate objects, and manage communication channels.

GIOP transport assumptions: The GIOP specification describes the type of transport protocols that can carry GIOP messages. In addition, the GIOP specification defines a connection management protocol and a set of constraints for message ordering.

The most common concrete mapping of GIOP onto the TCP/IP transport protocol is known as the Internet Inter-ORB Protocol (IIOP). The GIOP and IIOP specifications are described further in [51] and Appendix B.1.

5.2.2 Overview of the SunSoft IIOP Protocol Engine

SunSoft IIOP is a protocol engine that implements IIOP version 1.0. The key features and architecture of SunSoft IIOP are outlined below.

CORBA Features Supported by SunSoft IIOP

The SunSoft IIOP protocol engine is written in C++ and provides the features of a CORBA ORB Core. It handles connection management, socket endpoint demultiplexing, concurrency control, and the IIOP protocol. It is not a complete ORB, however, since it lacks an IDL compiler, an interface repository, and a Portable Object Adapter (POA).

On the client-side, SunSoft IIOP provides a *static invocation interface* (SII) and a *dynamic invocation interface* (DII). The SII is used by the client-side stubs. The DII is used by clients that have no compile-time knowledge of the interface they are calling. Thus, the DII allows clients to create CORBA requests at run-time.

In SunSoft IIOP, requests are created and parameters marshaled using the `Request`, `NVList`, `NamedValue`, and `TypeCode` *pseudo-object* interfaces defined by CORBA. Pseudo-objects are entities that are neither CORBA primitive types nor constructed types. Operations on pseudo-object references cannot be invoked using the DII mechanism since the interface repository does not keep any information about them. In addition, pseudo-objects are typically *locality constrained*, *i.e.*, they cannot be transferred as parameters to methods of an IDL interface.

SunSoft IIOP supports dynamic skeletons via the dynamic skeleton interface (DSI). The DSI is used by applications and ORB bridges [51] that have no compile-time knowledge of the interfaces they implement. Thus, the DSI parses incoming requests, unmarshals their parameters, and demultiplexes requests to the appropriate servants.

Servers that use the SunSoft DSI mechanism must provide `TypeCode` information used to interpret incoming requests and demarshal the parameters. `TypeCode`s are CORBA pseudo-objects that describe the format and layout of primitive and constructed IDL data types in the incoming request stream. This information is used by SunSoft IIOP's interpretive marshaling engine for each data type as it is marshaled and transmitted over a network.

The Sunsoft IIOP Software Architecture

The components in SunSoft IIOP are shown in Figure 5.1. The `TypeCode` marshaling and demarshaling protocol engine is the primary component of SunSoft IIOP. SunSoft IIOP's protocol engine is an interpreter that encodes or decodes parameter data by identifying their `TypeCode`s at run-time using the `_kind` field of each `TypeCode` object.

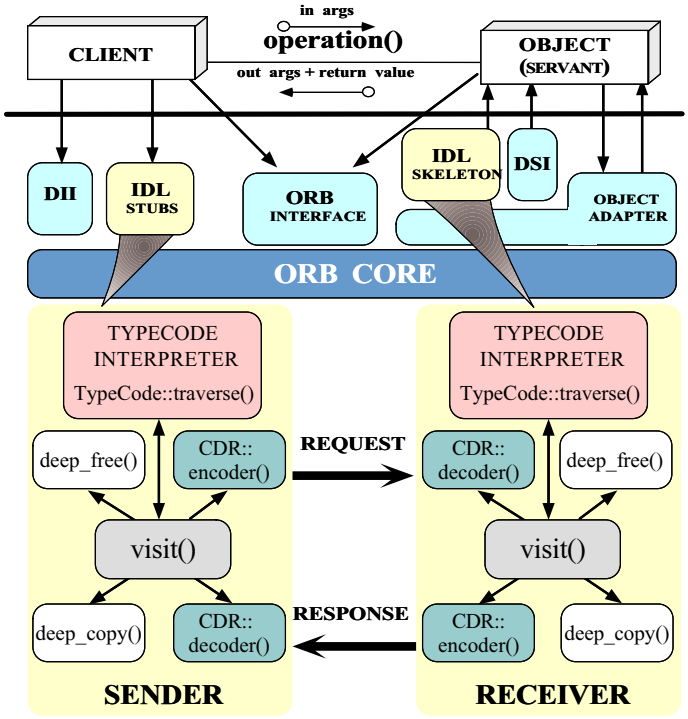


Figure 5.1: Components in the SunSoft IIOP Implementation

SunSoft IIOP uses an interpreter to reduce the space utilization of its protocol engine. Minimizing the memory footprint of a protocol engine is important for hand-held devices, such as PDAs. SunSoft IIOP's code size is less than 100 Kbytes on a real-time operating system like VxWorks. ORBs with small memory footprints are also useful for general-purpose operating systems since the protocol interpreter can be small enough to fit entirely within a processor cache.

Each component of the SunSoft IIOP software architecture is outlined below:

The TypeCode::traverse method: The SunSoft IIOP interpreter is implemented within the `traverse` method of the `TypeCode` class. All parameter marshaling and demarshaling is performed interpretively by traversing the data structure according to the layout of the `TypeCode/Request` tuple passed to `traverse`. This method is passed a pointer to a `visit` method (described below), which interprets CORBA requests based on their `TypeCode` layout. The request part of the tuple contains the data that was passed by an application on the client-side or received from the OS protocol stack on the server-side.

The visit method: The `TypeCode` interpreter invokes the `visit` method to marshal or demarshal the data associated with the `TypeCode` it is currently interpreting. The `visit` method is a pointer that contains the address of one of the four methods described below:

- *The* `CDR::encoder` *method* – The `encoder` method of the `CDR` class converts application data types from their native host representation into the CDR representation used to transmit CORBA requests over a network.
- *The* `CDR::decoder` *method* – The `decoder` method of the `CDR` class is the inverse of the `encoder` method. It converts request values from the incoming CDR stream into the native host representation.
- *The* `deep_copy` *method* – The `deep_copy` method is used by the SunSoft DII mechanism to allocate storage and marshal parameters into the CDR stream using the `TypeCode` interpreter.
- *The* `deep_free` *method* – The `deep_free` method is used by the SunSoft DSI server to free dynamically allocated memory after incoming data has been demarshaled and passed to a server application.

The utility methods: The following SunSoft IIOP methods perform various utility tasks:

- *The* `calc_nested_size_and_alignment` *method* – This method calculates the size and alignment of composite IDL data types like `struct`s or `union`s.
- *The* `struct_traverse` *method* – The `TypeCode` interpreter uses this method to traverse the fields in an IDL `struct` recursively.

Appendix B.3 examines the run-time behavior of SunSoft IIOP by tracing the path taken by requests used to transmit the `sequence` of `BinStruct`s. The IDL definition for BinStructs is shown in Appendix A.
The performance of SunSoft IIOP for these data types is examined in Section 5.3.1.

5.2.3 Overview of TAO

To avoid unnecessarily re-inventing existing ORB components, TAO is based on SunSoft IIOP's protocol engine. However, SunSoft IIOP has the following limitations:

Lack of complete ORB features: Although SunSoft IIOP provides an ORB Core, an IIOP protocol engine, and a DII and DSI implementation, it lacks an IDL compiler, an interface repository, and a Portable Object Adapter (POA). TAO implements many of these missing features and provides several new features such as real-time scheduling and dispatching mechanisms [69].

Lack of real-time features: SunSoft IIOP provides no support for real-time features. For instance, it uses a FIFO strategy for scheduling and dispatching client IIOP requests. FIFO strategies can yield unbounded priority inversions when lower priority requests block

the execution of higher priority requests [70]. TAO is designed carefully to prevent unbounded priority inversions. For instance, it provides a flexible scheduling service [69] that utilizes QoS information associated with the I/O subsystem [41] to schedule and dispatch requests according to their end-to-end priorities. To enable this, TAO extends SunSoft IIOP to support QoS parameters, as well as to allow separate IIOP connections to run within real-time threads with suitable priorities.

Lack of IIOP optimizations: As described in Section 5.3.1, SunSoft IIOP yielded relatively poor performance due to excessive marshaling/demarshaling overhead, data copying, and high-levels of function call overhead. Therefore, we applied *principle-based optimizations* [75] that improved its performance considerably [25].

The principles that directed our optimizations of SunSoft IIOP include: *(1) optimizing for the common case, (2) eliminating gratuitous waste, (3) replacing general-purpose methods with efficient special-purpose ones, (4) precomputing values, if possible, (5) storing redundant state to speed up expensive operations, (6) passing information between layers, and (7) optimizing for processor cache affinity.* As shown in Section 5.3.1, our optimizations yielded speedups of 2 to 6.7 for various types of OMG IDL data.

TAO alleviates the limitations with SunSoft IIOP described above to create a complete real-time ORB endsystem. An overview of TAO is provided in Section 1.4.

5.3 Performance Results and Benefits of Optimization Principles

5.3.1 Methodology

CORBA implementations like SunSoft IIOP are representative of complex communication software. Optimizing such software is hard, particularly since seemingly minor "mistakes," such as excessive data copying, dynamic allocation, or locking, can reduce performance significantly [20, 70]. Therefore, developing high-performance, predictable, and space-efficient ORBs requires an iterative, multi-step process. The first step involves measuring the performance of the system and pinpointing the sources of overhead. The second step involves a careful analysis of these sources of overhead and application of optimizations to remove them.

This section describes the optimizations we applied to SunSoft IIOP to improve its throughput performance. First, we show the performance of the original SunSoft IIOP for the IDL data types defined in Appendix B.2. Next, we use `Quantify` to illustrate the key sources of overhead in SunSoft IIOP. Finally, we describe the benefits applying optimization principles to improve the performance of SunSoft IIOP.

The optimizations described in this section are based on the core principles shown in Table 5.1 for implementing protocols efficiently.

Table 5.1: Summary of Principles for Efficient Protocol Implementations

Number	Principle
1	Optimizing for the common case
2	Eliminating gratuitous waste
3	Replacing inefficient general-purpose methods with efficient special-purpose ones
4	Precomputing values, if possible
5	Storing redundant state to speed up expensive operations
6	Passing information between layers
7	Optimizing for the processor cache

[75] describes a collection of optimization principles in detail and illustrates how they have been applied in existing protocol implementations, such as TCP/IP. This section focuses on the principles that we applied to systematically improve the performance of SunSoft IIOP. We focused on these principles since our experiments revealed they were the most strategic to improving SunSoft IIOP's performance. When describing our optimizations, we refer to these principles and explain how their use is justified.

The SunSoft IIOP optimizations were performed in the following three steps, corresponding to the principles from Table 5.1:

1. *Aggressive inlining to optimize for the common case* – which is discussed in Section 5.3.3;
2. *Precomputing, adding redundant state, passing information through layers, eliminating gratuitous waste, and specializing generic methods* – which is discussed in Section 5.3.4;
3. *Optimizing for the processor cache* – which is discussed in Section 5.3.5.

The order we applied the principles was based on the most significant sources of overhead identified empirically at each step and the principle(s) that most effectively reduced the overhead. For each step, we describe the principles and specific optimization techniques that were applied to reduce the overhead remaining from previous steps. After each step, we show the improved throughput measurements for selected data types. In addition, we compare the throughput obtained in the previous steps with that obtained in the current step.

The comparisons focus on data types that exhibited the widest range of performance, *i.e.*, `double` and `BinStruct`. As shown below, the first optimization step did not improve

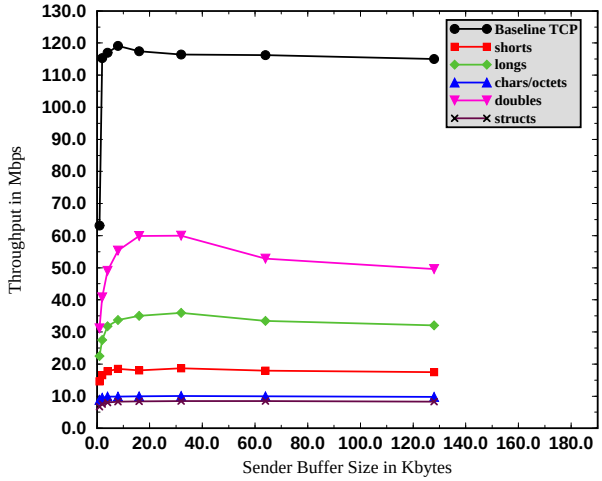
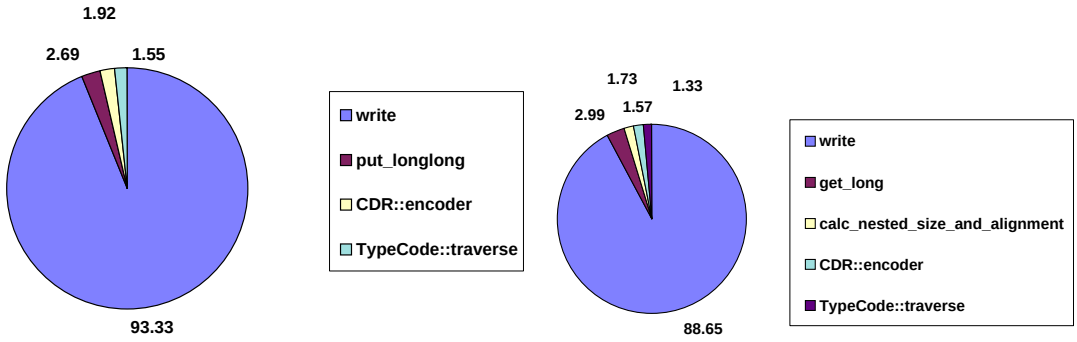


Figure 5.2: Throughput for the Original SunSoft IIOP Implementation



Analysis for `doubles` **Analysis for `BinStructs`**

Figure 5.3: Sender-side Overhead in the Original IIOP Implementation

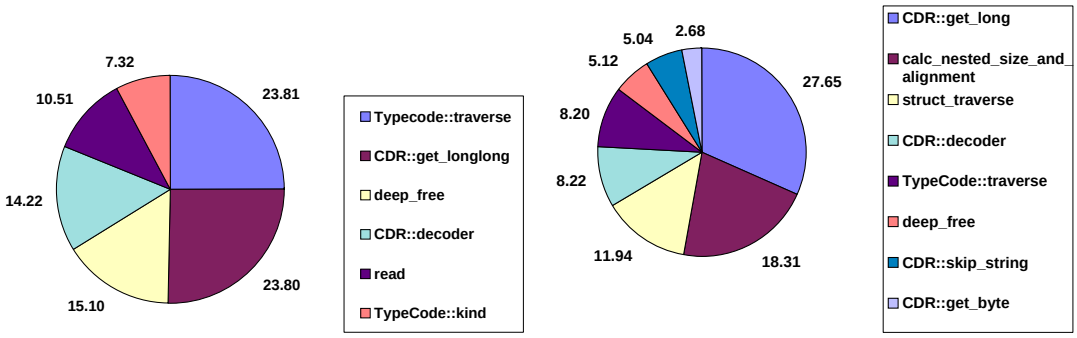
performance significantly. However, this step was necessary since it revealed the actual sources of overhead, which were then alleviated by the optimizations in subsequent steps.

5.3.2 Performance of the Original SunSoft IIOP Implementation

Sender-side performance:

Figure 5.2 illustrates the sender-side throughput obtained by sending 64 Mbytes of various data types for buffer sizes ranging from 1 Kbytes to 128 Kbytes (incremented by powers of two). The figure compares SunSoft IIOP with a hand-optimized baseline implementation that uses TCP/IP and sockets. These results indicate that different data types achieved substantially different levels of throughput.

The highest ORB throughput results from sending `doubles`, whereas `BinStructs` displayed the worst behavior. This variation in behavior stems from the marshaling and



Analysis for `doubles` **Analysis for `BinStructs`**

Figure 5.4: Receiver-side Overhead in the Original IIOP Implementation

demarshaling overhead for different data types. In addition, the original implementation of the interpretive marshaling/demarshaling engine in SunSoft IIOP incurred a large number of recursive method calls.

Figure 5.3 presents the results of using `Quantify` to send 64 Mbytes of `doubles` and `BinStructs` using a 128 Kbyte sender buffer. The results reveal that the sender spends ~90% of its run-time performing `write` system calls to the network. This overhead stems from the transport protocol flow control enforced by the receiving side, which cannot keep pace with the sender due to excessive presentation layer overhead. Table 5.2 provides detailed `Quantify` measurements indicating the time taken by dominant operations and the number of times they were invoked.

Receiver-side performance:

The `Quantify` analysis for the receiver-side is shown in Figure 5.4 and Table 5.3. The receiver-side results[2] for sending primitive data types indicate that most of the run-time costs are incurred by the following methods:

1. *The* `TypeCode` *interpreter* – *i.e.*, the `traverse` method in class `TypeCode`.
2. *The CDR methods that retrieve the value from the incoming data* – *e.g.,* `get_long` and `get_short`.
3. *The* `deep_free` *method* – which deallocates memory.
4. *The* `CDR::decoder` *method* – The receiver spends a significant amount of time traversing the `BinStruct TypeCode` (`struct_traverse`) and calculating the size and alignment of each member in the `struct`.

[2]Throughput measurements from the receiver-side were nearly identical to the sender measurements and are not presented here.

Table 5.2: Sender-side Overhead in the Original IIOP Implementation

Data Type	Analysis			
	Method Name	msec	Called	%
double	write	78,051	512	93.33
	put_longlong	2,250	8,388,608	2.69
	CDR::encoder	1,605	8,393,216	1.92
	TypeCode::traverse	1,300	1,024	1.55
long	write	134,141	512	92.92
	put_long	3,799	16,780,288	2.63
	CDR::encoder	3,303	16,781,824	2.29
	TypeCode::traverse	2,598	1,024	1.80
short	write	265,392	512	93.02
	put_short	7,593	33,554,432	2.66
	CDR::encoder	6,598	33,559,040	2.31
	TypeCode::traverse	5,195	1,024	1.82
octet	write	530,134	512	93.43
	CDR::encoder	15,986	67,113,472	2.82
	put_byte	10,391	67,118,080	1.83
	TypeCode::traverse	10,388	1,024	1.83
BinStruct	write	588,039	512	88.65
	get_long	19,846	44,053,504	2.99
	calc_nested_size...	11,499	14,683,648	1.73
	CDR::encoder	10,394	31,461,888	1.57
	TypeCode::traverse	8,803	4,195,328	1.33

As noted above, the receiver's run-time costs adversely affect the sender by increasing the time required to perform `write` system calls to the network due to flow control.

The remainder of this section describes the various optimization principles we applied to SunSoft IIOP, as well as the motivations and consequences of applying these optimizations. After applying the optimizations, we examine the new throughput measurements for sending different data types. In addition, we show how our optimizations affect the performance of the best case (`doubles`) and the worst case (`BinStruct`). Likewise, detailed profiling results from `Quantify` are provided only for the best and the worst cases.

Figures 5.3 and 5.4 illustrate the receiver is the principal performance bottleneck. Therefore, our initial set of optimizations are designed to improve receiver performance. Likewise, since the receiver is the bottleneck, we show only its `Quantify` profile measurements.

5.3.3 Optimization Step 1: Inlining to Optimize for the Common Case

Problem: high invocation overhead for small, frequently called methods:

This section describes an optimization to improve the performance of IIOP receivers. We applied Principle 1 from Table 5.1, which *optimizes for the common case*. Figure 5.4 illustrates that the appropriate `get` method of the CDR class must be invoked to retrieve the data from the incoming stream into a local copy. For instance, depending on the data type, methods like `CDR::get_long` or `CDR::get_longlong` are called between 10-80 million times

Table 5.3: Receiver-side Overhead in the Original IIOP Implementation

Data Type	Analysis			
	Method Name	msec	Called	%
double	TypeCode::traverse	2,598	1,539	23.81
	CDR::get_longlong	2,596	8,388,608	23.80
	deep_free	1,648	8,389,633	15.10
	CDR::decoder	1,551	8,395,797	14.22
	read	1,146	1,866	10.51
	TypeCode::kind	799	8,389,120	7.32
long	TypeCode::traverse	5,194	1,539	25.31
	CDR::get_long	4,596	16,783,379	22.40
	deep_free	3,296	16,778,241	16.06
	CDR::decoder	3,099	16,784,405	15.10
	read	1,682	2,574	8.20
	TypeCode::kind	1,598	16,777,728	7.79
short	TypeCode::traverse	10,387	1,539	27.22
	CDR::get_short	9,188	33,554,432	24.07
	deep_free	6,591	33,554,457	17.27
	CDR::decoder	6,195	33,561,621	16.23
	TypeCode::kind	3,196	33,554,944	8.37
octet	TypeCode::traverse	20,773	1,539	29.30
	CDR::decoder	13,984	67,116,053	19.73
	deep_free	13,182	67,109,889	18.59
	CDR::get_byte	10,787	67,118,113	15.22
	TypeCode::kind	6,391	67,109,376	9.02
BinStruct	CDR::get_long	35,091	83,921,427	27.65
	calc_nested_size...	23,001	29,370,880	18.31
	struct_traverse	15,154	4,194,304	11.94
	CDR::decoder	10,436	33,561,621	8.22
	TypeCode::traverse	10,401	6,292,995	8.20
	deep_free	6,492	14,681,089	5.12
	CDR::skip_string	6,394	33,566,720	5.04
	CDR::get_byte	3,399	21,153,313	2.68

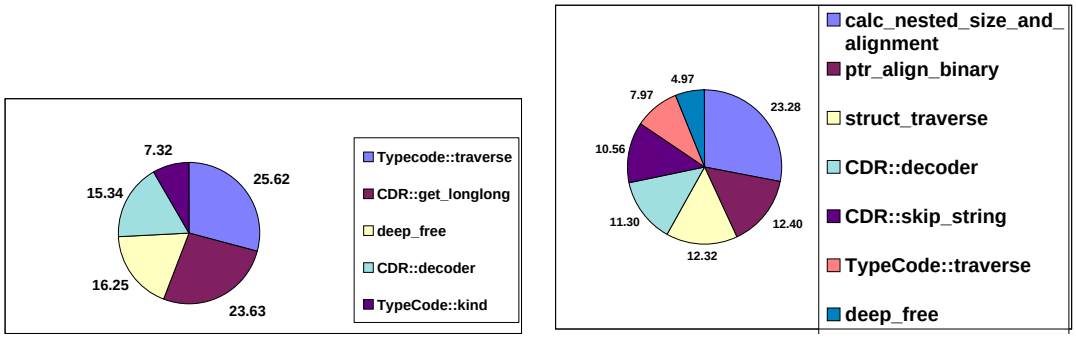
to decode the 64 Mbytes of data, as indicated in Table 5.3. Since these `get` methods are invoked quite frequently they are prime targets for our first optimization step.

Solution: inline method calls:

Our solution to reduce invocation overhead for small, frequently called methods was to inline these methods. Initially, we used the C++ `inline` language feature.

Problem: lack of C++ compiler support for aggressive inlining:

Our intermediate `Quantify` results after inlining, shown in Figure 5.5, reveal that supplying the `inline` keyword to the compiler does not always work since the compiler occasionally ignores this "hint." Likewise, inlining some methods may cause others to become "non-inlined." This occurs since the originally inlined operations (*e.g.,* `ptr _align _binary`) now invoke newly inlined operations thereby increasing their size. The C++ compiler then chooses to not inline the originally inlined operations.



Analysis for `doubles` **Analysis for `BinStructs`**

Figure 5.5: Receiver-side Overhead in the IIOP Implementation after Simple Inlining

Solution: replace inline methods with preprocessor macros:

To ensure inlining for all small, frequently called methods, we employ a more aggressive inlining strategy. This strategy *forcibly* inlined methods like `ptr _align _binary` (which aligns a pointer at the specified byte alignment) using preprocessor macros instead of as C++ `inline` methods.

In addition, the Sun C++ compiler did not inline certain methods, such as `skip _string` and `get _longlong`, due to their length. For instance, the code in method `get _longlong` swaps 16 bytes in a manually un-rolled loop if the arriving data was in a different byte order. This increases the size of the code, which caused the C++ compiler to ignore the `inline` keyword.

To workaround the compiler design, we defined a helper method that performs the byte swapping. This helper method is invoked only if byte swapping is necessary. This decreases the size of the code so that the compiler selected the method for inlining. For our experiments, this optimization was valid since we were transferring data between UltraSPARC machines with the same byte order.

Optimization results:

The throughput measurements after aggressive inlining are shown in Figure 5.6. Figures 5.7 and 5.8 illustrate the effect of aggressive inlining on the throughput of `doubles` and `BinStructs`. Figures 5.7 and 5.8 also compare the new results with the original results. After aggressive inlining, the new throughput results indicate only a marginal (*i.e.*, 4%) increase in performance. Figures 5.9 and 5.10, and Tables 5.4 and 5.5 show profiling measurements for the sender and receiver, respectively. As before, the analysis of overhead for the sender-side reveals that most run-time overhead stems from `write` calls to the network.

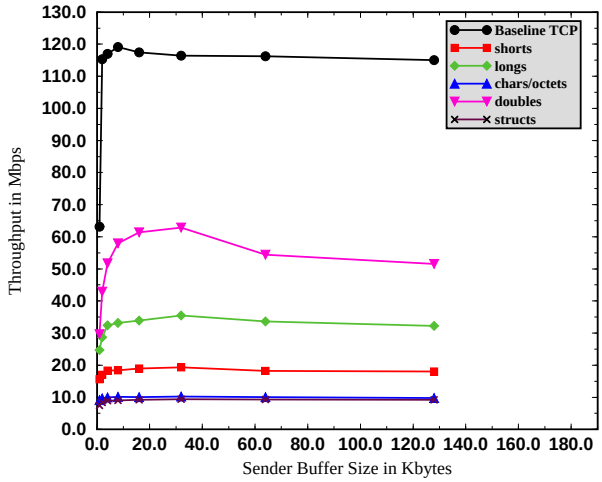


Figure 5.6: Throughput After Applying the First Optimization (inlining)

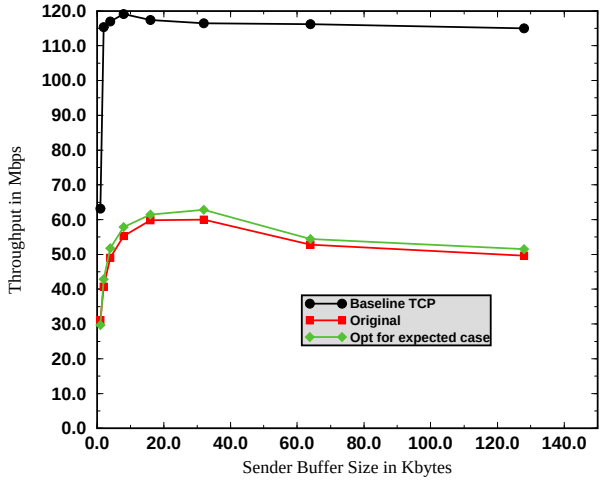


Figure 5.7: Throughput Comparison for Doubles After Applying the First Optimization (inlining)

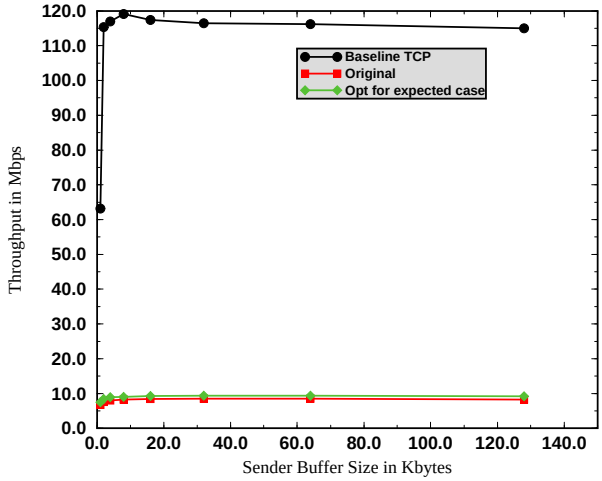


Figure 5.8: Throughput Comparison for Structs After Applying the First Optimization (inlining)

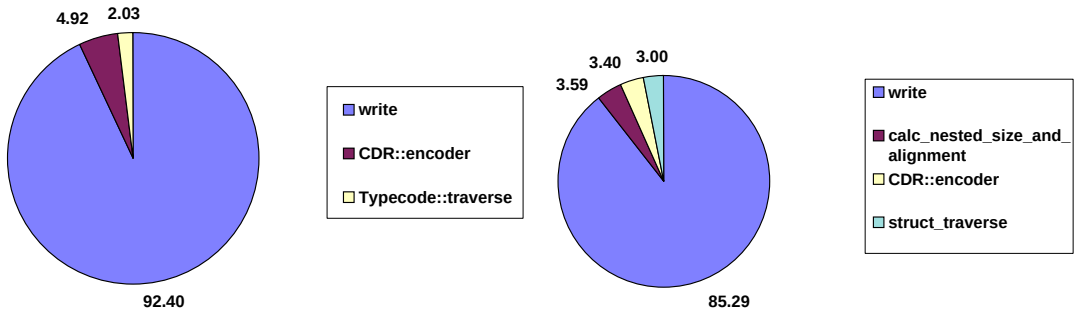
Table 5.4: Sender-side Overhead After Applying the First Optimization (inlining)

Data Type	Analysis			
	Method Name	msec	Called	%
double	write	59,260	512	92.40
	CDR::encoder	3,154	8,393,216	4.92
	TypeCode::traverse	1,300	1,024	2.03
BinStruct	write	436,694	512	85.29
	calc_nested_size...	14871	14,683,648	3.59
	CDR::encoder	14,101	31,461,888	3.40
	struct_traverse	12,425	2,097,152	3.00

The receiver-side `Quantify` profile output reveals that aggressive inlining does force operations to be inlined. However, this inlining increases the code size for other methods such as `struct_traverse`, `CDR::decoder`, and `calc _nested _size _and _alignment`, thereby increasing their run-time costs. As shown in Figures B.6 and B.7, these methods are called a large number of times (indicated in Figure 5.10 and Table 5.5).

Certain SunSoft IIOP methods such as `CDR::decoder` and `TypeCode::traverse` are large and general-purpose. Inlining the small methods described above causes further "code bloat" for these methods. Thus, when they call each other recursively a large number of times, very high method call overhead results. In addition, due to their large size, it is unlikely that code for both these methods can be resident in the processor cache at the same time, which explains why inlining does not result in significant performance improvement.

In summary, although our first optimization step did not improve performance dramatically, it helped to reveal the actual sources of overhead in the code, as explained in Section 5.3.4.



Analysis for `doubles` **Analysis for `BinStructs`**

Figure 5.9: Sender-side Overhead After Applying the First Optimization (aggressive inlining)

Table 5.5: Receiver-side Overhead After Applying the First Optimization (aggressive inlining)

Data Type	Analysis			
	Method Name	msec	Called	%
`double`	`CDR::decoder`	3,402	8,393,237	35.11
	`TypeCode::traverse`	2,598	1,539	26.82
	`deep_free`	1,648	8,389,633	17.01
	`TypeCode::kind`	799	8,389,120	8.25
`BinStruct`	`calc_nested_size...`	29,741	29,367,801	29.69
	`struct_traverse`	24,840	4,194,303	24.80
	`CDR::decoder`	14,641	33,554,437	14.62
	`TypeCode::traverse`	7,032	6,292,481	7.02
	`TypeCode::param_count`	4,020	4,195,846	4.01
	`deep_free`	6,492	14,681,089	4.97

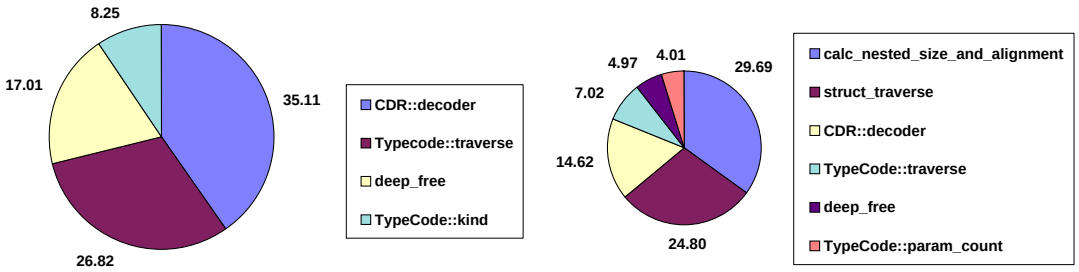
5.3.4 Optimization Step 2: Precomputing, Adding Redundant State, Passing Information Through Layers, Eliminating Gratuitous Waste, and Specializing Generic Methods

Problem: too many method calls:

The aggressive inlining optimization in Section 5.3.3 did not cause substantial improvement in performance due to processor cache effects as shown in this section and Section 5.3.5.

Table 5.5 reveals that for sending `structs`, the high cost methods are `calc _nested _size _and _alignment`, `CDR::decoder`, and `struct_traverse`. These methods are invoked a substantial number of times (29,367,801, 33,554,437, and 4,194,303 times, respectively) to process incoming requests.

To see why these methods were invoked so frequently, we analyzed the calls to `struct_traverse`. The `TypeCode` interpreter invoked `struct_traverse` 2,097,152 times for data transmissions of 64 Mbytes in `sequences` of 32-byte `Binstructs`. In addition, the `TypeCode` interpreter calculated the size of `BinStruct` (using the `calc _nested _size`



Analysis for `doubles` **Analysis for `BinStructs`**

Figure 5.10: Receiver-side Overhead After Applying the First Optimization (aggressive inlining)

`_and _alignment` function), which called `struct_traverse` internally for every `BinStruct`. This accounted for an additional 2,097,152 calls.

Although inlining did not improve performance substantially, it helped to answer a key performance question: *why were these high cost methods invoked so frequently*? Based on our detailed analysis of the SunSoft IIOP implementation (shown in Figure B.7 and in the explanation in Section 5.2.2), we recognized that to demarshal an incoming `sequence` of `BinStructs`, the receiver's `TypeCode` interpreter method `TypeCode::traverse` must traverse each of its members using the method `struct_traverse`. As each member is traversed, the `calc _nested _size _and _alignment` method determines the member's size and alignment requirements. Each call to the `calc _nested _size _and _alignment` method can invoke the `CDR::decoder` method, which may invoke the `traverse` method.

Close scrutiny of the CORBA request datapath shown in Figure B.7 reveals that the `struct_traverse` method calculates the size and alignment requirements every time it is invoked. As shown above, this yields a substantial number of method calls for large amounts of data.

Several solutions to remedy this problem are outlined below:

Solution 1: reduce gratuitous waste by precomputing values and storing additional state:

The first solution is based on the following two observations. First, for incoming `sequences`, the `TypeCode` of each element is constant. Second, each `BinStruct` in the IDL `sequence` has the same fixed size. These observations enabled us to pinpoint a key source of *gratuitous waste* (Principle 2 from Table 5.1). In this case, the gratuitous waste involves recalculating the size and alignment requirements of each element of the `sequence`. In our experiments, the methods `calc _nested _size _and _alignment` and `struct _traverse` are expensive. Therefore, it is crucial to optimize them.

To eliminate this gratuitous waste, we can *precompute* (Principle 4) the size and alignment requirements of each member and store them using *additional state* (Principle 5) to speed up expensive operations. We store this additional state as private data members of the SunSift's `TypeCode` class. Thus, the `TypeCode` for `BinStruct` will calculate the size and alignment *once* and store these in the private data members. Every time the interpreter wants to traverse `BinStruct`, it uses the `TypeCode` for `BinStruct` that has already precomputed its size and alignment. Note that our additional state does not affect the IIOP protocol since this state is stored locally in the `TypeCode` interpreter and is not passed across the network.

In general, all `struct` elements in a `sequence` may not have the same size. For instance, a `sequence` of `Any`s or `struct`s with `string` fields may have elements with variable sizes. In such cases, this optimization will not apply. For the `BinStruct` case described in this paper, however, a highly optimizing IDL compiler, such as Flick [13] could determine that all `sequence` elements have identical sizes. It could then generate stub and skeleton code that can eliminate gratuitous waste.

Solution 2: convert generic methods into special-purpose, efficient ones:

To further reduce method call overhead, and to decrease the potential for processor cache misses, we moved the `struct_traverse` logic for handling `struct`s into the `traverse` method. In addition, we introduced the `encoder`, `decoder`, `deep_copy`, and `deep_free` logic into the `traverse` method. This optimization illustrates an application of Principle 3 (*Convert generic methods into special-purpose, efficient ones*).

We chose to keep the `traverse` method generic, yet make it efficient since we want our demarshaling engine to remain in the cache. However, this scheme may not result in the best cache hit performance for machine architectures with small caches since the `traverse` method is excessively large. Section 5.3.5 describes optimizations we used to improve processor cache performance.

Problem: expensive no-ops for memory deallocation:

Figure 5.10 reveals that the overhead of the `deep_free` method remains significant for primitive data types. This method is similar to the `decoder` method that traverses the `TypeCode` and deallocates dynamic memory. For instance, the `deep_free` method has the same type signature as the `decoder` method. Therefore, it can use the recursive `traverse` method to navigate the data structure corresponding to the parameter and deallocate memory.

Careful analysis of the `deep_free` method indicates that memory must be freed for constructed data structures (such as IDL `sequences` and `structs`). In contrast, for `sequences` of primitive types, the `deep_free` method simply deallocates the buffer containing the `sequence`.

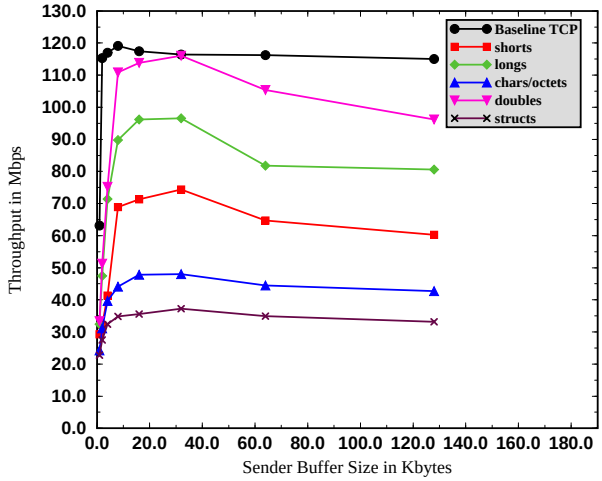


Figure 5.11: Throughput After Applying the Second Optimization (precomputation and eliminating waste)

Instead of limiting itself to this simple logic, however, the `deep_free` method uses `traverse` to find the element type that comprises the IDL `sequence`. Then, for the entire length of the `sequence`, it invokes the `deep_free` method with the element's `TypeCode`. The `deep_free` method immediately determines that this is a primitive type and returns. However, this traversal process is wasteful since it creates a large number of "no-op" method calls.

Solution: eliminate gratuitous waste:

To optimize the no-op memory deallocations, we changed the deletion strategy for `sequences` so that the element's `TypeCode` is checked *first*. If it is a primitive type, such as `double`, the traversal is not done and memory is deallocated directly.

Optimization results:

The throughput measurements recorded after incorporating these optimizations are shown in Figure 5.11. Figures 5.12 and 5.13 illustrate the benefits of the optimizations from step 2 by comparing the throughput obtained for `doubles` and `BinStructs`, respectively, with results from previous optimization steps.

Tables 5.6 and 5.7, and Figures 5.14 and 5.15 depict the profiling measurements for the sender and receiver, respectively. The receiver methods accounting for the most execution time for `doubles` include `traverse`, `decoder`, and `deep_free`. For `BinStructs`, the run-time costs of the `traverse` method in the receiver increases significantly compared to the previous optimization steps. This is due primarily to the inclusion of the

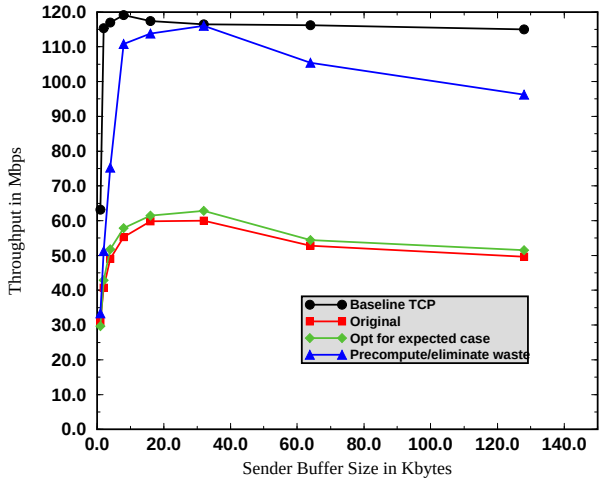


Figure 5.12: Throughput Comparison for Doubles After Applying the Second Optimization (precomputation and eliminating waste)

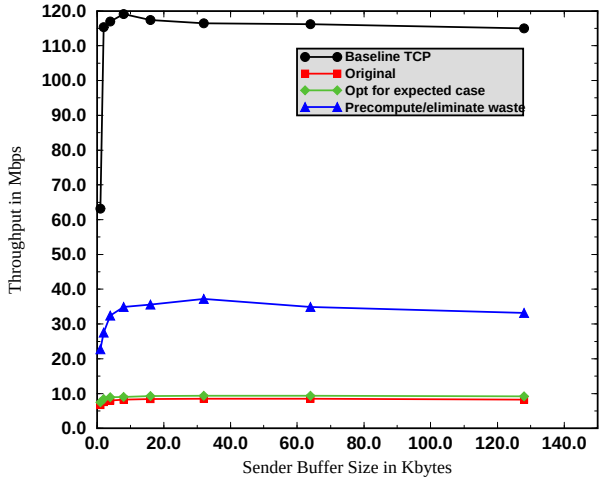
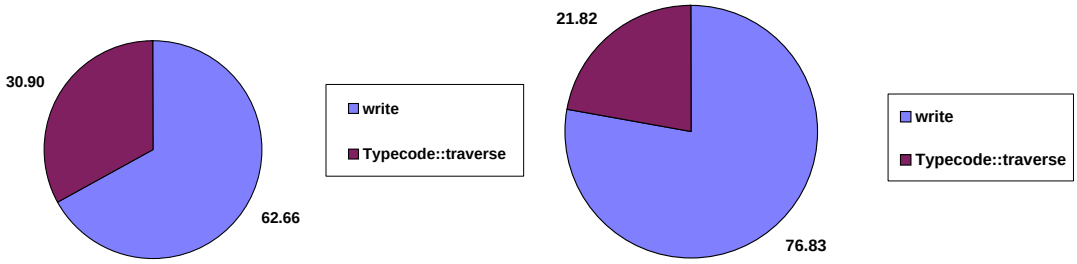


Figure 5.13: Throughput Comparison for Structs After Applying the Second Optimization (precomputation and eliminating waste)

Table 5.6: Sender-side Overhead After Applying the Second Optimization (getting rid of waste and precomputation)

Data Type	Analysis			
	Method Name	msec	Called	%
double	write	4,966	512	62.66
	TypeCode::traverse	2,449	1,024	30.90
BinStruct	write	61,641	512	76.83
	TypeCode::traverse	17,505	2,098,176	21.82



Analysis for `doubles` **Analysis for** `BinStructs`

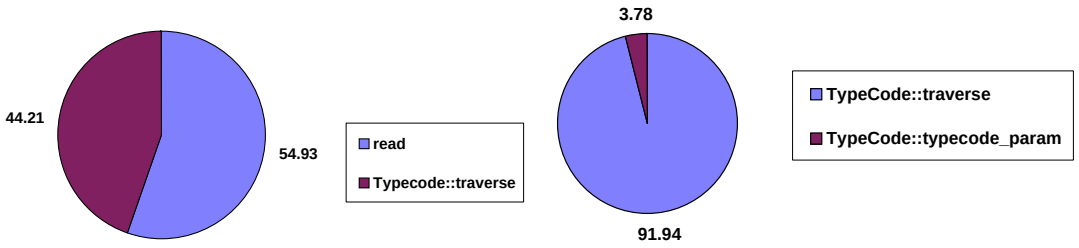
Figure 5.14: Sender-side Overhead After Applying the Second Optimization (getting rid of waste and precomputation)

`struct_traverse`, `encoder`, and `decoder` logic. Although the run-time costs of the interpreter increased, the overall performance improved since the number of calls to functions other than itself decreased. As a result, this design improved processor cache affinity, which yielded better performance. In addition, due to precomputation, `calc _nested _size _and _alignment` method need not be called repeatedly.

This result represents a substantial improvement and illustrates that the marshaling overhead of IIOP need not be a limiting factor in ORB performance.

Table 5.7: Receiver-side Overhead After Applying the Second Optimization (getting rid of waste and precomputation)

Data Type	Analysis			
	Method Name	msec	Called	%
double	read	3,413	4,665	54.93
	TypeCode::traverse	2,747	1,539	44.21
BinStruct	TypeCode::traverse	27,976	4,195,331	91.94
	TypeCode:: typecode_param	1,151	4,201,475	3.78



Analysis for `doubles` **Analysis for `BinStructs`**

Figure 5.15: Receiver-side Overhead After Applying the Second Optimization (getting rid of waste and precomputation)

5.3.5 Optimization Steps 3 and 4: Optimizing for Processor Caches

Processor caches are small, very fast memory used to significantly speed up operations [32]. To leverage off the advantages offered by the processor cache it is imperative that the footprint of the operations be small.

[47] describes several techniques to improve protocol latency. One of the primary areas to be considered for improving protocol performance is to improve the processor cache effectiveness. Hence, the optimizations described in this section are aimed at improving processor cache affinity, thereby improving performance.

Problem: very large, monolithic interpreter:

Section 5.3.4 describes optimizations based on precomputation, eliminating waste, and specializing generic methods. These optimizations yield an efficient, albeit excessively large, `TypeCode` interpreter. The efficiency stems from the fact that the monolithic structure results in low function call overhead. Recursive function calls are affordable since the processor cache is already loaded with the instructions for the same function. However, for machine architectures with smaller cache sizes, it may be desirable to have smaller functions.

Solution: split large functions into smaller ones and outlining:

This section describes optimizations we used to improve processor cache affinity for SunSoft IIOP. Our optimizations are based on two principles described below:

1. Splitting large, monolithic functions into small, modular functions: In our case, the `TypeCode` interpreter `traverse` method is the prime target for this optimization. As described earlier in Section 5.3.4, the logic for `encoder`, `decoder`, `struct_traverse`, `deep_free`, and `deep_copy` is merged into the interpreter, which increases its code size. The primary purpose of merging these methods is to reduce excessive function call overhead.

To improve processor cache affinity, however, it is desirable to have both smaller functions and minimal function call overhead. We accomplish this by splitting the interpreter into smaller functions that are targeted for specific tasks. These include functions that can encode or decode individual data types. This is in contrast to a generic encoder or decoder that can marshal any OMG IDL data type. Thus, to decode a `sequence`, the receiver uses the `decode_sequence` method of the `CDR` class and to decode a `struct`, it uses the `decode_struct` method.

It is possible to split the `decode_sequence` method further to support highly specialized methods (*e.g.,* `decode_sequence_long` to decode a `sequence` of `long`s). We are currently working on providing optimizations at this level of granularity. A smaller piece of code that demonstrates high locality of reference is ideally suited to take the benefits of processor caches.

This optimization principle is similar to Principle 3 from Table 5.1, which replaces general-purpose methods with efficient special-purpose ones. In the present case, however, the large, monolithic interpreter is replaced by special-purpose methods for encoding and decoding.

2. Using "outlining" to optimize for the frequently executed case: Outlining [47] is used to remove gaps that are introduced in the processor cache as a result of branch instructions arising from error handling code. Processor cache gaps are undesirable because they waste memory bandwidth and introduce useless no-op instructions in the cache.

The purpose of outlining is to move error handling code, which is rarely executed, to the end of the function. This enables frequently executed code to remain in contiguous memory locations, thereby preventing unnecessary jumps and hence increasing cache affinity by virtue of spatial locality.

Spatial locality is a property whereby data closely associated with currently referenced data are likely to be referenced soon. According to the 90-10 locality principle [32], a program executes 90% of its instructions in 10% of its code. If that 10% of the code demonstrates spatial locality, we can derive substantial cache affinity, which improves performance. Increased spatial locality can be achieved by using outlining, which reduces the number of gaps in the processor cache.

Outlining is a technique based on Principles 1 and 7 from Table 5.1, which optimize for the expected case and optimize for the processor cache, respectively.

The optimizations described in this section were applied in two steps. Since the `Quantify` analysis in the previous steps revealed the receiver as the source of overhead, we optimized the receiver side to gain greater processor cache effectiveness. However, the resulting `Quantify` analysis for `BinStructs` revealed that the sender-side which was `write`

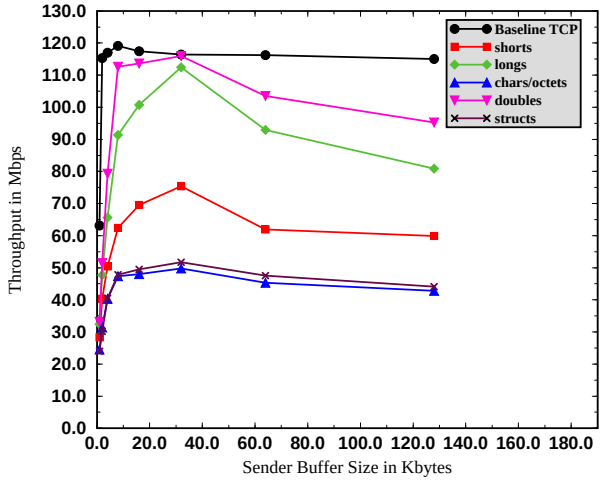


Figure 5.16: Throughput After Applying the Third Optimization (receiver-side processor cache optimization)

Table 5.8: Sender-side Overhead After Applying the Third Optimization (receiver-side processor cache optimization)

Data Type	Analysis			
	Method Name	msec	Called	%
`double`	`write`	3,385	512	53.21
	`TypeCode::traverse`	2,449	1,024	38.68
`BinStruct`	`TypeCode::traverse`	17,557	2,098,176	88.16
	`write`	1,270	512	6.37

bound after the optimizations in step 2, spends a substantial amount of time (88%) in the interpreter. Hence we applied the similar optimizations for the cache for the sender side. Specifically, the sender-side processor cache optimizations involve splitting the interpreter into smaller, specialized functions that can encode different OMG IDL data types.

Optimization Step 3: Receiver-side optimizations:

Figures 5.14 and 5.15 reveal that the sender is largely `write` bound. In contrast, the receiver spends most of its time in the interpreter. Therefore, it is appropriate to optimize the receiver-side code first to improve processor cache performance.

The throughput measurements recorded after incorporating these optimizations are shown in Figure 5.16. Figures 5.17 and 5.18 illustrate the benefits of the optimizations from step 3 by comparing the throughput obtained for `doubles` and `BinStructs`, respectively, with those from the previous optimization steps.

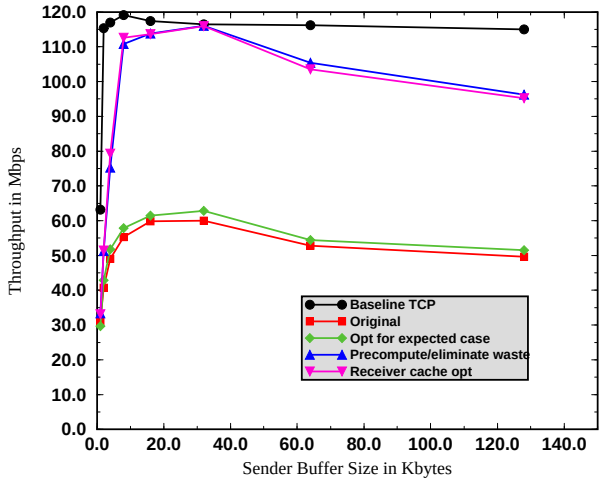


Figure 5.17: Throughput Comparison for Doubles After Applying the Third Optimization (receiver-side processor cache optimization)

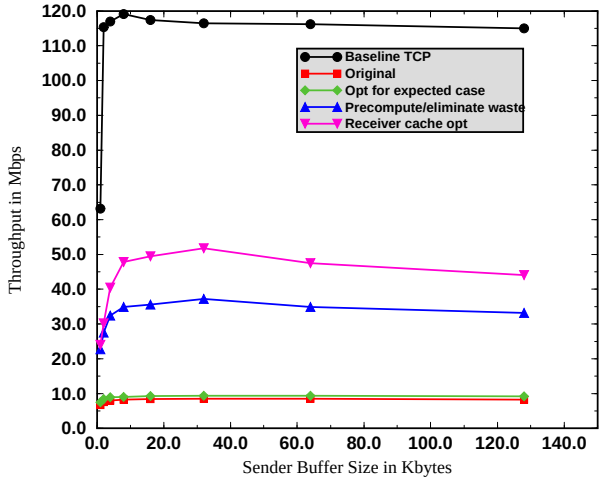
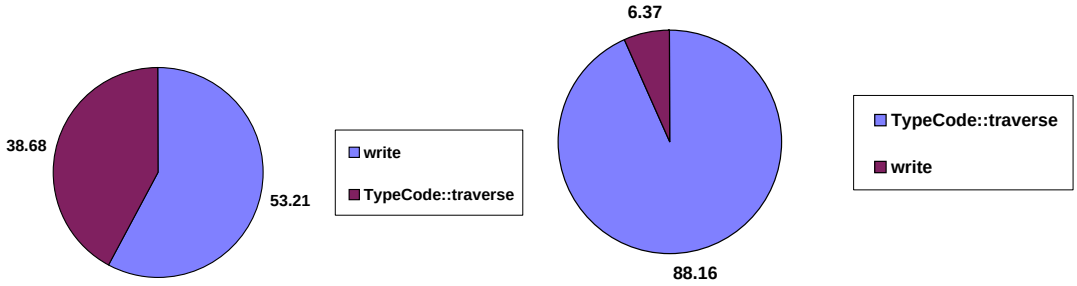


Figure 5.18: Throughput Comparison for Structs After Applying the Third Optimization (receiver-side processor cache optimization)

Table 5.9: Receiver-side Overhead After Applying the Third Optimization (receiver-side processor cache optimizations)

Data Type	Analysis			
	Method Name	msec	Called	%
double	read	3,392	5,688	53.21
	TypeCode::decode_seq	2,897	512	45.43
BinStruct	CDR::decode_seq	6,666	512	29.61
	CDR::decode_array	5,839	2,096,128	25.94
	deep_free_seq	4,359	512	19.36
	read	3,712	6,379	16.49
	typecode_param	1,150	4,200,963	5.11
	deep_free_array	712	2,097,152	3.16



Analysis for `doubles` **Analysis for `BinStructs`**

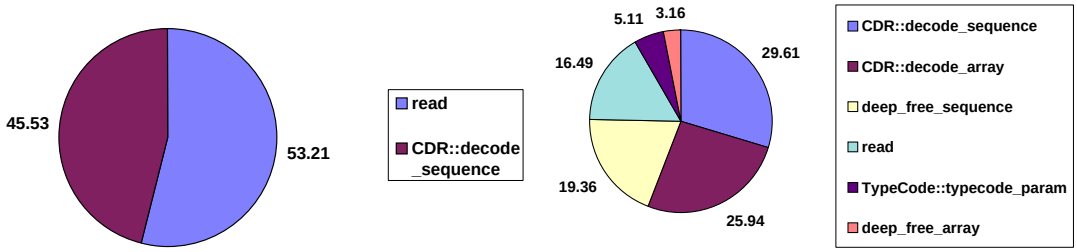
Figure 5.19: Sender-side Overhead After Applying the Third Optimization (receiver-side processor cache optimization)

Figures 5.19 and 5.20, and Tables 5.8 and 5.9 illustrate the remaining high cost sender-side and receiver-side methods, respectively. These indicate that for primitive types, the cost of writing to the network and reading from the network becomes the primary contributor to the run-time costs. These results represent a substantial improvement over the original results presented in Section 5.3.2 and illustrate that IIOP's marshalling overhead need not unduly limit ORB performance. For `BinStructs`, however, the sender-side (which was `write` bound after the optimizations in step 2) spends a substantial amount of time (88%) in the interpreter. The receiver spends most of its time in the specialized functions such as `decode_sequence` (30%), and `decode_array` (26%). The receiver-side analysis also reveals that the function call overhead has decreased significantly compared to step 2.

Optimization Step 4: Sender-side optimizations:

The sender-side processor cache optimizations involve splitting the interpreter into smaller, specialized functions that can encode different OMG IDL data types.

The throughput measurements recorded after incorporating these optimizations are shown in Figure 5.21. Figures 5.22 and 5.23 illustrate the benefits of the optimizations from



Analysis for `doubles` **Analysis for `BinStructs`**

Figure 5.20: Receiver-side Overhead After Applying the Third Optimization (receiver-side processor cache optimizations)

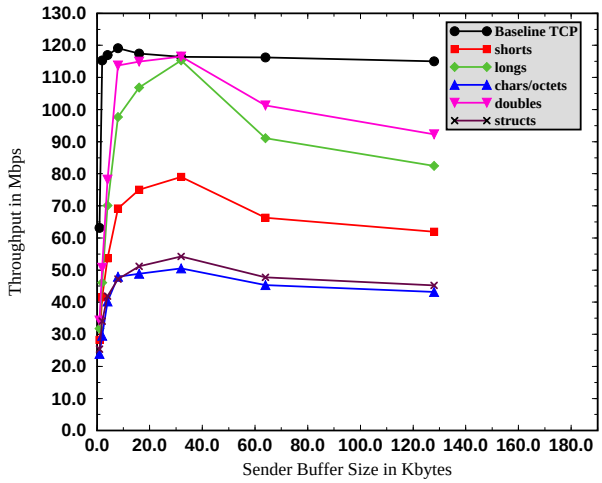


Figure 5.21: Throughput After Applying the Fourth Optimization (sender-side processor cache optimization)

step 4 by comparing the throughput obtained for `doubles` and `BinStructs`, respectively, with those from the previous optimization steps.

Figures 5.24 and 5.25, and Tables 5.10 and 5.11 illustrate the remaining high cost sender-side and receiver-side methods, respectively.

5.4 Maintaining CORBA Compliance and Interoperability

The optimizations presented in the previous sections are useful only to the extent that we maintain CORBA compliance and can interoperate with IIOP-based ORBs.

This subsection presents the throughput results obtained for running the same experiment with Visigenic's VisiBroker for C++ client and the SunSoft IIOP server. Figures 5.26 and 5.27 illustrate the throughput measurements for the original SunSoft IIOP server and our highly optimized implementation, respectively.

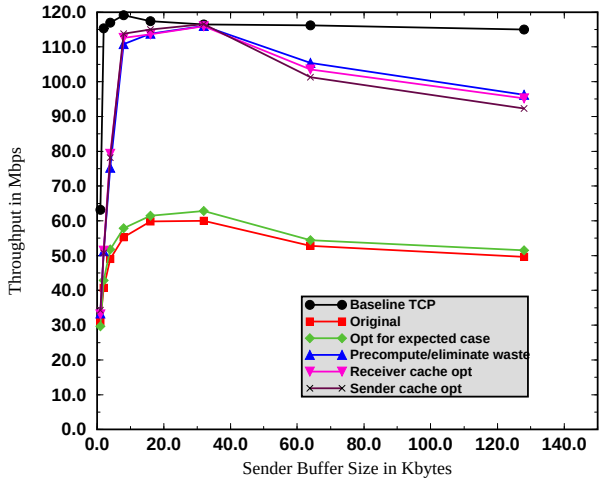


Figure 5.22: Throughput Comparison for Doubles After Applying the Fourth Optimization (sender-side processor cache optimization)

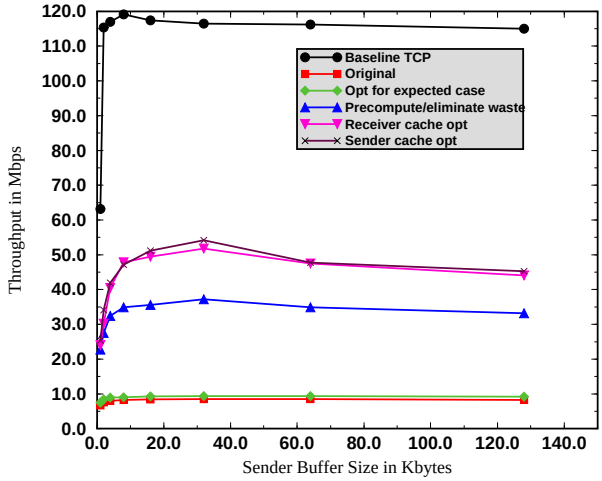
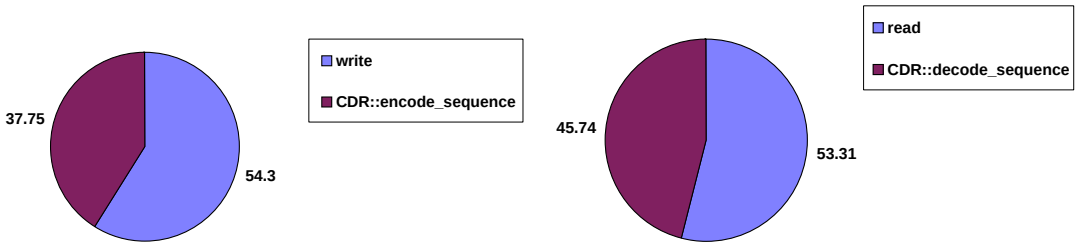


Figure 5.23: Throughput Comparison for Structs After Applying the Fourth Optimization (sender-side processor cache optimization)

Table 5.10: Sender-side Overhead After Applying the Fourth Optimization (sender-side processor cache optimization)

Data Type	Analysis			
	Method Name	msec	Called	%
double	write	3,522	512	54.30
	CDR::encode_seq	2,448	512	37.75
BinStruct	write	24,430	512	64.93
	CDR::encode_seq	6,357	512	16.90
	CDR::encode_arr	5,744	2,097,152	15.27



Analysis for `doubles` **Analysis for `BinStructs`**

Figure 5.24: Sender-side Overhead After Applying the Fourth Optimization (sender-side processor cache optimization)

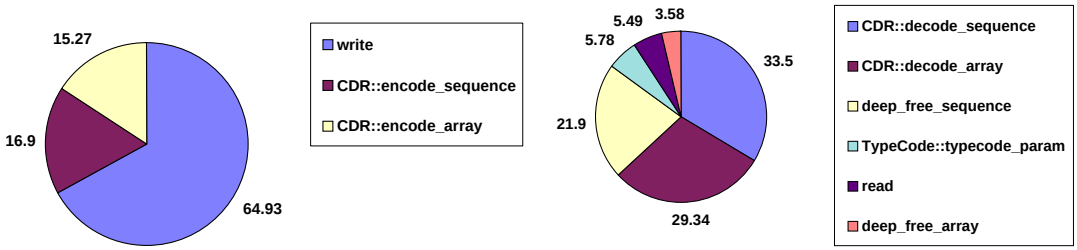
Table 5.11: Receiver-side Overhead After Applying the Fourth Optimization (sender-side processor cache optimizations)

Data Type	Analysis			
	Method Name	msec	Called	%
double	read	3,376	5,470	53.31
	TypeCode::decode_seq	2,897	512	45.74
BinStruct	CDR::decode_seq	6,666	512	33.50
	CDR::decode_array	5,839	2,096,128	29.34
	deep_free_seq	4,359	512	21.90
	typecode_param	1,150	4,200,963	5.78
	read	1,093	1,985	5.49
	deep_free_array	712	2,097,152	3.58¡

5.5 Summary and Research Contributions

This chapter illustrates the benefits of applying *measurement-driven, principle-based optimizations* that substantially improve the performance of CORBA Inter-ORB Protocol (IIOP) middleware. The seven principles that directed our optimizations include: *(1) optimizing for the common case, (2) eliminating gratuitous waste, (3) replacing general-purpose methods with efficient special-purpose ones, (4) precomputing values, if possible, (5) storing redundant state to speed up expensive operations, (6) passing information between layers, and (7) optimizing for processor cache affinity.*

Table 5.12 summarizes the problems encountered, the solutions proposed, and the optimization principle used to derive the solutions. The results of applying these optimization principles to SunSoft IIOP improved its performance 1.9 times for `doubles`, 3.3 times for `longs`, 4 times for `shorts`, 5 times for `chars/octets`, and 6.7 times for richly-typed `structs` over ATM networks. Our optimized implementation is now competitive with existing commercial ORBs [20, 23] using the static invocation interface (SII) and 2 to 4.5 times (depending on the data type) faster than commercial ORBs using the dynamic skeleton



Analysis for `doubles` **Analysis for `BinStructs`**

Figure 5.25: Receiver-side Overhead After Applying the Fourth Optimization (sender-side processor cache optimizations)

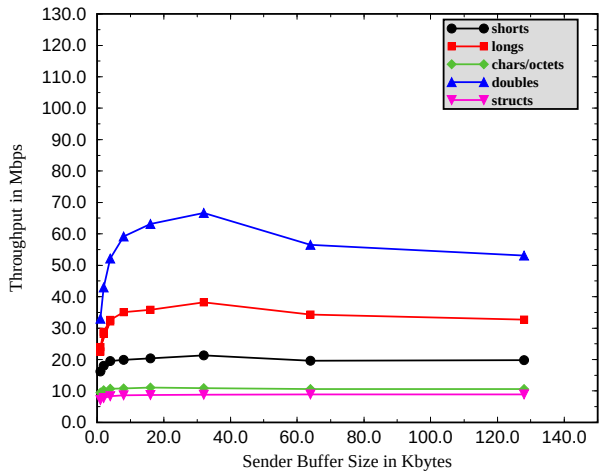


Figure 5.26: Throughput for VisiBroker Client and Original SunSoft IIOP server

interface (DSI) [21]. The results of our optimizations provide sufficient proof that performance of complex distributed systems software can be improved by a systematic application of principle-driven optimizations.

Moreover, we show that our optimized implementation of IIOP interoperates seamlessly with Visigenic's VisiBroker for C++ ORB which is a commercially available ORB.

We have integrated the optimized SunSoft IIOP implementation with our real-time ORB called TAO [67].[3]

[3]The TAO ORB is freely available at URL `www.cs.wustl.edu/~schmidt/TAO.html`.

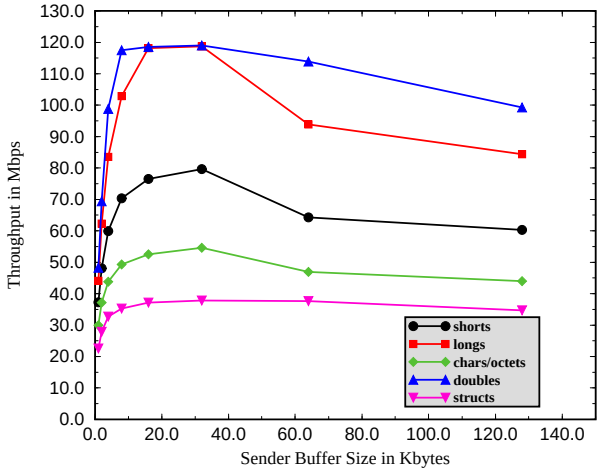


Figure 5.27: Throughput for VisiBroker Client and TAO's Optimized SunSoft IIOP Server

Table 5.12: Optimization Principles Applied in TAO

Problem	Solution	Principle
High overhead of small, frequently called methods	C++ `inline` hints	Optimize for common case
Lack of support for aggressive inlining	C preprocessor macros	Optimize for common case
Too many method calls	Specialize `TypeCode` interpreter	Generic to specialized
Expensive no-ops for deep_free of scalar types	Insert a check and delete at top level	Eliminate waste
Repetitive size and alignment calculation of sequence elements	Precompute size and alignment info in extra state in `TypeCode`	Precompute and maintain extra state
Duplication of tasks between function calls	Use default parameters solution and pass info. when appropriate	Pass info. across layers
Cache miss penalty	Split large interpreter into specialized methods and outline	Optimize for cache

Chapter 6

Optimized Demultiplexing Strategies for Real-time CORBA

6.1 Overview of CORBA Demultiplexing

6.1.1 Conventional CORBA Demultiplexing Architectures

A CORBA request header contains the identity of its remote object implementation (which is called a *servant* by the CORBA specification [51]) and its intended remote operation. A servant is uniquely identified by an *object key* and an *operation name.* An object key is represented as an IDL `sequence`, which is a single dimensional dynamic array of bytes; an operation name is represented as a string.

The Object Adapter is the component in the CORBA architecture that associates a servant with the ORB, demultiplexes incoming requests to the servant, and dispatches the appropriate operation of that servant. While current CORBA implementations typically provide a single Object Adapter per ORB, recent ORBOS portability enhancements [51] define the Portable Object Adapter (POA) to support multiple Object Adapters per ORB.

The demultiplexing strategy used by an ORB can impact performance significantly. Conventional ORBs demultiplex client requests to the appropriate operation of the servant using the following steps shown in Figure 6.1.

• **Steps 1 and 2:** The OS protocol stack demultiplexes the incoming client request multiple times, *e.g.,* through the data link, network, and transport layers up to the user/kernel boundary and the ORB core;

• **Steps 3, 4, and 5:** The ORB core uses the addressing information in the client's object key to locate the appropriate Object Adapter, servant, and the skeleton of the target IDL operation;

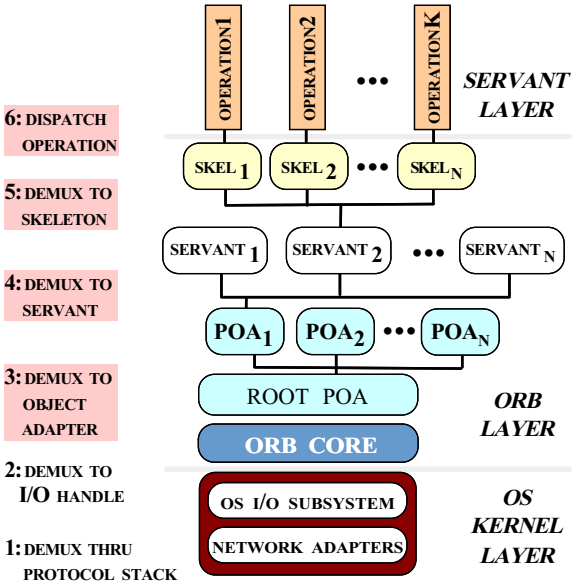


Figure 6.1: Layered CORBA Request Demultiplexing

• **Step 6:** The IDL skeleton locates the appropriate operation, demarshals the request buffer into operation parameters, and performs the operation upcall.

Demultiplexing client requests through all these layers is expensive, particularly when a large number of operations appear in an IDL interface and/or a large number of servants are managed by an ORB. Layered demultiplexing is particularly inappropriate for real-time applications [73] because it increases latency by increasing the number of times that internal tables must be searched while incoming client requests traverse various protocol processing layers. In addition, layered demultiplexing can cause priority inversions because servant-level QoS information is inaccessible to the lowest level device drivers and protocol stacks in the I/O subsystem of an ORB endsystem.

Conventional implementations of CORBA incur significant demultiplexing overhead. In particular, [20, 23] show that ~17% of the total server processing time is spent demultiplexing requests. Unless this overhead is reduced and demultiplexing is performed predictably, ORBs cannot provide real-time quality of service guarantees to applications.

6.1.2 Design of a Real-time Object Adapter for TAO

TAO is a high performance, real-time ORB developed at Washington University [69] in which the optimizations developed for this dissertation are incorporated. It runs on a range of OS platforms that support real-time features including VxWorks, Solaris 2.x, and Windows NT. TAO provides a highly optimized version of SunSoft's implementation of the CORBA Internet Inter-ORB Protocol (IIOP)[25].

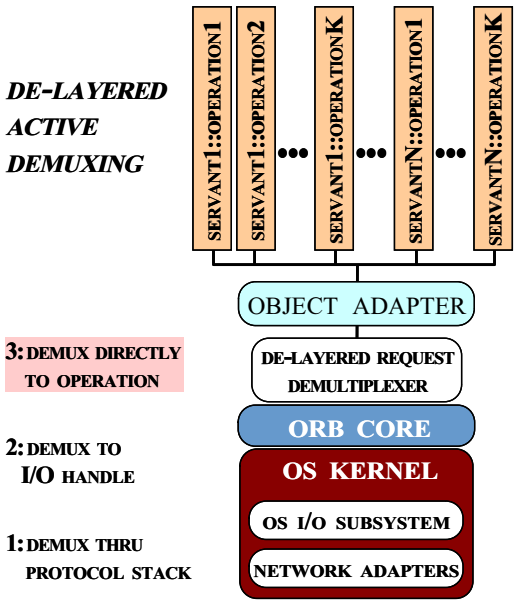


Figure 6.2: De-layered CORBA Request Demultiplexing

TAO's Object Adapter is designed to minimize overhead via de-layered demultiplexing [73] shown in Figure 6.2. This approach maps client requests directly to servant/operation tuples that perform application-level upcalls. The result is $O(1)$ performance for the average- and worst-cases.

Figure 6.3 illustrates the components in the CORBA architecture and the various demultiplexing strategies supported by TAO. TAO's flexible design allows different demultiplexing strategies [66] to be plugged into its Object Adapter. Section 6.3 presents the results of experiments using the following four demultiplexing strategies: (A) linear search, (B) perfect hashing, (C) dynamic hashing, and (D) de-layered active demultiplexing shown in Figure 6.3:

Linear search: The linear search demultiplexing strategy is a two-step layered demultiplexing strategy (shown in Figure 6.3(A)). In the first step, the Object Adapter uses the object key to linearly search through the active object map[1] to locate the right object and its skeleton (each entry in an active object map maintains a pointer to its associated skeleton). The skeleton maintains a table of operations defined by the IDL interface. In the second step, the Object Adapter uses the operation name to linearly search the operation table of the associated skeleton to locate the appropriate operation and invoke an upcall on it.

Linear search is known to be expensive and non-scalable. We include it in our experiments for two reasons: (1) to provide an upper bound on the worst-case performance,

[1]The active object map [51] associates object keys to servants maintained by an Object Adapter.

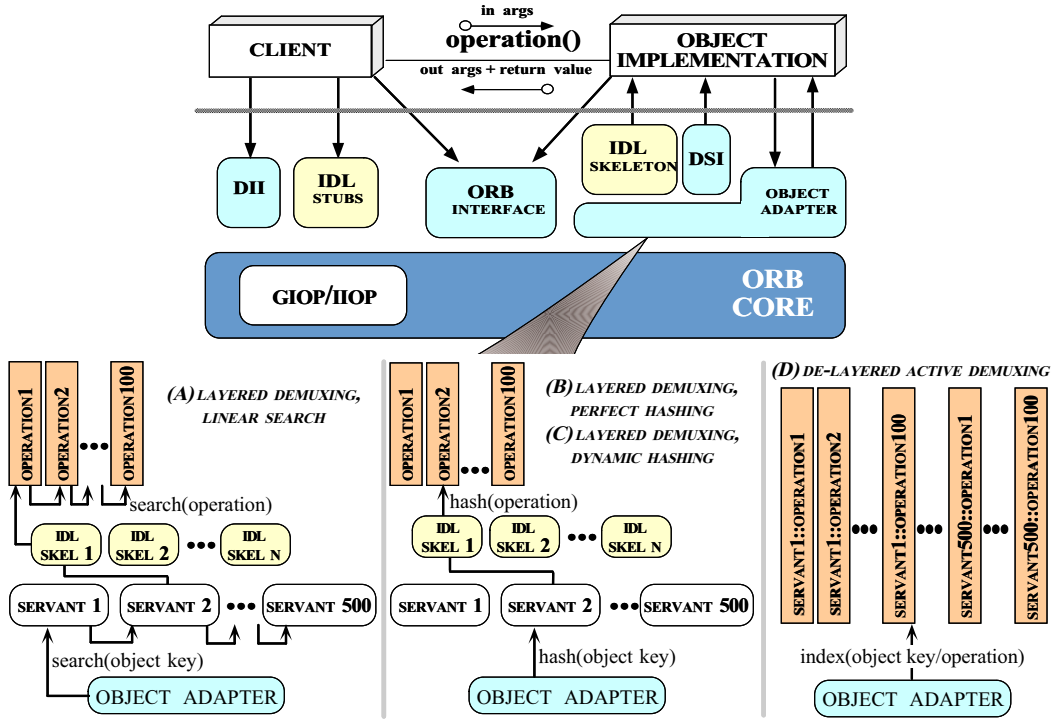


Figure 6.3: Alternative Demultiplexing Strategies in TAO

and (2) to contrast our optimizing demultiplexing strategies with strategies used in existing ORBs (such as Orbix) that use linear search for their operation demultiplexing.

Perfect hashing: The perfect hashing strategy is also a two-step layered demultiplexing strategy (shown in Figure 6.3(B)). In contrast to linear search, the perfect hashing strategy uses an automatically-generated perfect hashing function to locate the servant. A second perfect hashing function is then used to locate the operation. Both servant and operation lookup take constant time.

Perfect hashing is applicable when the keys to be hashed are known *a priori*. In many hard real-time systems (such as avionic control systems [31]), the objects and operations can be configured statically. In this scenario, it is possible to use perfect hashing to hash the object and operations. For our experiment, we used the GNU `gperf` [63] tool to generate perfect hash functions for object keys and operation names.

The following is a code fragment from the GNU `gperf` generated hash function for 500 object keys used in our experiments:

```
class Object_Hash
{
  // ...
  static u_int hash (const char *str, int len);
};

u_int
Object_Hash::hash (register const char *str,
```

```
                          register int len)
{
  static const u_short asso_values[] =
    {
     // all values not shown here
     1032, 1032, 1032, 1032, 1032, 1032, 1032,
      100,  105,  130,   20,  100,  395,  435,
      505,  330,  475,   45,  365,  180,  390,
      440,  160,  125, 1032, 1032, 1032, 1032,
    };
  return len + asso_values[str[len - 1]]
             + asso_values[str[0]];
}
```

The code above works as follows: upon receiving a client request, the Object Adapter retrieves the object key. It uses the object key to obtain a handle to the active object map by using the perfect `hash` function shown above. The `hash` function uses an automatically generated active object map (`asso_values`) to return a unique hash value for each object key.

Dynamic hashing: The dynamic hashing strategy is also a two-step layered demultiplexing strategy (shown in Figure 6.3(C)). In contrast to perfect hashing, which has $O(1)$ worst-case behavior and low constant overhead, dynamic hashing has higher overhead and $O(n^2)$ worse-case behavior. In particular, two or more keys may dynamically hash to the same bucket. These collisions are resolved using linear search, which can yield poor worse-case performance. The primary benefit of dynamic hashing is that it can be used when the object keys are not known *a priori*. In order to minimize collisions, the object and the operation hash tables contained twice as many array elements as the number of servants and operations, respectively.

De-layered active demultiplexing: The fourth demultiplexing strategy is called *de-layered active demultiplexing* (shown in Figure 6.3(D)). In this strategy, the client includes a handle to the object in the active object map and the operation table in the CORBA request header. This handle is configured into the client when the target object reference is registered with a Naming service or Trading service. On the receiving side, the Object Adapter uses the handle supplied in the CORBA request header to locate the object and its associated operation in a single step.[2]

[2]Detailed description on the four demultiplexing strategies is available at URL `http://www.cs.wustl.edu/~schmidt/GLOBECOM-97.ps.gz`

6.2 Additional features of the CORBA/ATM Testbed and Experimental Methods

6.2.1 Parameter Settings

Our earlier studies [20, 23] of CORBA performance over ATM demonstrate the performance impact of parameters such as the number of servants on an endsystem (*e.g.,* a server), and interfaces with large number of methods. Therefore, our benchmarks systematically varied these parameters for each experiment as follows:

• **Number of servants:** Increasing the number of objects on the server increases the demultiplexing effort required to dispatch the incoming request to the appropriate object. To pinpoint this demultiplexing overhead and to evaluate the efficiency of different demultiplexing strategies, we benchmarked a range of objects (1, 100, 200, 300, 400, and 500) on the server.

• **Number of operations defined by the interface:** In addition to the number of objects, demutiplexing overhead increases with the number of operations defined in an interface. To measure this demultiplexing overhead, our experiments defined a range of operations (1, 10, and 100) in the IDL interface. Since our experiments measured the overhead of demultiplexing, these operations defined no parameters, thereby eliminating the overhead of presentation layer conversions.

6.2.2 Request Invocation Strategies

Our experiments used two different invocation strategies for invoking different operations on the server objects. The two invocation strategies are:

• **Random request invocation strategy:** In this case, the client makes a request on a randomly chosen object reference for a randomly chosen operation. This strategy is useful to test the efficiency of the hashing-based strategy. In addition, it measures the average performance of the linear search based strategy. The algorithm for randomly sending client requests is shown below.

```
for (int i = 0; i < NUM_OBJECTS; i++) {
  for (int j = 0; j < NUM_OPERATIONS; j++) {
     choose an object at random from
        the set [0, NUM_OBJECTS - 1];
     choose an operation at random from
        the set [0, NUM_OPERATIONS - 1];
     invoke the operation on that object;
  }
}
```

• **Worst-case request invocation:** In this case, we choose the last operation of the last object. This strategy elicits the worst-case performance of the linear search strategy. The algorithm for sending the worse-case client requests is shown below:

```
for (int i = 0; i < NUM_OBJECTS; i++) {
  for (int j = 0; j < NUM_OPERATIONS; j++) {
     invoke the last operation on the
       last object
  }
}
```

6.3 Demultiplexing Performance Results

This section presents our experimental results that measure the overhead of the four demultiplexing strategies described in Section 6.1.2. Section 6.3.1 presents the blackbox results of our experiments. Section 6.3.2 provides detailed whitebox analysis of the blackbox results. Section 6.3.3 evaluates the pros and cons of each demultiplexing strategy.

6.3.1 Performance Results for Demultiplexing Strategies

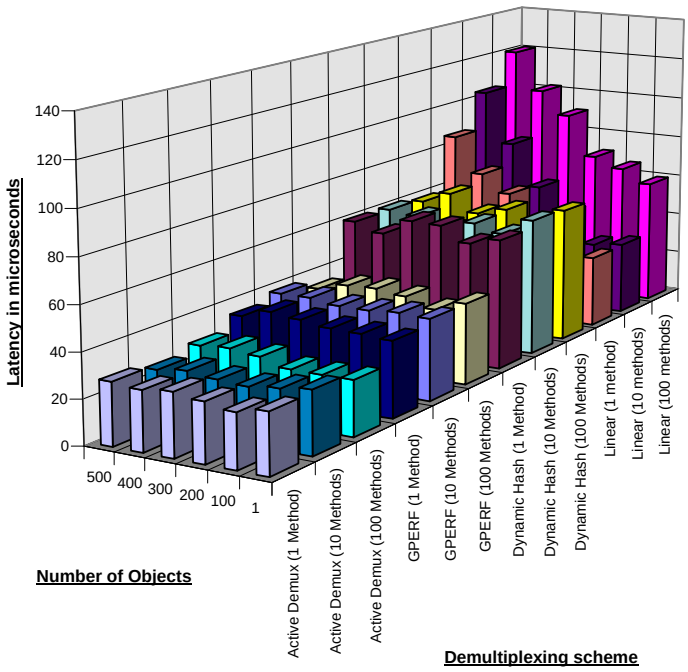


Figure 6.4: Demultiplexing Overhead for the Random Invocation Strategy

Figures 6.4 and 6.5 illustrate the performance of the four demultiplexing strategies for the random and worst-case invocation strategies, respectively. These figures reveal that in both cases, the de-layered active demultiplexing and perfect hash-based demultiplexing strategies substantially outperform the linear-search strategy and the dynamic hashing

strategy. Moreover, the worst-case performance overhead of the linear-search strategy for 500 objects and 100 operations is ~1.87 times greater than random invocation, which illustrates the non-scalability of linear search as a demultiplexing strategy.

In addition, the figures reveal that both the active demultiplexing and perfect hash-based demultiplexing perform quite efficiently and predictably regardless of the invocation strategies. The de-layered active demultiplexing strategy performs slightly better than the perfect hash-based strategy for both invocation strategies. Section 6.3.2 explains the reasons for these results.

6.3.2 Detailed Analysis of Demultiplexing Overhead

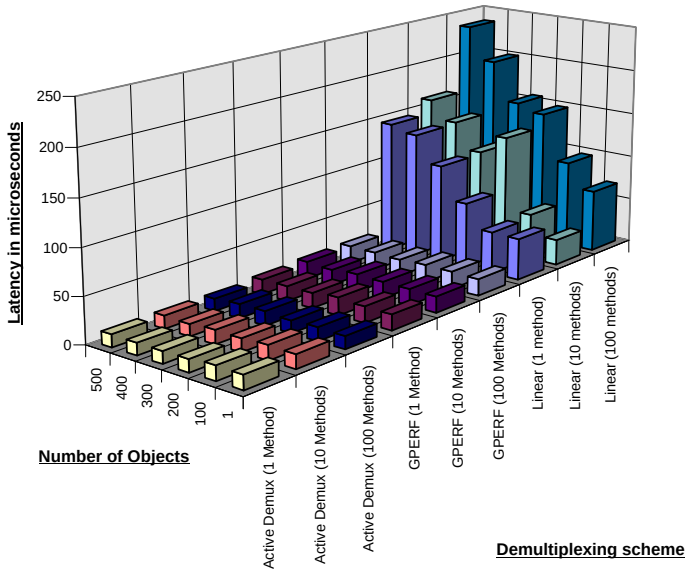


Figure 6.5: Demultiplexing Overhead for the Worst-case Invocation Strategy

This section presents the results of our whitebox profiling to illustrate the overhead of each demultiplexing strategy shown in Section 6.3.1. We explain the `Quantify` results from invoking 100 operations on 500 objects using the worst-case invocation strategy in Table 6.1.

The `dispatch` method used for our experiments is shown below:

```
int
CORBA_BOA::dispatch (CORBA_OctetSequence key,
                     CORBA_ServerRequest& req,
                     CORBA_Object_ptr obj)
{
   skeleton *skel;
   CORBA_Object_ptr obj;
   CORBA_String opname;

   // Find the object ptr corresponding
   //  to the key in the object table.
   // Use one of the 4 demux strategies
```

Table 6.1: Analysis of Demultiplexing Overhead

Strategy	Analysis			
	Name	Time in msec	Called	%
Active demux	OA::find	280.45	50,000	9.14
	skeleton	249.94	50,000	8.14
	Object::find	36.71	50,000	1.20
Perfect hash	OA::find	346.42	50,000	23.73
	skeleton	253.62	50,000	17.37
	Object::find	78.67	50,000	5.39
Dynamic hash	OA::find	670.56	50,000	34.13
	Object::find	265.39	50,000	13.51
	skeleton	253.62	50,000	12.91
Linear search	OA::find	12,122.73	50,000	72.46
	Object::find	3,617.48	50,000	21.62
	skeleton	249.65	50,000	1.49

```
   if (this->find (key, obj) {
     opname = req.opname ();
     // Now find the skeleton corresponding
     //  to the operation name.
     if (obj->find (opname, skel)) {
       // invoke the skeleton
     }
   }
}
```

Only the overhead incurred by the `CORBA::OA::dispatch` method and its descendants are shown in Table 6.1. The primary descendants of the `dispatch` method include the following:

- `OA::find` – which locates a servant corresponding to an object key in the object table maintained by the `OA`;
- `Object::find` – which locates a skeleton corresponding to the operation name in the operation table in the servant;
- `The skeleton` – which parses any arguments and makes the final upcall on the target servant.

Column `Name` identifies the name of the high cost descendants. The execution time is shown under column `Time in msec`. The `Called` column indicates the number of times the method was invoked and `%` indicates the percentage of the total execution time incurred by this method and its descendants.

Table 6.1 reveals that the linear-search based strategy spends a substantial portion of its time in the `OA::find` and `Object::find`. In turn, they perform string comparisons on the object keys and operation names, respectively. In contrast, both the hashing-based and active demultiplexing strategies incur no measurable overhead to locate the object and

the associated operation. The dynamic hashing scheme involves fewer string comparisons compared to the linear search strategy and hence it performs better.

The `OA::dispatch` method and its descendants account for ∼20% of the total execution time for the *De-layered Active Demultiplexing* strategy, ∼50% for the *Perfect hash* strategy, ∼63% for the *Dynamic hash* strategy, and ∼95% for the *Linear search* strategy. This explains why the de-layered active demultiplexing strategy outperforms the rest of the strategies.

6.3.3 Analysis of the Demultiplexing Strategies

The performance results and analysis presented in Sections 6.3.1 and 6.3.2 reveal that to provide low-latency and predictable real-time support, a CORBA Object Adapter must use demultiplexing strategies based on active demultiplexing or perfect hashing rather than strategies such as linear-search (which does not scale) and dynamic hashing (which has high overhead).

The perfect hashing strategy is primarily applicable when the object keys are known *a priori*. The number of operations are always known *a priori* since they are defined in an IDL interface. Thus, an IDL compiler can generate stubs and skeletons that use perfect hashing for operation lookup. However, objects implementing an interface can be created dynamically. In this case, the perfect hashing strategy cannot generally be used for object lookup.[3] In this situation, more dynamic forms of hashing can be used as long as they provide predictable collision resolution strategies. In many hard real-time environments it is possible to configure the system *a priori*. In this situation, however, perfect hashing-based demultiplexing can be used.

Our results show that de-layered active demultiplexing outperforms the other demultiplexing strategies. However, it requires the client to possess a handle for each object and its associated operations in the active object map and operation tables, respectively. Therefore, active demultiplexing requires either (1) preconfiguring the client with this knowledge or (2) defining a protocol for dynamically managing handles to add and remove objects correctly and securely.[4]

For hard real-time systems, this preconfiguration is typically feasible and beneficial. For this reason, we are using the perfect hashing demultiplexing strategy in the TAO ORB we are building for real-time avionics applications [69, 31].

[3] It is possible to add new objects at run-time using dynamic linking, though this is generally disparaged in hard real-time environments.

[4] We assume that the security implications of using active demultiplexing are addressed via the CORBA security service.

Chapter 7

Generating Efficient and Small Footprint Stubs and Skeletons

7.1 Introduction

Existing markets for hand-held devices, such as Personal Digital Assistants (PDAs), are being revolutionized by the advent of newer operating systems for hand-held devices such as Inferno, Windows CE 2.0, and Palm OS. Analysts estimate in excess of five million units of PDAs being sold by 1999. However, responses to recent surveys by users of PDAs indicate a dearth of software and applications. Many users require PDAs to possess high-speed, built-in data/cellular/fax modems that enable the PDA to be used as a cellular phone, a fax machine, and for sending and receiving emails, and browsing the internet.

Adding efficient and predictable communication capability to hand-held devices yields many research challenges related to mobile computing [17]. These challenges include dealing with low bandwidth, heterogeneity in the network connections, frequent changes and disruptions in the established connections due to migrating targets, maintaining consistency of data, and dealing with heterogeneous architectures to which these devices can be docked. In addition to the mobility issues, the restrictions on the physical size and power consumptions of these devices constrains the amount of storage capabilities possessed by these devices.

The challenge is to maintain a small footprint *i.e.,* smaller code size, for the ORB core, and the IDL compiler-generated stubs. Additionally, the performance of the generated stubs should not be compromised due to constraints on the footprint.The OMG has recently issued a Request for Proposals (RFPs) for a minimal-CORBA implementation geared towards embedded systems and other special systems that have constraints on the available resources such as memory.

This chapter compares compiled and interpretive marshaling used by CORBA IDL stubs and skeletons in terms of their performance and footprint. For this chapter, the stubs and skeletons using interpretive marshaling are generated by the TAO [69] IDL compiler. TAO's IDL compiler-generated stubs use a highly optimized interpretive scheme described in Chapter 5 to marshal and demarshal data types. The stubs and skeletons using compiled marshaling are hand-crafted. Our goal is to extend TAO IDL compiler's functionality to generate compiled marshaling stubs and skeletons.

Typically the code size for stubs and skeletons that use interpretive schemes is smaller in size compared to the compiled form. In addition, interpreted stubs/skeletons are slower than their counterparts. However, since TAO's interpretive marshaling engine is highly optimized, the performance of the stubs/skeletons is almost comparable to that of the compiled stubs. At the same time, the footprint of TAO IDL compiler (`tao_idl`) generated stubs/skeletons is significantly smaller than the compiled version. This quality makes it applicable to be used in PDAs and other embedded systems that have stringent restrictions on memory.

7.2 Optimizing TAO's IDL Compiler

Figure 7.1 illustrates the design of the TAO IDL Compiler. The TAO IDL compiler is based on the freely available SunSoft IDL compiler front-end.[1] The front-end of the compiler parses OMG IDL input and generates an in-memory abstract syntax tree (AST). We customized the back-end to process the AST and generate C++ source code that is optimized for the interpretive IIOP protocol engine described in Section 5.3.1.

7.2.1 The Design of TAO's IDL Compiler Front-end

TAO's IDL compiler front-end contains the following components adapted from the original SunSoft IDL compiler:

OMG IDL Parser: The parser comprises a `yacc` specification of the OMG IDL grammar. The action for each grammar rule invokes methods of the AST node classes to build the AST.

Abstract Syntax Tree Generator: Different nodes of the AST correspond to the different constructs of CORBA IDL. The front-end defines a base class called `AST_Decl` that maintains information common to all AST node types. Specialized AST node classes (such as `AST_Interface`) inherit from this base class.

In addition, the TAO IDL compiler defines a class called `UTL_Scope`, which maintains scoping information such as the nesting level and each component of the fully scoped name.

[1] The original SunSoft IDL compiler implementation is available at `ftp://ftp.omg.org/pub/OMG_IDL_CFE_1.3`.

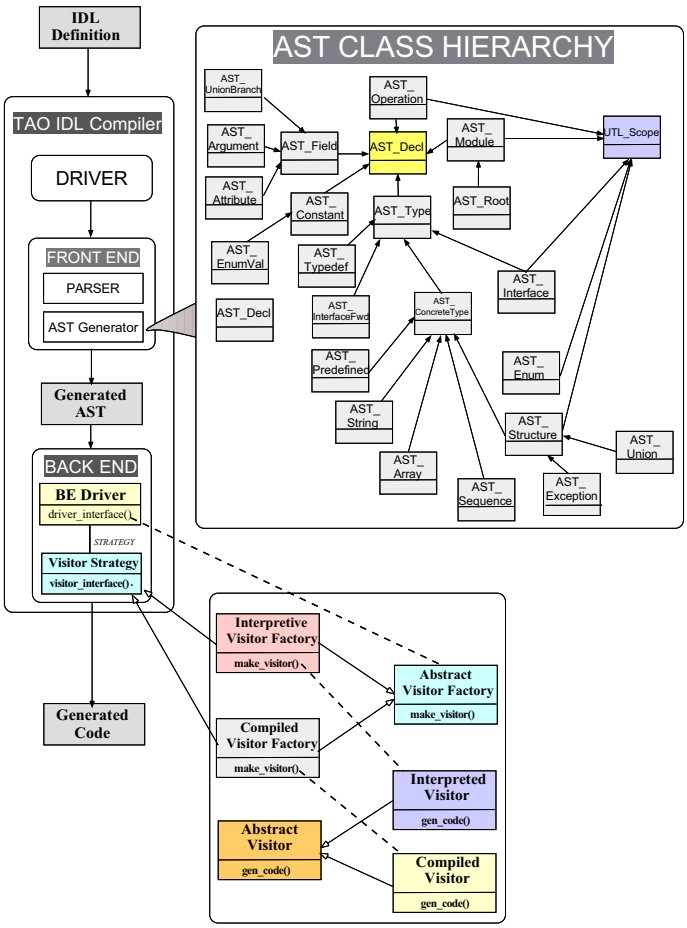


Figure 7.1: The TAO IDL Compiler

All AST nodes representing CORBA IDL constructs that can define scopes (such as `structs` and `interfaces`) also inherit from the `UTL_Scope` class.

Driver program: The driver program directs the parsing and AST generation process. It reads an input OMG IDL file and invokes the parser and the AST generator.

7.2.2 The Design of TAO's Back-end Code Generator

The original SunSoft IDL compiler front-end parses OMG IDL and generates the abstract syntax tree (AST). To create a complete CORBA IDL compiler for TAO, we developed a back-end for the OMG IDL to C++ mapping. TAO's IDL compiler back-end uses several design patterns [18] such as *Abstract Factory*, *Strategy*, and *Visitor*. As a consequence, TAO's back-end can be reconfigured to produce stubs/skeletons that use either compiled or interpretive marshaling.

The back-end of TAO's IDL compiler produces stubs and skeletons that integrate with TAO's highly optimized IIOP interpretive protocol engine described in Section 5.3.1.

We are currently developing a back-end to produce compiled stubs, though the compiled stubs/skeletons for this paper were hand-crafted.

The interpreted stubs and skeletons generated by TAO's IDL compiler are explained below. We use the `test_short` method from the `Param_Test` interface shown in Appendix C to explain the behavior of the stubs and skeletons.

Interpreted stubs

The stubs produced by TAO's IDL compiler use a table-driven technique to pass parameters to the underlying interpretive marshaling engine. The basic structure of a stub is outlined below:

1. Initialize table entries describing each parameter's type via its `TypeCode`, and its parameter passing mode.
2. Initialize a table describing the operation including its name, whether it is oneway or two-way, the number of parameters it takes, and a pointer to the table described in Step 1.
3. A variable for the return value, if any, is allocated.
4. A stub object is retrieved from the object reference on which this operation is invoked.
5. The `do_call` method is invoked on this stub object passing it the operation description table and the parameter values in the same order in which they were defined in the IDL definition.

The `do_call` method described above is the interface to TAO's interpretive protocol engine. It takes a variable number of parameters starting with the operation description table followed by the parameters of the operation. The stub for the `test_short` operation is shown in Appendix D.1.

The table-driven technique and the `do_call` interface were available in the original SunSoft IIOP implementation. However, since the SunSoft IIOP implementation did not have an IDL compiler each stub was hand-crafted. In contrast, the TAO IDL compiler automatically generates stubs that use this scheme.

Interpreted skeletons

The skeletons use a Dynamic Skeleton Interface (DSI) approach for marshaling parameters. The basic algorithm for a skeleton is described below and in Figure 7.2.

1. Create an `NVList`, which is a container class, to hold the parameters.

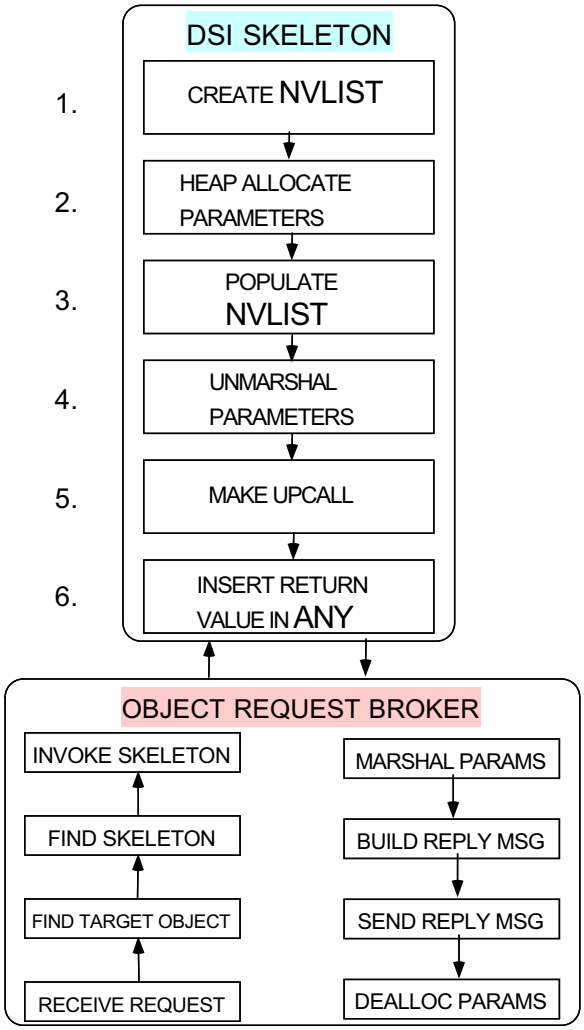


Figure 7.2: Unoptimized skeletons

2. Heap allocate all the `return`, `inout`, and `out` parameters since they qhave to be marshaled back into the outgoing stream. The `in` parameters can be allocated on the skeleton call stack.
3. Add each parameter value to the `NVList` using the operations provided by the DSI mechanism.
4. Use the DSI operation `arguments` to unmarshal incoming parameters.
5. Make an upcall on the target object passing it all the unmarshaled parameters.
6. Create a `CORBA::Any` to hold the return value, if any.
7. Return from the skeleton and let the ORB internally handle the task of marshaling the `return`, `inout`, and `out` parameters.

The skeleton for `test _short _skel` operation is shown in Appendix D.2.

Memory is allocated from the heap for `inout`, `out`, and `return` values. This heap allocation is essential because these parameters are marshaled into the outgoing IIOP Reply message after the call to the the skeleton has returned. As a result, it is not possible to allocate the parameters on the call stack of the skeleton. These heap allocated data structures are owned by the ORB and are freed using an interpretive scheme in the same way as they are marshaled interpretively.

As mentioned before, SunSoft IIOP does not provide an IDL compiler and these skeletons must be hand-crafted. TAO's IDL compiler produces these skeletons automatically.

Compiled stubs and skeletons

For this paper, the compiled stubs and skeletons are hand-crafted.[2] The basic structure of a compiled stub is shown below. The skeleton's algorithm is very similar to the stub:

1. Retrieve the stub object from the object reference.
2. Setup a CDR stream object into which the parameters will be marshaled.
3. The CDR stream object is initialized with the details of the receiving endpoint.
4. Insert each parameter into the stream in the same order in ehich they are defined in the IDL description.
5. Send the parameters and wait for return values.
6. Unmarshal all the return, inout, and out parameters and return.

The hand-crafted compiled stub and skeleton for `test_short` are shown in Appendix D.3.

The compiled stubs and skeletons use overloaded `operator<<` and `operator>>` to marshal data types to/from the underlying CDR (Common Data Representation) stream. TAO's ORB Core provides these operators for primitive types. However, for user-defined types, these must be provided by the IDL compiler. In this paper, we hand-crafted these overloaded operators for the different user-defined types we tested.

A significant difference between the compiled skeleton and the interpreted skeleton is that no unnecessary heap allocation is required in the compiled skeleton. This is due to the fact that a compiled skeleton has static knowledge of the types it is marshaling and unmarshaling. At the same time, all the unmarshaling and marshaling of the parameters occur in the scope of the skeleton. In contrast, in the interpretive skeletons using the DSI scheme, the marshaing of `return`, `inout`, and `out` parameters occur inside the ORB after the activation record of the skeleton has been destroyed.

[2]We are currently implementing a back-end for TAO IDL compiler that contains strategies for producing stubs and skeletons using compiled marshaling.

7.2.3 Techniques for Reducing Stub/Skeleton Code Size

As described in Section 7.2.2 and shown in Appendix D, the size of the interpreted stubs and skeletons is large. However, due to stringent constraints on the available memory in hand-held devices, it is imperative to maintain a small footprint of the stubs and skeletons as well as the ORB core.

Therefore, we devised a technique to reduce the code size. Our code-size reduction techniques for interpreted stubs/skeletons are guided by the following three optimization principles:

1. Factor out all common features;
2. Avoid unnecessary heap allocation;
3. Leverage compile time knowledge of data types.

Implementing these optimizations required us to add several features to TAO's ORB core. These included providing a pair of methods that can marshal and unmarshal parameters while the activation record of the stub and skeleton is active. This also means that parameters can now be allocated on the stack instead of from the heap.

Interpretive stub size reduction

As mentioned before, the interpreted stubs need the underlying stub object on which the `do_call` method is invoked. For this each stub is required to call `QueryInterface` on the object reference. `QueryInterface` is an internal operation in SunSoft IIOP that retrieves the underlying stub object that maintains information necessary to identify the object. This information contains the object key and the TCP/IP endpoint. Therefore, we factored out common code and added an operation on the `CORBA::Object` class to return the underlying stub object. The modified code fragment is shown in Appendix D.4.

Interpretive skeleton size reduction

As shown in Figure 7.2, each interpretive skeleton is required to create an `NVList` and populate it with parameters. In addition, memory is allocated from the heap rather than on the call stack for the `inout`, `out`, and `return` types. This is necessary since the marshaling of these parameters in the outgoing stream takes place after the call to the skeleton has returned. Applications using DSI must comply with this style. However, the ORB Core can be modified to provide the necessary operations that is used on by the IDL generated code. Applications cannot directly access these operations since they are protected.

Close scrutiny of the skeleton code reveals that each skeleton must create an `NVList` and populate it with parameters. Similarly, any return value must be stored in a `CORBA::Any`

data structure. These are common features that can be factored out into a method provided by the ORB Core. Based on this observation, we implemented a table-driven technique similar to the one used in the stubs. We defined two new interfaces to the marshaling engine that are similar to the `do_call` method. We could not reuse the `do_call` method since these two interfaces were required on the server-side "request" object. The space-efficient skeleton is shown in Figure 7.3. The corresponding code using this approach is shown Appendix D.5.

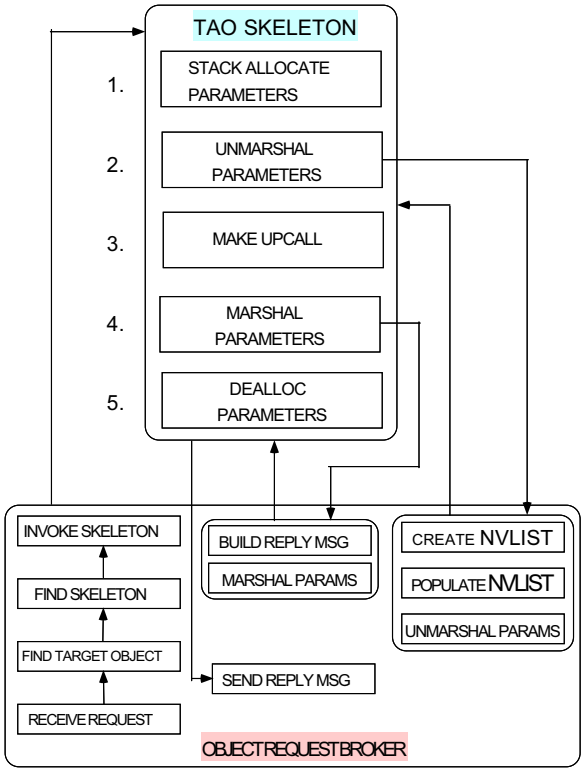


Figure 7.3: Optimized skeletons

The main benefit of this scheme is that marshaling of outgoing parameters can occur while the activation record on the call stack frame of the skeleton is still valid. As a result there is no need to allocate the `inout`, `out`, and `return` parameters from the heap. They can be allocated on the call stack in the same way that the compiled skeletons do. In addition, if any of the outgoing types is a pointer type, we can directly invoke the C++ `delete` operator rather than interpretively deallocating it.

The rest of the chapter describes the results of our experiments comparing the performance of the optimized interpretive stubs and skeletons with that of the hand-crafted compiled stubs and skeletons. In addition, we also report the individual sizes of the stubs and skeletons for both the approaches.

7.3 Experimental Setup

7.3.1 Hardware and Software Platforms

The experiments reported in this section were conducted on three different combinations of hardware and software, including:

- An UltraSPARC-II with two 300 MHz CPUs, a 512 Mbyte RAM, running SunOS 5.5.1, and C++ Workshop Compilers version 4.2;
- A Pentium Pro 200 with 128 Mbyte RAM running Windows NT 4.0 and the Microsoft Visual C++ 5.0 compiler;
- A Pentium Pro 180 with 128Mb RAM running Redhat Linux 4.2 kernel recompiled for SMP support and LinuxThreads 0.5. The GNU g++ 2.7.2.1 C++ compiler was used.

7.3.2 Profiling Tools

The code size information for various methods reported in Section 7.4 is obtained using the GNU `objdump` binary utility on SunOS 5.5.1 and Linux. On Window NT, we used the `dumpbin` binary utility. In both cases, we used the *disasm* and *linenumbers* options to disassemble the object code and insert line numbers in the assembly listing, respectively. Code size for individual stubs/skeletons is reported by counting the total number of bytes of assembly level instructions produced. In addition, we used the UNIX `strip` utility to measure the total size of the object code after removing the symbols and other debug information.

The profile information for the empirical analysis was obtained using the `Quantify` [35] performance measurement tool.

7.3.3 Parameter Types for Stubs/Skeletons

Appendix C provides the `Param_Test` IDL interface and its operations. All the operations test the four parameter passing modes (1) `in`, (2) `inout`, (3) `out`, and (4) `return` for a wide range of data types. The data types we tested include primitives such as `shorts`, and complex data types such as unbounded strings, fixed size structures, variable sized structures, nested structures, sequence of strings, and sequence of structures. All operations are two-way. Sequences are limited to a length of 9 elements and strings are 128 chars long.

7.3.4 Methodology

Each operation of the `Param_Test` interface is invoked 2,000 times. We measure the average latency for making the two-way call for the 2,000 iterations. Since we are interested in

measuring the performance of the stubs and skeletons, all tests were run in the loopback mode. This way we avoided any network transfer overhead. However, OS effects such as paging, context switching, and interrupts are measured. In addition, delays incurred due to the run-time costs of the implementation of the operation by the servant object are also measured. The servant object implements each operation by copying its `in` parameter into the `inout`, `out`, and `return` parameters. For complex data types such as `struct_sequence`, this overhead becomes significant compared to the others.

One approach to eliminating OS effects in the measurements is to use collocated objects. However, even though TAO supports co-located objects, we cannot use them in our measurements. This is due to the fact that operations on collocated objects result in a direct C++ method call on the target object. Collocated objects *i.e.,* objects residing in the same address space, entirely bypass the stubs/skeleton that perform the marshaling of data types. However, we are interested in measuring the overhead of marshaling. Forcing the collocated objects to use the stubs/skeletons required significant reengineering of the TAO ORB Core and the IDL Compiler. Therefore, we decided to approximate this behavior by running the tests in loopback mode. We configured our profiling tool `Quantify` (explained in Section 7.3.2) to measure only the overhead of the stubs and skeletons.

The code size of individual stubs and skeletons is measured using the GNU binary utility *objdump* and Windows NT's *dumpbin* as explained in Section 7.3.2.

7.4 Comparing Interpreted versus Compiled Marshaling

This section describes the results comparing the performance and code size of stubs and skeletons using interpretive and compiled form of marshaling. As explained in Section 7.3.4, each operation of the `Param_Test` interface is invoked 2,000 times. The tests are performed in a loopback mode to avoid unnecessary network delays. The two-way average latency of invoking the operations is reported. First, we report the performance results followed by comparison of the code sizes.

7.4.1 Comparing Twoway Average Latencies

Tables 7.1, 7.2, and 7.3 depict the two-way average latency for invoking different methods of the `Param_Test` for 2,000 iterations for the UltraSPARC, a PC running NT, and PC running Linux, respectively. Figure 7.4 illustrates this information graphically for UltraSPARC.

These tables indicate that the two-way latency of the interpreted stubs and skeletons is within 75 to 95 % of the compiled stubs for primitive types such as `shorts`, and complex types such as unbounded strings and fixed size structs. However, for other complex types such as `sequence` of `strings`, `sequence` of `structs`, variable-sized `structs`, and nested `structs`, the two-way latency for interpreted stubs/skeletons exceeds that of the compiled

Table 7.1: Twoway Latency of Stubs/Skeletons on UltraSPARC Running SunOS5.5.1

Data Type	Compiled		Interpreted	
	Avg time in msec	Calls/sec	Avg time in msec	Calls/sec
short	0.802	1,247	0.938	1,066
ubstring	0.907	1,102	1.065	938
fixed_struct	0.182	1,202	1.11	901
strseq	1.852	540	1.74	576
var_struct	2.014	497	2.020	493
nested_struct	2.011	497	2.102	476
struct_seq	10.99	91	10.25	98

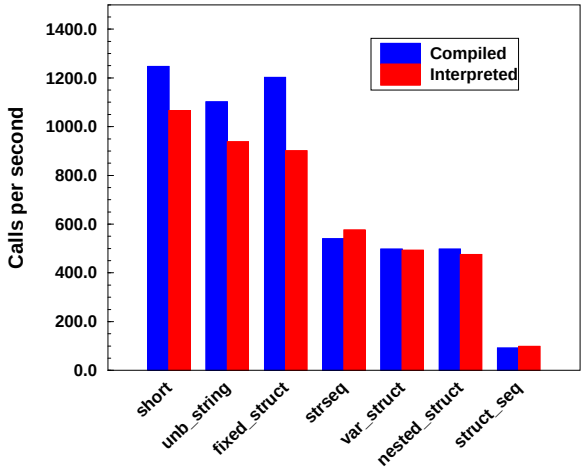


Figure 7.4: SPARC Performance

Table 7.2: Twoway Latency of Stubs/Skeletons on Pentium Pro 200 Running Windows NT 4.0

Data Type	Compiled		Interpreted	
	Avg time in msec	Calls/sec	Avg time in msec	Calls/sec
short	1.234	810	1.312	762
ubstring	1.453	688	1.491	670
fixed_struct	1.275	785	1.414	707
strseq	2.84	352	2.819	355
var_struct	3.29	325	2.97	336
nested_struct	3.489	325	3.02	331
struct_seq	22.93	44	17.247	58

Table 7.3: Twoway Latency of Stubs/Skeletons on Pentium Pro 180 Running Linux

Data Type	Compiled		Interpreted	
	Avg time in msec	Calls/sec	Avg time in msec	Calls/sec
short	0.745	1,342	0.8277	1,226
ubstring	0.114	1,094	0.982	1,017
fixed_struct	0.7735	1,293	0.898	1,112
strseq	2.21	452	1.814	551
var_struct	2.47	406	2.05	487
nested_struct	2.48	405	2.098	477
struct_seq	22.0	44	13.22	76

Table 7.4: Whitebox Analysis of Performance of Stubs/Skeletons on UltraSPARC

Data Type	Role	Type	Interpreted		Compiled	
			msec	called	msec	called
`fixed_struct`	server	marshal	84.21	6,000	13.76	6,000
		demarshal	57.29	4,000	9.93	4,000
	client	marshal	56.17	4,000	9.17	4,000
		demarshal	85.89	6,000	14.90	6,000
`strseq`	server	marshal	335.46	6,000	279.00	6,000
		demarshal	98.52	4,000	219.00	4,000
	client	marshal	95.43	4,000	68.76	4,000
		demarshal	256.24	6,000	665.71	6,000

stubs. The superior performance of the interpreted stubs/skeletons was more prominent to that of the compiled stubs/skeletons on the Pentium Pro 180 running Linux.

As mentioned in Section 7.3.4, these measurements do not exclude the effects of the OS, as well as the runtime costs of the implementation of the operations. The runtime costs of the implementation of the operations is more significant for the `test_struct_seq` case. Each `sequence` of `struct`s has 9 variable sized structs. Each variable-sized `struct` element in turn has two string members, each of length 128, and a `sequence` of `string` member. This member in turn has 9 `string` elements, each of length 128. The costs of copying the *in* parameter into the *inout*, *out*, and *return* is significant. However, irrespective of the type of marshaling used by the stubs and skeletons, the implementation of the operations is same in both cases. Hence, our comparisons of two-way latency are valid.

The blackbox results presented in Tables 7.1, 7.2, and 7.3 do *not* convey the effects of the OS, as well as runtime costs of the implementation of the operations. To pinpoint precisely the runtime costs of the stubs and skeletons in marshaling and demarshaling, we configured our profiling tool `Quantify` to measure only these costs. Table 7.4 illustrates the Quantify analysis for the `test_fixed_struct` and the `test_strseq` tests on the UltraSPARC platform.[3]

[3]We did not have Quantify for Linux and Windows NT.

Table 7.5: Sizes of Overloaded Operators for Compiled Stubs/Skeletons on UltraSPARC

Operator	Size
operator<< (char *)	192
operator>> (char *)	240
operator<< (fixed_struct)	280
operator>> (fixed_struct)	256
operator<< (strseq)	312
operator>> (strseq)	264
operator<< (var_struct)	176
operator>> (var_struct)	192
operator<< (nested_struct)	88
operator>> (nested_struct)	88
operator<< (struct_seq)	208
operator>> (struct_seq)	208

Table 7.4 indicates that for `fixed_struct`, the compiled stubs and skeletons accounted for 47.76 msec compared to 283.56 msec required for the interpretive stubs and skeletons. This explains why the compiled marshaling is significantly better than the interpretive marshaling for fixed size structs. On the other hand, for `sequence`s of `string`s, the compiled stubs and skeletons required 1,232.47 msec compared to only 785.65 msec by the interpretive stubs and skeletons. This explains why the interpretive stubs perform better than the compiled stubs for all the data types that are `sequence`s or have `sequence`s as their members.

TAO's interpretive marshaling engine has a highly optimized component for marshaling `sequence`s. It defines a generic base `sequence` class with virtual methods. For every user-defined `sequence`, the TAO IDL compiler generates a C++ class that inherits from this base `sequence` class. The C++ class generated for the `sequence`s (in accordance with the IDL to C++ mapping) overrides all the methods of the base class. The derived class does not define any data members since these are already defined in the base class. The interpreter marshals and demarshals `sequence`s by invoking methods on the base class. At runtime, due to polymorphism and dynamic binding, these calls are made on the derived class. This speeds up the marshaling and demarshaling of `sequence`s significantly.

7.4.2 Comparing Code Size for Stubs and Skeletons

This section describes the measurements of code size we did for the stubs and skeletons. As mentioned in Section 7.3.2, we used the GNU binary utility called `objdump` and NT's `dumpbin` to measure the individual code sizes.

Table 7.5 depicts the code sizes for the overloaded operators used for marshaling and demarshaling user-defined IDL data types. The code size of the `nested_struct` is only 88 bytes since internally it calls the overloaded operator for `var_struct`.

Table 7.6: Stub Sizes on UltraSPARC

Stub name	Interpreted size			Compiled size		
	stub	table	total	stub	helper	total
test_short	320	88	408	1,112		1,112
test_ubstring	352	88	440	1,000	432	1,432
test_fixed_struct	344	88	432	1,112	536	1,648
test_strseq	496	88	584	1,120	576	1,696
test_var_struct	496	88	584	1,120	368	1,488
test_nested_struct	496	88	584	1,120	176	1,296
test_struct_seq	496	88	584	1,120	416	1,536

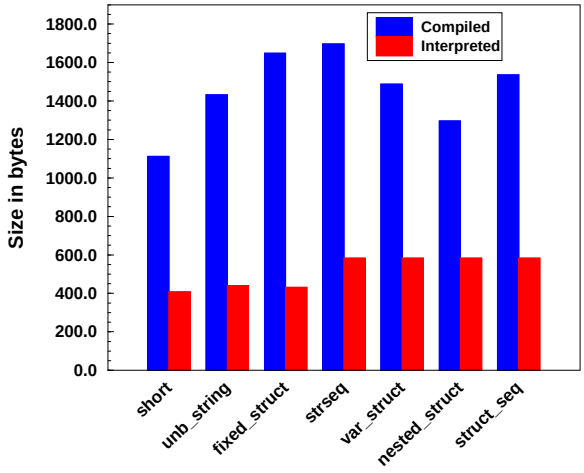


Figure 7.5: SPARC Stub Sizes

Tables 7.6 and 7.7 illustrate the code sizes for the stubs and skeletons, respectively. We account for the size of the tables in the size of the stubs and skeletons using interpreted marshaling. Hence the total size of the stub/skeleton is the size of the stub/skeleton and the size of the statically allocated tables. Similarly, for the compiled marshaling, we account for the size of helper overloaded operator methods used to marshal/demarshal user-defined data types. Since these helper methods are not inlined by the compiler, we account for them only once. Thus, although the `nested_struct`'s helper calls the helper for `var_struct`, we do not add the latter's size to the size computation of the stub/skeleton of `nested_struct`. Figures 7.5 and 7.6 illustrate this information graphically for the UltraSPARC platform.

Tables 7.6 and 7.7 indicate that the stubs for interpretive marshaling are much smaller than the ones for compiled marshaling. As shown in Section 7.4.3, the interpretive stub sizes are roughly 26-45% of the size of the compiled stubs. As shown in Section 7.2, in addition to explicitly marshaling and demarshaling parameters, the compiled stub must initialize a GIOP/IIOP request message and invoke it. On the contrary, for the interpretive stubs the `do_call` method provided by the ORB Core handles all this. The size of skeletons for compiled marshaling is relatively smaller since the `ServerRequest` object is

Table 7.7: Skeleton Sizes on UltraSPARC

Stub name	Interpreted size			Compiled size		
	skel	table	total	skel	helper	total
test_short_skel	440	88	528	544		544
test_ubstring_skel	552	88	640	688	432	1,120
test_fixed_struct_skel	480	88	568	584	536	1,120
test_strseq_skel	848	88	936	952	576	1,528
test_var_struct_skel	680	88	768	784	368	1,152
test_nested_struct_skel	680	88	768	784	176	960
test_struct_seq_skel	848	88	936	952	416	1,368

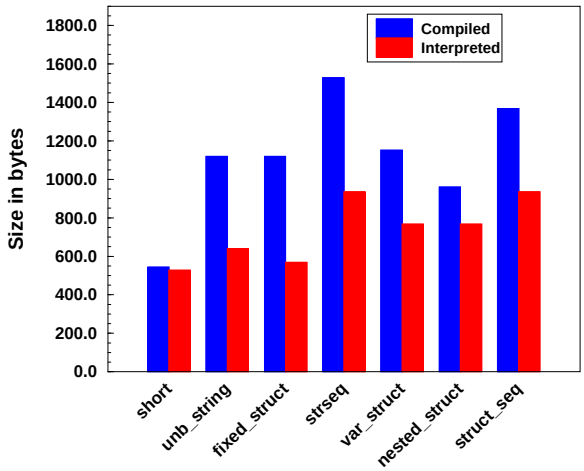


Figure 7.6: SPARC Skeleton Sizes

Table 7.8: Comparison of Interpretive with Compiled Code on UltraSPARC in Percentages

operation	Performance	Stub size	Skeleton size
test_short	85.48	36.69	97.06
test_ubstring	85.12	30.73	57.14
test_fixed_struct	74.96	26.21	50.17
test_strseq	106.66	34.43	61.26
test_var_struct	99.20	39.25	66.66
test_nested_struct	95.77	45.06	80.00
test_struct_seq	107.69	38.02	68.42

already available. The skeleton code size for primitives such as `shorts` are comparable for interpretive and compiled marshaling since the overloaded operators for primitives are provided by the ORB Core. As a result, there is no extra code generated. However, for the rest of the data types, the skeleton code size for the interpreted marshaling is between 50-80% of the compiled form.

The results of code size measurements for NT and Linux are shown in Appendix E. Although the code sizes are smaller for both types of marshaling compared to the code size on UltraSPARC, the relative measurements are comparable. There is one exception, however, where the code for the skeleton for shorts using compiled marshaling is slightly smaller compared to the interpretive one. This is due primarily to the extra overhead of the two statically allocated tables.

7.4.3 Summary of Comparisons

This section summarizes the results of Sections 7.4.1 and 7.4.2. Table 7.8 illustrates how interpreted stubs and skeletons compared with the compiled versions for the UltraSPARC platform. Similar results are observed for the other two platforms. Appendix E provides the detailed measurements. All values are in percentages.

Our results comparing the performance of the compiled and interpretive stubs indicate that on an average, the interpretive stubs perform 86% for primitive types, 75% for fixed size structures, and over 100% for data types with `sequences` as well as the compiled stubs. At the same time, the code size for user-defined types for interpreted stubs was 26-45% and for interpreted skeletons was 50-80% of the size of the compiled stubs and skeletons, respectively. For primitive types, the skeleton sizes were comparable. However, the interpreted stubs were $\sim$40% the size of the compiled stubs.

7.5 Benefits of TAO's Interpretive Stubs and Skeletons

This section illustrates how the efficiency and small footprint of TAO's interpretive stubs and skeletons can be useful in implementing a number of important CORBA services on memory-constrained systems.

Table 7.9 depicts a number of important CORBA higher-level services. We provide details on the number of user-defined structures, sequences, and total number of operations and/or attributes defined by their IDL definitions. We have not reported other data types such as unions, enums, and exceptions defined by these IDLs.

As shown in Sections 7.2 and 7.4, the total size of all the stubs and skeletons using compiled form of marshaling will exceed that of interpretive marshaling as the number of operations and user-defined types increase.

Table 7.9 shows examples of CORBA services such as the trading service, naming service, and others. As shown in the table, the IDL definitions for these services define a

Table 7.9: Number of operations and user-defined types in well-known OMG services

Service	structures	sequences	op/attributes
`CosTrading`	11	7	63
`AVStreams`	2	3	50
`CosPropertyService`	3	5	33
`CosNaming`	2	2	13

very large number of operations and/or attributes.[4] The total size of stubs and skeletons using compiled marshaling will far exceed that of interpretive marshaling.

In addition, for every user-defined type, the compiled form will produce overloaded operators to marshal and demarshal these types. As shown in Section 7.4, the performance of the interpreted stubs and skeletons is comparable or exceeds that of the compiled ones. At the same time, their code size is much smaller than the compiled ones.

7.6 Summary and Research Contributions

This chapter compares the performance and code size of stubs and skeletons using interpretive and compiled form of marshaling. The interpretive stubs and skeletons are generated by the TAO IDL compiler. The compiled stubs and skeletons are hand-crafted.

Our results comparing the performance of the compiled and interpretive stubs indicate that on an average, the interpretive stubs perform roughly 86% for primitive types, 75% for fixed size structures, and over 100% for data types with sequences as well as the

[4] Attributes are handled similar to operations. TAO's IDL compiler generates two operations for each read/write attribute.

compiled stubs. At the same time, the code size for user-defined types for interpreted stubs was roughly 26-45% and for interpreted skeletons was 50-80% of the size of the compiled stubs and skeletons, respectively. For primitive types, the skeleton sizes were comparable. However, the interpreted stubs were roughly 40% of the compiled stubs.

Our results are encouraging since the code size of the generated stubs and skeletons are significantly smaller. At the same time, they do not compromise on performance. Hence, these results indicate a positive step towards implementing efficient middleware for hand-held devices and other memory-constrained embedded systems. We are currently investigating techniques to implement the minimal ORB specification for which the OMG has recently issued request for proposals (RFP).

Chapter 8

Related Work

Our CORBA research focuses on optimizing communication middleware at multiple protocol layers and multiple levels of abstraction including the I/O subsystem, communication protocols, and higher-level CORBA implementation itself.

8.1 Related Work on Optimization Principles:

This section describes results from existing work on protocol optimization based on one or more of the principles in Table 5.1. We use these principles to optimize the performance of the IIOP marshaling engine. None of the related work mentioned below has dealt with optimizing middleware protocols.

8.1.1 Optimizing for the expected case

[6] describes a technique called *header prediction* that predicts the message header of incoming TCP packets. This technique is based on the observation that many members in the header remain constant between consecutive packets. This observation led to the creation of a template for the expected packet header. The optimizations reported in [6] are based on Principle 1, which *optimizes for the common case* and Principle 3, which is *precompute, if possible*. We present the results of applying these principles to optimize IIOP in Sections 5.3.3, 5.3.4, and 5.3.5.

8.1.2 Eliminating gratuitous waste

[7, 1, 4] describe the application of an optimization mechanism called *Integrated Layer Processing* (ILP). ILP is based on the observation that data manipulation loops that operate on the same protocol data are wasteful and expensive. The ILP mechanism integrates these loops into a smaller number of loops that perform all the protocol processing. The ILP optimization scheme is based on Principle 2, which *gets rid of gratuitous waste*. We

demonstrate the application of this principle to IIOP in Section 5.3.4 where we eliminated unnecessary calls to the `deep_free` method, which frees primitive data types. [4] cautions against improper use of ILP since this may increase processor cache misses.

8.1.3 Passing information between layers

Packet filters [44, 2, 14] are a classic example of Principle 6, which recommends *passing information between layers*. A packet filter demultiplexes incoming packets to the appropriate target application(s). Rather than having demultiplexing occur at every layer, each protocol layer passes certain information to the packet filter, which allows it to identify which packets are destined for which protocol layer. We applied Principle 6 for IIOP in Section 5.3.4 where we passed the `TypeCode` information and size of the element type of a `sequence` to the `TypeCode` interpreter. Therefore, the interpreter need not calculate the same quantities repeatedly.

8.1.4 Moving from generic to specialized functionality

[12] describes a facility called fast buffers (FBUFS). FBUFS combines virtual page remapping with shared virtual memory to reduce unnecessary data copying and achieve high throughput. This optimization is based on Principle 2, which focuses on *eliminating gratuitous waste* and Principle 3, which *replaces generic schemes with efficient, special purpose ones*. We applied these principles for IIOP in Section 5.3.4 where we incorporated the `struct_traverse` logic and some of the `decoder` logic into the `TypeCode` interpreter.

8.1.5 Improving cache-affinity

[47] describes a scheme called "outlining" that when used improves processor cache effectiveness, thereby improving performance. We describe optimizatons for processor cache in Section 5.3.5.

8.1.6 Efficient Demultiplexing

Demultiplexing routes messages between different levels of functionality in layered communication protocol stacks. Most conventional communication models (such as the Internet model or the ISO/OSI reference model) require some form of multiplexing to support interoperability with existing operating systems and protocol stacks. In addition, conventional CORBA ORBs utilize several extra levels of demultiplexing at the application layer to associate incoming client requests with the appropriate servant and operation (as shown in Figure 6.1). Layered multiplexing and demultiplexing is generally disparaged for high-performance communication systems [73] due to the additional overhead incurred at each layer. [14] describes a fast and flexible message demultiplexing strategy based on dynamic

code generation. [24] evaluates the performance of alternative demultiplexing strategies for real-time CORBA.

Our results for latency measurements have shown that with increasing number of servants, the latency increases. This is partly due to the additional overhead of demultiplexing the request to the appropriate operation of the appropriate servant. TAO uses a de-layered demultiplexing architecture [24] that can select optimal demultiplexing strategies based on compile-time and run-time analysis of CORBA IDL interfaces. Related work on efficient demultiplexing is mostly restricted to efficiently demultiplexing packets through the protocol layers inside the operating system.

8.2 Related Work on Presentation Layer Conversions

8.2.1 Interpretive versus Compiled forms of marshaling:

SunSoft IIOP uses an interpretive marshaling/demarshaling engine. An alternative approach is to use *compiled* marshaling/demarshaling. A compiled marshaling scheme is based on *a priori* knowledge of the type of an object to be marshaled. Thus, in this scheme there is no necessity to decipher the type of the data to be marshaled at run-time. Instead, the type is known in advance, which can be used to marshal the data directly.

[33] describes the tradeoffs of using compiled and interpreted marshaling schemes. Although compiled stubs are faster, they are also larger. In contrast, interpretive marshaling is slower, but smaller in size. [33] describes a hybrid scheme that combines compiled and interpretive marshaling to achieve better performance. This work was done in the context of the ASN.1/BER encoding [36].

According to the SunSoft IIOP developers, interpretive marshaling is preferable since it decreases code size and increases the likelihood of remaining in the processor cache. As explained in Chapter 9, we are currently implementing a CORBA IDL compiler [26] that can generate compiled stubs and skeletons. Our goal is to generate efficient stubs and skeletons by extending optimizations provided in USC [52] and "Flick" [13], which is a flexible, optimizing IDL compiler. Flick uses an innovative scheme where intermediate representations guide the generation of optimized stubs. In addition, due to the intermediate stages, it is possible for Flick to map different IDLs (*e.g.,* CORBA IDL, ONC RPC IDL, MIG IDL) to a variety of target languages such as C, C++. TAO's IDL compiler implements optimizations to improve the performance of its interpretive stubs. The stubs and skeletons produced by USC and Flick are compiled in nature.

8.2.2 Presentation Layer and Data Copying

The presentation layer is a major bottleneck in high-performance communication subsystems [7]. This layer transforms typed data objects from higher-level representations to lower-level representations (marshalling) and vice versa (demarshalling). In both RPC toolkits and CORBA, this transformation process is performed by client-side stubs and server-side skeletons that are generated by interface definition language (IDL) compilers. IDL compilers translate interfaces written in an IDL (such as Sun RPC XDR [71], DCE NDR, or CORBA CDR [51]) to other forms such as a network wire format. A significant amount of research has been devoted to developing efficient stub generators. We cite a few of these and classify them as below.

- **Annotating high level programming languages:**

The Universal Stub Compiler (USC) [52] annotates the C programming language with layouts of various data types. The USC stub compiler supports the automatic generation of device and protocol header marshalling code. The USC tool generates optimized C code that automatically aligns data structures and performs network/host byte order conversions. TAO and its IDL compiler does not use annotations.

- **Generating code based on Control Flow Analysis of interface specification:**

[33] describes a technique of exploiting application-specific knowledge contained in the type specifications of an application to generate optimized marshalling code. This work tries to achieve an optimal tradeoff between interpreted code (which is slow but compact in size) and compiled code (which is fast but larger in size). A frequency-based ranking of application data types is used to decide between interpreted and compiled code for each data type. Our implementations of the stub compiler will be designed to adapt according to the runtime access characteristics of various data types and methods. The runtime usage of a given data type or method can be used to dynamically link in either the compiled or the interpreted version. Dynamic linking has been shown to be useful for mid-stream adaptation of protocol implementations [61].

Our goal is to extend TAO IDL compiler's capabilites such that it can generate a mix of interpreted as well as compiled stubs and skeletons as specified by the user for optimal performance.

- **Using high level programming languages for distributed applications:**

[54] describes a stub compiler for the C++ language. This stub compiler does not need an auxiliary interface definition language. Instead, it uses the operator overloading feature of C++ to enable parameter marshalling. This approach enables distributed applications to

be constructed in a straightforward manner. A drawback of using a programming language like C++ is that it allows programmers to use constructs (such as references or pointers) that do not have any meaning on the remote side. Instead, IDLs are more restrictive and disallow such constructs. CORBA IDL has the added advantage that it resembles C++ in many respects and a well-defined mapping from the IDL to C++ has been standardized.

Application Level Framing and Integrated Layer Processing on Communication Subsystems

Conventional layered protocol stacks and distributed object middleware lack the flexibility and efficiency required to meet the quality of service requirements of diverse applications running over high-speed networks. One proposed remedy for this problem is to use *Application Level Framing* (ALF) [7, 5, 19] and *Integrated Layer Processing* (ILP) [7, 1, 61].

ILP ensures that lower layer protocols deal with data in units specified by the application. ILP provides the implementor with the option of performing all data manipulations in one or two integrated processing loops, rather than manipulating the data sequentially. [4] have shown that although ILP reduces the number of memory accesses, it does not reduce the number of cache misses compared to a carefully designed non-ILP implementation.

A major limitation of ILP described in [4] is its applicability to only non-ordering constrained protocol functions and its uses of macros that restrict the protocol implementation from being dynamically adapted to changing requirements.

As shown by our results, CORBA ORBs suffer from a number of overheads that includes the many layers of software and large chain of function calls. We plan to use integrated layer processing to minimize the overhead of the various software layers. We are developing a factory of ILP based `inline` functions that are targeted to perform different functions. This allows us to dynamically link required functionality as the requirements change and yet have an ILP-based implementation.

8.3 Optimizations to the lower layers of the Protocol Stack

Existing research in gigabit networking has focused extensively on enhancements to TCP/IP. None of the systems described below are explicitly targeted for the requirements and constraints of communication middleware like CORBA. In particular, less attention has been paid to integrating the following topics related to communication middleware:

8.3.1 Transport Protocol Performance over ATM Networks

The underlying transport protocols used by the ORB must be flexible and possess the necessary hooks to tune different parameters of the underlying transport protocol. [9, 11, 45] present results on performance of TCP/IP (and UDP/IP [9]) on ATM networks by varying

a number of parameters (such as TCP window size, socket queue size, and user data size). This work indicates that in addition to the host architecture and host network interface, parameters configurable in software (like TCP window size, socket queue size, and user data size) significantly affect throughput. [9] shows that UDP performs better than TCP over ATM networks, which is attributed to redundant TCP processing overhead on highly-reliable ATM links. [9] also describes techniques to tune TCP to be a less bulky protocol so that its performance can be comparable to UDP. They also show that the TCP delay characteristics are predictable and that it varies with the throughput.

[39] present detailed measurements of various categories of processing overhead times of TCP/IP and UDP/IP. The authors conclude that whenever a realistic distribution of message sizes is considered, the aggregate costs of non-data touching overheads (such as network buffer manipulation) consume a majority of the software processing time (84% for TCP and 60% for UDP). The authors show that most messages sent are short (less than 200 bytes). They claim that these overheads are hard to eliminate and techniques such as integrated layer processing can be used to reduce the overhead. [53] presents performance results of the SunOS 4.x IPC and TCP/IP implementations. They show that increasing the socket buffer sizes improves the IPC performance. They also show that the socket layer overhead is more significant on the receiver side. [43] discusses the `TCP_NODELAY` option, which allows TCP to send small packets as soon as possible to reduce latency.

Earlier work [20, 21] using untyped data and typed data in a similar CORBA/ATM testbed as the one in this dissertation reveal that the low-level C socket version and the C++ socket wrapper versions of TTCP are nearly equivalent for a given socket queue size. Likewise, the performance of Orbix for sequences of scalar data types is almost the same as that reported for untyped data sequences. However, the performance of transferring sequences of CORBA `structs` for 64 K and 8 K socket queue sizes was much worse than those for the scalars. This overhead arises from the amount of time the CORBA ORBs spend performing presentation layer conversions and data copying.

8.3.2 High-performance I/O subsystems

The `FBUF` system [12] is a data transport service used in the *x*-kernel [34]. The `fbuf` system consists of a data transport service that uses page remapping. Other I/O subsystems [27, 29] provide a broad range of services including real-time upcalls and clock-driven interrupts for polling I/O devices and applications to support periodic data transfer and to reduce asynchronous interrupts and system calls.

8.3.3 Demultiplexing

Demultiplexing is a task that routes messages between different levels of functionality in layered communication protocol stacks. Most conventional communication models (such

as the Internet model or the ISO/OSI reference model) require some form of multiplexing to support interoperability with existing operating systems and protocol stacks. Conventional CORBA implementations utilize several additional levels of demultiplexing at the application layer to associate incoming CORBA requests with the appropriate object implementation and method. Layered multiplexing and demultiplexing is generally disparaged for high-performance communication systems [73] due to the additional overhead incurred at each layer.

Related work on demultiplexing focuses largely on the lower layers of the protocol stack (*i.e.*, the transport layer and below) as opposed to the CORBA middleware. For instance, [73, 16, 10] study demultiplexing issues in communication systems and show how layered demultiplexing is not suitable for applications that require real-time quality of service guarantees.

Packet filters are a mechanism for efficiently demultiplexing incoming packets to application endpoints [46]. A number of schemes to implement fast and efficient packet filters are available. These include the BSD Packet Filter (BPF) [44], the Mach Packet Filter (MPF) [77], PathFinder [2], demultiplexing based on automatic parsing [37], and the Dynamic Packet Filter (DPF) [14].

As mentioned before, most existing demultiplexing strategies are implemented within the OS kernel. However, to optimally reduce ORB endsystem demultiplexing overhead requires a vertically integrated architecture that extends from the OS kernel to the application servants. Since our ORB is currently implemented in user-space, however, our work focuses on minimizing the demultiplexing overhead in steps 3, 4, 5, and 6 (which are shaded in Figure 6.1).

Our framework uses a delayered demultiplexing architecture to select optimal demultiplexing strategies based on compile-time and run-time analysis of CORBA IDL interfaces.

Chapter 9

Concluding Remarks and Future Work

9.1 Conclusions

An important class of applications (such as avionics, distributed interactive simulation, and telecommunication systems) require scalable, low-latency communication. However, the results presented in Chapters 2, 3, and 4 indicate that conventional ORBs do not yet support latency-sensitive applications and servers that support a large number of servants. The chief sources of ORB latency and scalability overhead arise from (1) long chains of intra-ORB function calls, (2) excessive presentation layer conversions and data copying, (3) non-optimized buffering algorithms used for network reads and writes, (4) inefficient server demultiplexing techniques, and (5) lack of integration with OS and network features.

Our goal in precisely pinpointing the sources of overhead for CORBA is to optimize the performance of TAO [69]. TAO is a high-performance, real-time ORB endsystem designed to meet the QoS requirements of bandwidth- and delay-sensitive applications. Our development strategy for TAO is guided by applying *principle-driven performance optimizations* [25], such as optimizing for the common case; eliminating gratuitous waste; replacing general purpose methods with specialized, efficient ones; precomputing values, if possible; storing redundant state to speed up expensive operations; passing information between layers; optimizing for the processor cache; and optimizing demultiplexing strategies.

Applying these optimizations to TAO reduced its latency by a factor of ∼1.5 to 2.0 times for primitive data types and around 4 times for richly-typed data such as `BinStruct`. The performance of TAO is now equal to, or better than, commercial ORBs using static invocation. Moreover, TAO's dynamic invocation implementation is 2 to 4.5 times faster than commercial ORBs, depending on the data types.

The source code for the TAO ORB and the benchmarking tests reported in this paper are available at `www.cs.wustl.edu/~schmidt/TAO.html`.

9.2 Future Work

Existing markets for hand-held devices, such as Personal Digital Assistants (PDAs), are being revolutionized by the advent of newer operating systems for hand-held devices such as Inferno, Windows CE 2.0, and Palm OS. Analysts estimate in excess of five million units of PDAs being sold by 1999. However, responses to recent surveys by users of PDAs indicate a dearth of software and applications. Many users require PDAs to possess high-speed, built-in data/cellular/fax modems that enable the PDA to be used as a cellular phone, a fax machine, and for sending and receiving emails, and browsing the internet.

Adding efficient and predictable communication capability to hand-held devices yields many research challenges related to mobile computing [17]. These challenges include dealing with low bandwidth, heterogeneity in the network connections, frequent changes and disruptions in the established connections due to migrating targets, maintaining consistency of data, and dealing with heterogeneous architectures to which these devices can be docked. In addition to the mobility issues, the restrictions on the physical size and power consumptions of these devices constrains the amount of storage capabilities possessed by these devices.

CORBA [51] is a distributed object computing middleware standard defined by the Object Management Group (OMG). CORBA is designed to allow clients to invoke operations on remote objects without concern for where the object resides or what language the object is written in. In addition, CORBA shields applications from non-portable details related to the OS/hardware platform they run on and the communication protocols and networks used to interconnect distributed objects. These benefits of CORBA make it ideally suited to provide core communication services for the hand-held devices.

The challenge is to maintain a small footprint for the ORB core, the OMG IDL compiler, and the generated stubs. Additionally, the performance of the generated stubs should not be compromised due to constraints on the footprint.The OMG has recently issued a Request for Proposals (RFPs) for a minimal-CORBA implementation geared towards embedded systems and other special systems that have constraints on the available resources such as memory.

Chapter 7 compares the performance and code size of stubs and skeletons using interpretive and compiled form of marshaling. The interpretive stubs and skeletons are generated by the TAO IDL compiler. The compiled stubs and skeletons are hand-crafted.

Our results comparing the performance of the compiled and interpretive stubs indicate that on an average, the interpretive stubs perform roughly 86% for primitive types,

75% for fixed size structures, and over 100% for data types with sequences as well as the compiled stubs. At the same time, the code size for user-defined types for interpreted stubs was roughly 26-45% and for interpreted skeletons was 50-80% of the size of the compiled stubs and skeletons, respectively. For primitive types, the skeleton sizes were comparable. However, the interpreted stubs were roughly 40% of the compiled stubs.

Our results are encouraging since the code size of the generated stubs and skeletons are significantly smaller. At the same time, they do not compromise on performance. Hence, these results indicate a positive step towards implementing efficient middleware for hand-held devices and other memory-constrained embedded systems. We are currently investigating techniques to implement the minimal ORB specification for which the OMG has recently issued request for proposals (RFP).

Appendix A

IDL Definitions for Performance Experiments

The following `struct` declarations are representative of those used for the C/C++ and hand-optimized RPC implementations of TTCP:

C/C++ Struct Declarations

```
struct BinStruct {
        short s;
        char c;
        long l;
        octet o;
        double d;
};

typedef struct BinStruct BinStruct;
typedef struct {
        u_long type;
        u_long len;
        double *buffer;
}DoubleSeq;

typedef struct {
        u_long type;
        u_long len;
        BinStruct *buffer;
}StructSeq;
```

RPCL Definition

```
struct BinStruct {
    short s;
    char c;
    long l;
    octet o;
    double d;
};

typedef short ShortSeq<>;
typedef long LongSeq<>;
typedef char CharSeq<>;
typedef octet OctetSeq<>;
typedef double DoubleSeq<>;
typedef BinStruct StructSeq<>;

program TTCP {
    version TTCPVERS {
        void SEND_SHORT(ShortSeq) = 1;
        void SEND_LONG(LongSeq) = 2;
        void SEND_CHAR(CharSeq) = 3;
        void SEND_OCTET(OctetSeq) = 4;
        void SEND_DOUBLE(DoubleSeq) = 5;
        void SEND_STRUCT(StructSeq) = 6;
    } = 1;
} = 0x20000001;
```

The following CORBA IDL interface was used for the Orbix and ORBeline CORBA implementations:

```
// sizeof BinStruct == 24 bytes
// due to compiler alignment
struct BinStruct{ short s; char c; long  l;
                  octet o; double d; };

// Richly typed data
interface ttcp_sequence {
  typedef sequence<short> ShortSeq;
  typedef sequence<long> LongSeq;
  typedef sequence<double> DoubleSeq;
  typedef sequence<char> CharSeq;
  typedef sequence<octet> OctetSeq;
  typedef sequence<BinStruct> StructSeq;

  // Routines to send sequences of various data types
  oneway void sendShortSeq (in ShortSeq ttcp_seq);
  oneway void sendLongSeq (in LongSeq ttcp_seq);
  oneway void sendDoubleSeq (in DoubleSeq ttcp_seq);
  oneway void sendCharSeq (in CharSeq ttcp_seq);
  oneway void sendOctetSeq (in OctetSeq ttcp_seq);
  oneway void sendStructSeq (in StructSeq ttcp_seq);

  // to measure time taken for receipt of data
  oneway void start_timer ();
  oneway void stop_timer ();
};
```

Appendix B

The General Inter-ORB Protocol and the Internet Inter-ORB Protocol

B.1 Overview of the CORBA GIOP and IIOP Protocols

This section describes the components in the CORBA General Inter-ORB Protocol (GIOP) and Internet Inter-ORB Protocol (IIOP) protocol in detail.

B.1.1 Common Data Representation (CDR)

The GIOP Common Data Representation (CDR) defines a transfer syntax for transmitting OMG IDL data types across a network. The CDR definition maps the OMG IDL data types from their native host format into a bi-canonical network-level representation, which supports both little-endian and the big-endian formats. The salient features of CDR are:

Variable byte ordering: The sender encodes the data using its native byte-order. The byte order used by the sender is indicated by a special flag in the encoded stream. Thus, only receivers with byte ordering different from the sender must swap bytes to retrieve correct binary values.

Aligned Types: Primitive OMG IDL data types are aligned on their "natural" boundaries within GIOP messages, as shown in Table B.1.

Constructed data types (such as IDL `sequence` and `struct`) have no additional alignment restrictions beyond those of their primitive types. Thus, the size and alignment of the constructed type will depend on the size and alignment of the primitives that make up constructed type.

Table B.1: Alignment of Primitive Types

Type	Alignment
char	1
octet	1
short	2
unsigned short	2
long	4
unsigned long	4
float	4
double	8
boolean	1
enum	4

Complete OMG IDL mapping: CDR provides a mapping for all the OMG IDL data types, including transferable pseudo-objects such as `TypeCodes`. CORBA pseudo-objects are those entities that are neither CORBA primitive types nor constructed types. A client acquiring a reference to a pseudo-object cannot use DII to make calls to the methods described by the IDL interface of that pseudo-object. The DSI and DII interpreters use the `TypeCode` information passed to them by users of DSI and DII, respectively.

CDR Encapsulations: A CDR encapsulation is a `sequence` of *octets*. Encapsulations are typically used to marshal parameters of the following types:

• **TypeCodes:** OMG defined `TypeCodes` and their CDR encoding rules are shown in Table B.2.

• **IIOP protocol profiles inside interoperable object references (IORs):** A protocol profile provides information about the transport protocol that enables client applications to talk to the servers. In the IIOP profile (Figure B.1), this informations consists of the host name and port number on which the server is listening, and the object_key of the target object implemented by that server.

An IOR (see Figure B.2) represents a complete information about an object. This information includes the type it represents, the protocols it supports, whether it is null or not, and any ORB related services that are available.

• **Service-specific contexts:** The CORBA Object Services (COS) specification [48] defines a wide range of services, such as transactions, events, naming, concurrency, and audio/video streaming. For interoperability, it may be required to pass service-specific information via opaque parameters. This is achieved using the `service-specific context`.

Table B.2: TypeCode Enum Values, Parameter List Types, and Parameters

TCKind	Value (integer)	Type	Parameters
tk_null	0	empty	none
tk_void	1	empty	none
tk_short	2	empty	none
tk_long	3	empty	none
tk_ushort	4	empty	none
tk_ulong	5	empty	none
tk_float	6	empty	none
tk_double	7	empty	none
tk_boolean	8	empty	none
tk_char	9	empty	none
tk_octet	10	empty	none
tk_any	11	empty	none
tk_TypeCode	12	empty	none
tk_Principal	13	empty	none
tk_objref	14	complex	string(repository ID), string (name),
tk_struct	15	complex	string(repository ID), string (name), ulong(count), {string(member name), TypeCode(member type)}
tk_union	16	complex	string(repository ID), string (name), TypeCode (discriminant type), long(default used), ulong(count) {discriminant type(label val), string(member name), TypeCode (member type)}
tk_enum	17	complex	string(repository ID), string (name), ulong(count), {string(member name)}
tk_string	18	complex	ulong(max length)
tk_sequence	19	complex	TypeCode (element type), ulong (max length)
tk_array	20	complex	TypeCode (element type), ulong (max length)
tk_alias	21	complex	string(repository ID), string (name), TypeCode
tk_except	22	complex	string(repository ID), string (name), ulong (count), {string (member name), TypeCode (member type) }
none	0xffffffff	simple	string(repository ID), string (name)

The first byte of an encapsulation always denotes the byte-order used to create the encapsulation. Encapsulations can be nested inside of other encapsulations. Each encapsulation can use any byte-order, irrespective of its other encapsulations.

B.1.2 GIOP Message Formats

The GIOP specification defines seven standard message types. Each message is assigned a unique value. The originator of a `giop` message can be a client and/or a server. The Table B.3 depicts the seven types of messages and the permissible originators of these messages:

A GIOP message begins with a GIOP header (Figure B.3), followed by one of the message types (Figure B.4), and finally the body of the message, if any.

```
module IIOP {
    struct Version {
        char major;  // the number 1
        char minor;  // the number 0
    };

    struct ProfileBody {
        Version         iiop_version; //protocol version
        string          host;         //host name
        unsigned short  port;         //port number
        sequence<octet> object_key;   //opaque key
                                      //identifying the
                                      //object
    };
};
```

Figure B.1: Definition of IIOP Profile

```
module IOP {
    typedef unsigned long   ProfileId;
    const ProfileId         TAG_INTERNET_IOP = 0;

    struct TaggedProfile {
        ProfileId tag; //any one of the previously
                       //defined tag values
        sequence<octet> profile_data;
        //protocol specific data
    };

    // Interoperable Object Reference
    struct IOR {
        string  type_id; //assigned by the
                         //interface repository
        sequence<TaggedProfile>
            profiles; //profile information
    };
};
```

Figure B.2: Definition of an Interoperable Object Reference

```
module GIOP {
    enum MsgType {
        Request,
        Reply,
        CancelRequest,
        LocateRequest,
        LocateReply,
        CloseConnection,
        MessageError
    };

    struct Version {
        char major;  // the number 1
        char minor;  // the number 0
    };

    struct MessageHeader {
        char            magic[4];
        Version         GIOP_version; //protocol version
        boolean         byte_order;   //0=>big endian
        octet           message_type; //one of 7 types
        unsigned long   message_size; //length of msg
    };
};
```

Figure B.3: GIOP header

Table B.3: GIOP Message Types

Message Type	Originator	Value
Request	Client	0
Reply	Server	1
CancelRequest	Client	2
LocateRequest	Client	3
LocateReply	Server	4
CloseConnection	Server	5
MessageError	Both	6

B.1.3 GIOP Message Transport

The GIOP specification makes certain assumptions about the transport protocol that can be used to transfer GIOP messages. These assumptions are listed below:

- The transport mechanism must be connection-oriented;
- The transport protocol must be reliable;
- The transport data is a byte stream without message delimitations;
- The transport provides notification of disorderly connection loss;
- The transport's model of establishing a connection can be mapped onto a general connection model (such as TCP/IP).

Examples of transport protocols that meet these requirements are TCP/IP and OSI TP4.

B.1.4 Internet Inter-ORB Protocol (IIOP)

The IIOP is a specialized mapping of GIOP onto the TCP/IP protocols. ORBs that use IIOP can communicate with other ORBs that publish their TCP/IP addresses as interoperable object references (IORs). IIOP IOR profiles are shown in Figure B.1. Figure B.2 shows the format of an IOR. When IIOP is used, the `profile_data` member of the `TaggedProfile` structure holds the profile data of IIOP shown in Figure B.1.

B.2 TTCP IDL Description and TypeCode Layout

The following CORBA IDL interface was used in our experiments to measure the throughput of SunSoft IIOP described in Section 5.3.1. An example of a `TypeCode` description for `BinStruct` is presented in Section 5.2.2.

```
// Richly typed data.
interface ttcp_throughput
{
  typedef sequence<short> ShortSeq;
  typedef sequence<long> LongSeq;
  typedef sequence<double> DoubleSeq;
  typedef sequence<char> CharSeq;
  typedef sequence<octet> OctetSeq;
  typedef sequence<PerfStruct> StructSeq;

  // Methods to send various data type sequences.
  oneway void sendShortSeq (in ShortSeq ts);
  oneway void sendLongSeq (in LongSeq ts);
  oneway void sendDoubleSeq (in DoubleSeq ts);
  oneway void sendCharSeq (in CharSeq ts);
  oneway void sendOctetSeq (in OctetSeq ts);
  oneway void sendStructSeq (in StructSeq ts);

  oneway void start_timer ();
  oneway void stop_timer ();
};
```

Figure B.5 shows the representation of the `TypeCode` layout that defines the `sequence` of `BinStruct`s from Section 5.2.2. An IDL compiler is responsible for generating `TypeCode` information for all the data types described in an IDL definition. The `TypeCode` information generated by an IDL compiler is available at both the sender and the receiver end, which obviates the need to transmit typecode information along with the data over the network. Since SunSoft IIOP does not provide an IDL compiler the `TypeCode` information for all the `BinStruct` types was hand-crafted.

The layout of the `sequence` of `BinStruct`s and its parameters are shown in Table B.2 and described below:

TypeCode value: A CORBA `TypeCode` data structure contains a `_kind` field that indicates the `TCKind` value, which is an enumerated type. For instance, the `TCKind` value is `tk_sequence` in the "`sequence` of `BinStruct`" example.

TypeCode length and byte order: The `length` field indicates the length of the buffer that holds the CDR representation of the `TypeCode`'s parameters. In this example, the first byte of the CDR buffer indicates the byte order. Here, the IIOP standard value 0 indicates that big-endian byte-ordering is used.

Element type: For a `sequence` `TypeCode`, the next entry in the buffer is the `TypeCode` kind entry for the element type that comprises the `sequence`. In our example, this value is `tk_struct`.

Encapsulation length and sequence bound: The next entry is a length field indicating the length of the encapsulation that holds information about the `struct`'s members. The length field is followed by the encapsulation, which is followed by a field that indicates the bounds of the `sequence`. A value of 0 indicates an "unbounded" `sequence` (*i.e.*, the size of the `sequence` is determined at run-time, not at compile-time).

Encapsulation content and field layouts: The encapsulation contains two string entries, which follow the designation of the encapsulation's byte-order. Each string entry has a field specifying the length of the string followed by the string values. The first string specifies the "type ID" assigned by the interface repository. The second string holds the actual name of the data type as defined in the IDL definition. After this field is the number of members in the `BinStruct` IDL `struct`. This is followed by `TypeCode` layouts for each field (*e.g.* `short`, `char`, `long`, etc.) in the `struct`.

B.3 Tracing the Data Path of an IIOP Request

To illustrate the run-time behavior of SunSoft IIOP, we trace the path taken by requests that transmit a `sequence` of `BinStruct`s (shown in Appendix B.2). We show how the `TypeCode` interpreter consults the `TypeCode` information as it marshals and unmarshals parameters. We use the same `BinStruct` in this example and in our optimization experiments described in Section 5.3.1.

Client-side Data Path: The client-side data path is shown in Figure B.6. This figure depicts the path traced by outgoing client requests through the `TypeCode` interpreter. The `CDR::encoder` method marshals the parameters from native host format into a CDR representation suitable for transmission on the network.

The client uses the `do_call` method, which is the SII API provided by SunSoft IIOP that uses the `TypeCode` interpreter to marshal the parameters and send the client requests. The DII mechanism uses the `do_dynamic_call` method to send client requests.

Although the `do_call` and `do_dynamic_call` methods play similar roles, their type signatures are different. The `do_call` is used by the IDL compiler generated stubs to send client requests. The `do_dynamic_call` is used by the ORB's API (*e.g.* `send_oneway` and `invoke`) for DII to send client requests. The `do_dynamic_call` is passed an `NVList` representing all the parameters to the operation being invoked. In addition, it is passed a flag indicating whether the operation is oneway or two-way, a `string` argument that represents the operation name, and a `NamedValue` pseudo-object that holds the results.

The `do_call` method creates a CDR stream into which operations for CORBA parameters are marshaled before they are sent over the network. To marshal the parameters, `do_call` uses the `CDR::encoder` `visit` method. For primitive types (such as `octet`, `short`, `long`, and `double`), the `CDR::encoder` method marshals them into the CDR stream using the lowest-level `CDR::put` methods. For constructed data types (such as IDL `structs` and `sequences`), the `encoder` recursively invokes the `TypeCode` interpreter.

The `traverse` method of the `TypeCode` interpreter consults the `TypeCode` layout passed to it by an application to determine the data types that comprise a constructed data type. For each member of a constructed data type, the interpreter invokes the same `visit`

method that invoked it. In our case, the `encoder` is the `visit` method that originally called the interpreter. This process continues recursively until all parameters have been marshaled. At this point the request is transmitted over the network via the `invoke` method of the `GIOP::Invocation` class.

Server-side Data Path: The server-side data path is shown in Figure B.7. This figure depicts the path traced by incoming client requests through the `TypeCode` interpreter. An event handler (TCP_OA) waits in the ORB Core for incoming data. After a CORBA request is received, its GIOP type is decoded and the Object Adapter demultiplexes the request to the appropriate method of the target object. The `CDR::decoder` method then unmarshals the parameters from the CDR representation into the server's native host format. Finally, the server's dispatching mechanism dispatches the request to the skeleton of the target object via a user-supplied upcall method.

The SunSoft IIOP receiver supports the DSI mechanism. Therefore, an `NVList` CORBA pseudo-object is created and populated with the `TypeCode` information for the parameters retrieved from the incoming request. These parameters are retrieved by calling the `params` method of the `ServerRequest` class. Similar to the client-side data path, the server's `TypeCode` interpreter uses the `CDR::decoder` `visit` method to unmarshal individual data types into a parameter list. These parameters are subsequently passed to the server application's upcall method.

```
module GIOP {
    struct RequestHeader{
        IOP::ServiceContextList service_context;
        unsigned long   request_id;
        boolean         response_expected;
        sequence<octet> object_key;
        string          operation;
        Principal       requesting_principal;
    };

    enum ReplyStatusType {
        NO_EXCEPTION,
        USER_EXCEPTION,
        SYSTEM_EXCEPTION,
        LOCATION_FORWARD
    };

    struct ReplyHeader {
        IOP::ServiceContextList service_context;
        unsigned long   request_id;
        ReplyStatusType reply_status;
    };

    struct CancelRequestHeader {
        unsigned long   request_id;
        sequence<octet> object_key;
    };

    struct LocateRequestHeader {
        unsigned long   request_id;
        sequence<octet> object_key;
    };

    enum LocateStatusType {
        UNKNOWN_OBJECT,
        OBJECT_HERE,
        OBJECT_FORWARD
    };

    struct LocateReplyHeader {
        unsigned long     request_id;
        LocateStatusType  locate_status;
    };
};
```

Figure B.4: GIOP Messages

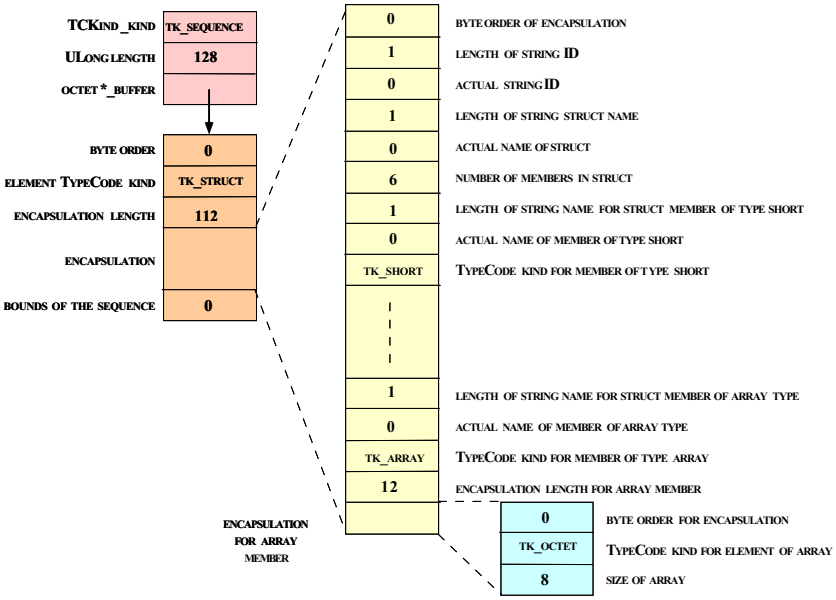


Figure B.5: TypeCode for Sequence of BinStruct

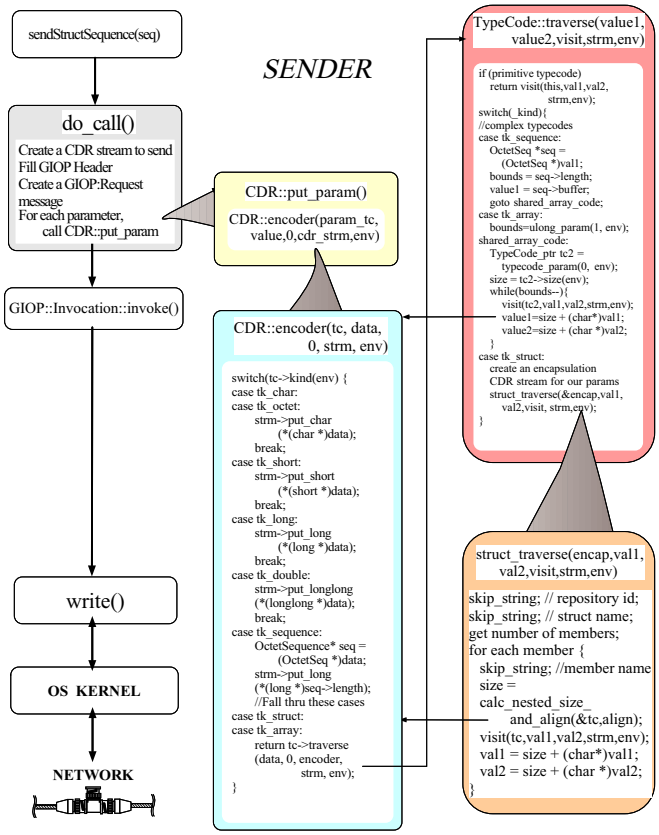


Figure B.6: Sender-side Datapath for the Original SunSoft IIOP Implementation

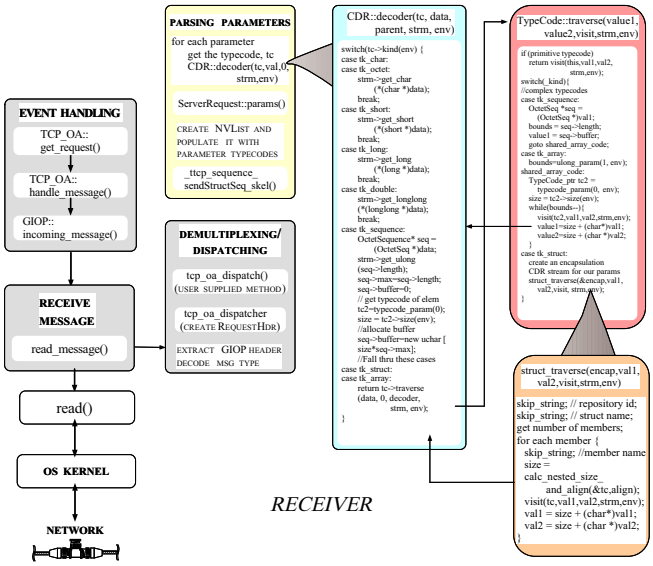


Figure B.7: Receiver-side Datapath for the Original SunSoft IIOP Implementation

Appendix C

IDL Definition for Param_Test Example

This section provides an OMG IDL description of an interface and its operations. In this paper, we use these operations and its parameters to study the cost of marshaling and the size of the generated stubs and skeletons.

```
interface Param_Test
{
  // Primitive types.
  short test_short
    (in short s1,
     inout short s2,
     out short s3);

  // Strings unbounded.
  string test_unbounded_string
    (in string s1,
     inout string s2,
     out string s3);

  // Structures (fixed size).
  struct Fixed_Struct
  {
    long l;
    char c;
    short s;
    octet o;
    float f;
    boolean b;
    double d;
  };
```

```
Fixed_Struct test_fixed_struct
  (in Fixed_Struct s1,
   inout Fixed_Struct s2,
   out Fixed_Struct s3);

// Sequences and typedefs.
typedef sequence<string> StrSeq;

StrSeq test_strseq
  (in StrSeq s1,
   inout StrSeq s2,
   out StrSeq s3);

typedef string DUMMY;
// variable structures
struct Var_Struct
{
  DUMMY dummy1;
  DUMMY dummy2;
  StrSeq seq;
};

Var_Struct test_var_struct
 (in Var_Struct s1,
  inout Var_Struct s2,
  out Var_Struct s3);

//
// Nested structs (We reuse the var_struct
// definition above to make a very
// complicated nested structure).
//

struct Nested_Struct
{
  Var_Struct vs;
};

Nested_Struct test_nested_struct
 (in Nested_Struct s1,
  inout Nested_Struct s2,
  out Nested_Struct s3);

//
// sequences of structs
//
```

```
  typedef sequence<Var_Struct> StructSeq;

  StructSeq test_struct_sequence
   (in StructSeq s1,
    inout StructSeq s2,
    out StructSeq s3);
};
```

Appendix D

Stubs and Skeletons

This section illustrates a stub and skeleton generated by TAO's IDL compiler for the `test_short` operation described in Appendix C.[1]

D.1 Unoptimized interpreted stub

The `do_call` method shown below is the interface to the interpretive marshaling engine. `do_call` takes a variable number of parameters. The number of parameters is determined by the `TAO_Call_Data` argument. This supplies information such as the operation name ("test_short"), whether it is oneway or two-way, and a pointer to a table of parameters (`TAO_Param_Data`). The `TAO_Param_Data` data structure provides the `TypeCode` and parameter type for each parameter of the operation. `TypeCode`s are CORBA pseudo-objects that describe the format and layout of primitive and constructed IDL data types.

```
CORBA::Short Param_Test::test_short
  (CORBA::Short s1,
   CORBA::Short &s2,
   CORBA::Short_out s3,
   CORBA::Environment &env)
{
   static const TAO_Param_Data
   Param_Test_test_short_paramdata [] =
   {
     {CORBA::_tc_short, PARAM_RETURN, 0},
     {CORBA::_tc_short, PARAM_IN, 0},
     {CORBA::_tc_short, PARAM_INOUT, 0},
     {CORBA::_tc_short, PARAM_OUT, 0}
   };

   static const TAO_Call_Data
```

[1] Some exception handling and error checking has been omitted to reduce space.

```
   Param_Test_test_short_calldata =
   {"test_short", 1, 4,
    Param_Test_test_short_paramdata,
    0, 0};

  CORBA::Short retval;
  STUB_Object *istub;

  this->QueryInterface (IID_STUB_Object,
                        (void **) &istub);
  // QueryInterface incremented refcount.
  this->Release ();
  istub->do_call
    (env,
    &Param_Test_test_short_calldata,
    &retval, &s1, &s2, &s3);
  return retval;
}
```

This stub first obtains the underlying stub object and then invokes the `do_call` method on it. The `do_call` internally creates the IIOP `Request` message and marshals all the *in* and *inout* parameters. It then invokes the remote procedure call. It blocks for the incoming reply if the IDL operation is two-way; otherwise it returns immediately. The return value, *inout*, and *out* parameters are demarshaled from the incoming IIOP `Reply` message.

D.2 Unoptimized skeleton

For the `test_short_skel` skeleton shown below, the skeleton first creates a `NVList`. An `NVList` is a CORBA pseudo object that holds a list of parameters. The parameter types, their `TypeCodes`, and memory to store their values are inserted into the `NVList`. The *in* and *inout* parameters are demarshaled via a call to the `params` method on `CORBA::ServerRequest`. The skeleton then makes the upcall on the target object passing the appropriate parameters to it.

```
void POA_Param_Test::test_short_skel
  (CORBA::ServerRequest &_tao_server_request,
   void *_tao_object_reference,
   void *context,
   CORBA::Environment &_tao_environment)
{
  CORBA::NVList_ptr nvlist;
  POA_Param_Test_ptr impl =
   (POA_Param_Test_ptr) _tao_object_reference;
```

```
  CORBA::Any *result;
  CORBA::Short *retval = new CORBA::Short;
  // Create an NV list and populate
  // it with typecodes.
  _tao_server_request.orb ()
    ->create_list (3, nvlist);

  // Add each argument according to the in,
  // out, inout semantics
  CORBA::Short s1;
  (void) nvlist->add_item
    ("s1", CORBA::ARG_IN,
     _tao_environment)->value ()
    ->replace (CORBA::_tc_short, &s1,
               0, _tao_environment);
  CORBA::Short *s2 = new CORBA::Short;
  (void) nvlist->add_item
    ("s2", CORBA::ARG_INOUT,
     _tao_environment)->value ()
    ->replace (CORBA::_tc_short,
               s2, 1, _tao_environment);
  CORBA::Short *s3 =
    new CORBA::Short;
  (void) nvlist->add_item
    ("s3",
     CORBA::ARG_OUT,
     _tao_environment)->value ()
    ->replace (CORBA::_tc_short,
               s3, 1, _tao_environment);
  // Parse the arguments.
  _tao_server_request.params
    (nvlist, _tao_environment);
  *retval = impl->test_short
    (s1, *s2, *s3, _tao_environment);

  // Store the result
  result = new CORBA::Any
   (CORBA::_tc_short, retval, 1);
  // Save the Any into the server request.
  _tao_server_request.result
    (result, _tao_environment);
}
```

D.3 Compiled stubs and skeletons

This section illustrates the hand-crafted compiled stub and skeleton for the `test_short` operation.

```
CORBA::Short Param_Test::test_short
  (CORBA::Short s1,
   CORBA::Short &s2,
   CORBA::Short_out s3,
   CORBA::Environment &env)
{
  CORBA::Short retval = 0;
  IIOP_Object *istub;

  this->QueryInterface (IID_IIOP_Object,
                        (void **) &istub);
  // QueryInterface incremented refcount.
  this->Release ();

  // Set up a GIOP/IIOP message.
  TAO_GIOP_Invocation call
    (istub, ACE_OS::strdup ("test_short"), 1);
  env.clear ();
  // Setup a IIOP Request message.
  call.start (env);

  // Get the marshal stream.
  CDR &stream = call.stream ();

  // Insert parameters.
  stream << s1; stream << s2;

  // Now invoke the request.
  TAO_GIOP_ReplyStatusType status;
  CORBA::ExceptionList exceptions;

  exceptions.length = exceptions.maximum = 0;
  exceptions.buffer = (CORBA::TypeCode_ptr *) 0;

  // Send the request.
  status = call.invoke (exceptions, env);

  // Retrieve the parameter values.
  if (status == TAO_GIOP_NO_EXCEPTION)
    stream >> retval; stream >> s2; stream >> s3;
  return retval;
```

```
}
```

The skeleton is similar and is shown below.

```
void POA_Param_Test::test_short_skel
  (CORBA::ServerRequest &_tao_server_request,
   void *_tao_object_reference,
   void * context,
   CORBA::Environment &_tao_environment)
{
  POA_Param_Test_ptr impl
    = (POA_Param_Test_ptr) _tao_object_reference;
  CORBA::Short retval, s1, s2, s3;

  // Get the incoming CDR stream.
  CDR &instream = _tao_server_request.incoming ();

  // Retrieve parameters.
  instream >> s1; instream >> s2;

  // Make upcall.
  retval = impl->test_short
    (s1, s2, s3, _tao_environment);

  // Get the outgoing CDR stream.
  CDR &outstream = _tao_server_request.outgoing ();

  // Create a IIOP Reply message.
  _tao_server_request.init_reply (_tao_environment);

  // Marshal outgoing parameters.
  outstream << retval;
  outstream << s2; outstream << s3;
}
```

D.4 Optimized stub

```
// Call_Data and Param_Data tables are the same.
CORBA::Short  Param_Test::test_short
  (CORBA::Short s1,
   CORBA::Short &s2,
   CORBA::Short_out s3,
   CORBA::Environment &env)
{
  STUB_Object *istub = this->stubobj (env);
```

```
  if (istub) {
    CORBA::Short retval;
    istub->do_call
      (env,
      &Param_Test_test_short_calldata,
      &retval, &s1, &s2, &s3);
    return retval;
  }
  return 0;
```

D.5 Optimized skeletons

This section illustrates the optimized skeletons that TAO's IDL compiler generates. The skeleton for `test_short_skel` is shown.

```
void POA_Param_Test::test_short_skel
   (CORBA::ServerRequest &_tao_server_request,
    void *_tao_object_reference,
    void *context,
    CORBA::Environment &_tao_environment)
{
   static const TAO_Param_Data_Skel
   Param_Test_test_short_paramdata [] =
   {
     {CORBA::_tc_short, 0, 0},
     {CORBA::_tc_short, CORBA::ARG_IN, 0},
     {CORBA::_tc_short, CORBA::ARG_INOUT, 0},
     {CORBA::_tc_short, CORBA::ARG_OUT, 0}
   };

   static const TAO_Call_Data_Skel
   Param_Test_test_short_calldata =
   {"test_short",
     1,
     4,
     Param_Test_test_short_paramdata};

  POA_Param_Test_ptr impl
    = (POA_Param_Test_ptr) _tao_object_reference;
  CORBA::Short retval, s1, s2, s3;

  // Demarshal parameters.
  _tao_server_request.demarshal
     (_tao_environment,
      &Param_Test_test_short_calldata,
      &retval, &s1, &s2, &s3);
```

```
  // Make upcall.
  retval = impl->test_short
    (s1, s2, s3, _tao_environment);
  // Marshal outgoing parameters.
  _tao_server_request.marshal
    (_tao_environment,
     &Param_Test_test_short_calldata,
     &retval, &s1, &s2, &s3);
}
```

Appendix E

Comparison of Performance and Code Size for Windows NT and Linux

This section provides the cost of marshaling and the size of the generated stubs and skeletons for the Windows NT and Linux platforms.

Table E.1: Sizes of Overloaded Operators for Compiled Stubs/Skeletons on PC running Windows NT

Operator	Size
operator<< (char *)	111
operator>> (char *)	95
operator<< (fixed_struct)	207
operator>> (fixed_struct)	159
operator<< (strseq)	111
operator>> (strseq)	221
operator<< (var_struct)	63
operator>> (var_struct)	95
operator<< (nested_struct)	31
operator>> (nested_struct)	31
operator<< (struct_seq)	239
operator>> (struct_seq)	287

Table E.2: Stub Sizes on PC running Windows NT

Stub name	**Interpreted size**			**Compiled size**		
	stub	**table**	**total**	**stub**	**helper**	total
test_short	205	88	293	570	0	570
test_ubstring	205	88	293	586	206	792
test_fixed_struct	205	88	293	570	366	936
test_strseq	269	88	357	664	332	996
test_var_struct	333	88	421	812	158	970
test_nested_struct	333	88	421	702	62	764
test_struct_seq	205	88	293	670	526	1,196

Table E.3: Skeleton sizes on PC running Windows NT

Stub name	**Interpreted size**			**Compiled size**		
	skel	**table**	**total**	**skel**	**helper**	total
test_short_skel	253	88	341	143	0	143
test_ubstring_skel	333	88	421	239	206	445
test_fixed_struct_skel	285	88	373	191	366	557
test_strseq_skel	468	88	556	358	332	690
test_var_struct_skel	856	88	944	794	158	952
test_nested_struct_skel	867	88	955	726	62	788
test_struct_seq_skel	615	88	703	616	526	1,142

Table E.4: Sizes of Overloaded Operators for Compiled Stubs/Skeletons on PC running Linux

Operator	**Size**
operator<< (char *)	92
operator>> (char *)	92
operator<< (fixed_struct)	200
operator>> (fixed_struct)	140
operator<< (strseq)	284
operator>> (strseq)	344
operator<< (var_struct)	192
operator>> (var_struct)	216
operator<< (nested_struct)	24
operator>> (nested_struct)	24
operator<< (struct_seq)	168
operator>> (struct_seq)	244

Table E.5: Stub Sizes on PC running Linux

Stub name	**Interpreted size**			**Compiled size**		
	stub	**table**	**total**	**stub**	**helper**	total
test_short	104	88	192	452	0	452
test_ubstring	116	88	204	492	184	676
test_fixed_struct	108	88	196	452	340	792
test_strseq	212	88	300	576	628	1,204
test_var_struct	240	88	328	608	408	1,016
test_nested_struct	240	88	328	588	48	636
test_struct_seq	212	88	300	576	412	988

Table E.6: Skeleton Sizes on PC running Linux

Stub name	**Interpreted size**			**Compiled size**		
	skel	**table**	**total**	**skel**	**helper**	total
test_short_skel	136	88	224	180	0	180
test_ubstring_skel	228	88	316	276	184	460
test_fixed_struct_skel	132	88	220	192	340	532
test_strseq_skel	308	88	396	368	628	996
test_var_struct_skel	544	88	632	596	408	1,004
test_nested_struct_skel	544	88	632	596	48	644
test_struct_seq_skel	308	88	396	368	412	780

References

[1] M. Abbott and L. Peterson. Increasing Network Throughput by Integrating Protocol Layers. *ACM Transactions on Networking*, 1(5), October 1993.

[2] Mary L. Bailey, Burra Gopal, Prasenjit Sarkar, Michael A. Pagels, and Larry L. Peterson. Pathfinder: A pattern-based packet classifier. In *Proceedings of the* 1^{st} *Symposium on Operating System Design and Implementation*. USENIX Association, November 1994.

[3] G.J Blaine, M.E. Boyd, and S.M. Crider. Project Spectrum: Scalable Bandwidth for the BJC Health System. *HIMSS, Health Care Communications*, pages 71–81, 1994.

[4] Torsten Braun and Christophe Diot. Protocol Implementation Using Integrated Layer Processnig. In *Proceedings of the Symposium on Communications Architectures and Protocols (SIGCOMM)*. ACM, September 1995.

[5] Isabelle Chrisment. Impact of ALF on Communication Subsystems Design and Performance. In *First International Workshop on High Performance Protocol Architectures, HIPPARCH '94*, Sophia Antipolis, France, December 1994. INRIA France.

[6] David D. Clark, Van Jacobson, John Romkey, and Howard Salwen. An Analysis of TCP Processing Overhead. *IEEE Communications Magazine*, 27(6):23–29, June 1989.

[7] David D. Clark and David L. Tennenhouse. Architectural Considerations for a New Generation of Protocols. In *Proceedings of the Symposium on Communications Architectures and Protocols (SIGCOMM)*, pages 200–208, Philadelphia, PA, September 1990. ACM.

[8] Z. Deng and J. W.-S. Liu. Scheduling Real-Time Applications in an Open Environment. In *Proceedings of the 18th IEEE Real-Time Systems Symposium*. IEEE Computer Society Press, December 1997.

[9] Sudheer Dharnikota, Kurt Maly, and C. M. Overstreet. Performance Evaluation of TCP(UDP)/IP over ATM networks. Department of Computer Science, Technical Report CSTR_94_23, Old Dominion University, September 1994.

[10] Zubin D. Dittia, Guru M. Parulkar, and Jerome R. Cox, Jr. The APIC Approach to High Performance Network Interface Design: Protected DMA and Other Techniques. In *Proceedings of INFOCOM '97*, pages 179–187, Kobe, Japan, April 1997. IEEE.

[11] Minh DoVan, Louis Humphrey, Geri Cox, and Carl Ravin. Initial Experience with Asynchronous Transfer Mode for Use in a Medical Imaging Network. *Journal of Digital Imaging*, 8(1):43–48, February 1995.

[12] Peter Druschel and Larry L. Peterson. Fbufs: A High-Bandwidth Cross-Domain Transfer Facility. In *Proceedings of the* 14^{th} *Symposium on Operating System Principles (SOSP)*, December 1993.

[13] Eric Eide, Kevin Frei, Bryan Ford, Jay Lepreau, and Gary Lindstrom. Flick: A Flexible, Optimizing IDL Compiler. In *Proceedings of ACM SIGPLAN '97 Conference on Programming Language Design and Implementation (PLDI)*, Las Vegas, NV, June 1997. ACM.

[14] Dawson R. Engler and M. Frans Kaashoek. DPF: Fast, Flexible Message Demultiplexing using Dynamic Code Generation. In *Proceedings of ACM SIGCOMM '96 Conference in Computer Communication Review*, pages 53–59, Stanford University, California, USA, August 1996. ACM Press.

[15] Victor Fay-Wolfe, John K. Black, Bhavanai Thuraisingham, and Peter Krupp. Real-time Method Invocations in Distributed Environments. Technical Report 95-244, University of Rhode Island, Department of Computer Science and Statistics, 1995.

[16] David C. Feldmeier. Multiplexing Issues in Communications System Design. In *Proceedings of the Symposium on Communications Architectures and Protocols (SIGCOMM)*, pages 209–219, Philadelphia, PA, September 1990. ACM.

[17] G. Forman and J. Zahorhan. The Challenges of Mobile Computing. *IEEE Computer*, 27(4):38–47, April 1994.

[18] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, MA, 1995.

[19] Atanu Ghosh, Jon Crowcroft, Michael Fry, and Mark Handley. Integrated Layer Video Decoding and Application Layer Framed Secure Login: General Lessons from Two or Three Very Different Applications. In *First International Workshop on High Performance Protocol Architectures, HIPPARCH '94*, Sophia Antipolis, France, December 1994. INRIA France.

[20] Aniruddha Gokhale and Douglas C. Schmidt. Measuring the Performance of Communication Middleware on High-Speed Networks. In *Proceedings of SIGCOMM '96*, pages 306–317, Stanford, CA, August 1996. ACM.

[21] Aniruddha Gokhale and Douglas C. Schmidt. The Performance of the CORBA Dynamic Invocation Interface and Dynamic Skeleton Interface over High-Speed ATM Networks. In *Proceedings of GLOBECOM '96*, pages 50–56, London, England, November 1996. IEEE.

[22] Aniruddha Gokhale and Douglas C. Schmidt. Design Principles and Optimizations for High-performance ORBs. In 12^{th} OOPSLA Conference, poster session, Atlanta, Georgia, October 1997. ACM.

[23] Aniruddha Gokhale and Douglas C. Schmidt. Evaluating Latency and Scalability of CORBA Over High-Speed ATM Networks. In *Proceedings of the International Conference on Distributed Computing Systems*, Baltimore, Maryland, May 1997. IEEE.

[24] Aniruddha Gokhale and Douglas C. Schmidt. Evaluating the Performance of Demultiplexing Strategies for Real-time CORBA. In *Proceedings of GLOBECOM '97*, Phoenix, AZ, November 1997. IEEE.

[25] Aniruddha Gokhale and Douglas C. Schmidt. Optimizing a CORBA IIOP Protocol Engine for Minimal Footprint Multimedia Systems. *Journal on Selected Areas in Communications special issue on Service Enabling Platforms for Networked Multimedia Systems*, 17(9), September 1999.

[26] Aniruddha Gokhale, Douglas C. Schmidt, and Stan Moyer. Tools for Automating the Migration from DCE to CORBA. In *Proceedings of ISS 97: World Telecommunications Congress*, Toronto, Canada, September 1997. IEEE Communications Society.

[27] R. Gopalakrishnan and G. Parulkar. A Real-time Upcall Facility for Protocol Processing with QoS Guarantees. In 15^{th} *Symposium on Operating System Principles (poster session)*, Copper Mountain Resort, Boulder, CO, December 1995. ACM.

[28] R. Gopalakrishnan and G. Parulkar. Bringing Real-time Scheduling Theory and Practice Closer for Multimedia Computing. In *SIGMETRICS Conference*, Philadelphia, PA, May 1996. ACM.

[29] R. Gopalakrishnan and Guru M. Parulkar. Real-time Upcalls: A Mechanism to Provide Real-time Processing guarantees. Technical Report 95-06, Dept. of Computer Science, Washington University in St. Louis, 1995.

[30] Tim Harrison, Douglas C. Schmidt, Aniruddha Gokhale, and Guru Parulkar. Operating System Support for High-Performance, Real-time CORBA. In *Proceedings of the* 5^{th} *International Workshop on Object-Orientation in Operating Systems*. IEEE, October 1996.

[31] Timothy H. Harrison, David L. Levine, and Douglas C. Schmidt. The Design and Performance of a Real-time CORBA Event Service. In *Proceedings of OOPSLA '97*, pages 184–199, Atlanta, GA, October 1997. ACM.

[32] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kauffman Publishers, San Francisco, California, 1990.

[33] Phillip Hoschka and Christian Huitema. Automatic Generation of Optimized Code for Marshalling Routines. In *IFIP Conference of Upper Layer Protocols, Architectures and Applications ULPAA'94*, Barcelona, Spain, 1994. IFIP.

[34] Norman C. Hutchinson and Larry L. Peterson. The *x*-kernel: An Architecture for Implementing Network Protocols. *IEEE Transactions on Software Engineering*, 17(1):64–76, January 1991.

[35] PureAtria Software Inc. *Quantify User's Guide*. PureAtria Software Inc., 1996.

[36] International Organization for Standardization. *Information processing systems - Open Systems Interconnection - Specification of Basic Encoding Rules for Abstract Syntax Notation One (ASN.1)*, May 1987.

[37] Mahesh Jayaram and Ron Cytron. Efficient Demultiplexing of Network Packets by Automatic Parsing. In *Proceedings of the Workshop on Compiler Support for System Software (WCSSS 96)*, University of Arizona, Tucson, AZ, February 1996.

[38] Ralph Johnson. Frameworks = Patterns + Components. *Communications of the ACM*, 40(10):39–42, October 1997.

[39] Jonathan Kay and Joseph Pasquale. The Importance of Non-Data Touching Processing Overheads in TCP/IP. In *Proceedings of SIGCOMM '93*, pages 259–269, San Francisco, CA, September 1993. ACM.

[40] Khanna, S., *et al.* Realtime Scheduling in SunOS 5.0. In *Proceedings of the USENIX Winter Conference*, pages 375–390. USENIX Association, 1992.

[41] Fred Kuhns, Douglas C. Schmidt, and David L. Levine. The Design and Performance of a Real-time I/O Subsystem. In *Proceedings of the* 5^{th} *IEEE Real-Time Technology and Applications Symposium*, pages 154–163, Vancouver, British Columbia, Canada, June 1999. IEEE.

[42] Sean Landis and Silvano Maffeis. Building Reliable Distributed Systems with CORBA. *Theory and Practice of Object Systems*, 3(1):31–43, 1997.

[43] S. J. Leffler, M. K. McKusick, M. J. Karels, and J. S. Quarterman. *The Design and Implementation of the 4.3BSD UNIX Operating System.* Addison-Wesley, Reading, MA, 1989.

[44] Steven McCanne and Van Jacobson. The BSD Packet Filter: A New Architecture for User-level Packet Capture. In *Proceedings of the Winter USENIX Conference*, pages 259–270, San Diego, CA, January 1993.

[45] K. Modeklev, E. Klovning, and O. Kure. TCP/IP Behavior in a High-Speed Local ATM Network Environment. In *Proceedings of the* 19^{th} *Conference on Local Computer Networks*, pages 176–185, Minneapolis, MN, October 1994. IEEE.

[46] Jeffrey C. Mogul, Richard F. Rashid, and Michal J. Accetta. The Packet Filter: an Efficient Mechanism for User-level Network Code. In *Proceedings of the* 11^{th} *Symposium on Operating System Principles (SOSP)*, November 1987.

[47] David Mosberger, Larry L. Peterson, Patrick G. Bridges, and Sean O'Malley. Analysis of Techniques to Improve Protocol Processing Latency. In *Proceedings of SIGCOMM '96*, pages 73–84, Stanford, CA, August 1996. ACM.

[48] Object Management Group. *CORBAServices: Common Object Services Specification, Revised Edition*, 95-3-31 edition, March 1995.

[49] Object Management Group. *Minimum CORBA - Request for Proposal*, OMG Document orbos/97-06-14 edition, June 1997.

[50] Object Management Group. *Realtime CORBA 1.0 Request for Proposals*, OMG Document orbos/97-09-31 edition, September 1997.

[51] Object Management Group. *The Common Object Request Broker: Architecture and Specification*, 2.2 edition, February 1998.

[52] Sean W. O'Malley, Todd A. Proebsting, and Allen B. Montz. USC: A Universal Stub Compiler. In *Proceedings of the Symposium on Communications Architectures and Protocols (SIGCOMM)*, London, UK, August 1994.

[53] Christos Papadopoulos and Gurudatta Parulkar. Experimental Evaluation of SUNOS IPC and TCP/IP Protocol Implementation. *IEEE/ACM Transactions on Networking*, 1(2):199–216, April 1993.

[54] Graham Parrington. A Stub Generation System for C++. *Computing Systems*, 8(2):135–170, Spring 1995.

[55] Guru Parulkar, Douglas C. Schmidt, and Jonathan S. Turner. a$^{\mathrm{I}}$t$^{\mathrm{P}}$m: a Strategy for Integrating IP with ATM. In *Proceedings of the Symposium on Communications Architectures and Protocols (SIGCOMM)*. ACM, September 1995.

[56] Joseph C. Pasquale, Eric W. Anderson, Kevin R. Fall, and Jonathan S. Kay. High-performance I/O and Networking Software in Sequoia 2000. *Digital Technical Journal*, 7(3), 1995.

[57] Irfan Pyarali, Timothy H. Harrison, and Douglas C. Schmidt. Design and Performance of an Object-Oriented Framework for High-Performance Electronic Medical Imaging. In *Proceedings of the* 2^{nd} *Conference on Object-Oriented Technologies and Systems*, Toronto, Canada, June 1996. USENIX.

[58] Irfan Pyarali, Timothy H. Harrison, and Douglas C. Schmidt. Design and Performance of an Object-Oriented Framework for High-Performance Electronic Medical Imaging. *USENIX Computing Systems*, 9(4), November/December 1996.

[59] Sanjay Radia, Graham Hamilton, Peter Kessler, and Michael Powell. The Spring Object Model. In *Proceedings of the Conference on Object-Oriented Technologies*, Monterey, CA, June 1995. USENIX.

[60] Ragunathan Rajkumar, Lui Sha, and John P. Lehoczky. Real-Time Synchronization Protocols for Multiprocessors. In *Proceedings of the Real-Time Systems Symposium*, pages 259–269, Huntsville, Alabama, December 1988.

[61] Antony Richards, Ranil De Silva, Anne Fladenmuller, Aruna Seneviratne, and Michael Fry. The Application of ILP/ALF to Configurable Protocols. In *First International Workshop on High Performance Protocol Architectures, HIPPARCH '94*, Sophia Antipolis, France, December 1994. INRIA France.

[62] Dennis Ritchie. A Stream Input–Output System. *AT&T Bell Labs Technical Journal*, 63(8):311–324, October 1984.

[63] Douglas C. Schmidt. GPERF: A Perfect Hash Function Generator. In *Proceedings of the* 2^{nd} *C++ Conference*, pages 87–102, San Francisco, California, April 1990. USENIX.

[64] Douglas C. Schmidt. ACE: an Object-Oriented Framework for Developing Distributed Applications. In *Proceedings of the* 6^{th} *USENIX C++ Technical Conference*, Cambridge, Massachusetts, April 1994. USENIX Association.

[65] Douglas C. Schmidt. A Family of Design Patterns for Application-level Gateways. *The Theory and Practice of Object Systems (Special Issue on Patterns and Pattern Languages)*, 2(1), 1996.

[66] Douglas C. Schmidt and Chris Cleeland. Applying Patterns to Develop Extensible ORB Middleware. *IEEE Communications Magazine*, 37(4), April 1999.

[67] Douglas C. Schmidt, Aniruddha Gokhale, Tim Harrison, and Guru Parulkar. A High-Performance Endsystem Architecture for Real-time CORBA. *IEEE Communications Magazine*, 14(2), February 1997.

[68] Douglas C. Schmidt, Timothy H. Harrison, and Ehab Al-Shaer. Object-Oriented Components for High-speed Network Programming. In *Proceedings of the* 1^{st} *Conference on Object-Oriented Technologies and Systems*, Monterey, CA, June 1995. USENIX.

[69] Douglas C. Schmidt, David L. Levine, and Sumedh Mungee. The Design and Performance of Real-Time Object Request Brokers. *Computer Communications*, 21(4):294–324, April 1998.

[70] Douglas C. Schmidt, Sumedh Mungee, Sergio Flores-Gaitan, and Aniruddha Gokhale. Alleviating Priority Inversion and Non-determinism in Real-time CORBA ORB Core Architectures. In *Proceedings of the* 4^{th} *IEEE Real-Time Technology and Applications Symposium*, Denver, CO, June 1998. IEEE.

[71] Sun Microsystems. XDR: External Data Representation Standard. *Network Information Center RFC 1014*, June 1987.

[72] Sun Microsystems. *Open Network Computing: Transport Independent RPC*, June 1995.

[73] David L. Tennenhouse. Layered Multiplexing Considered Harmful. In *Proceedings of the* 1^{st} *International Workshop on High-Speed Networks*, May 1989.

[74] USNA. *TTCP: a test of TCP and UDP Performance*, Dec 1984.

[75] George Varghese. Algorithmic Techniques for Efficient Protocol Implementations . In *SIGCOMM '96 Tutorial*, Stanford, CA, August 1996. ACM.

[76] Steve Vinoski. CORBA: Integrating Diverse Applications Within Distributed Heterogeneous Environments. *IEEE Communications Magazine*, 14(2), February 1997.

[77] M. Yuhara, B. Bershad, C. Maeda, and E. Moss. Efficient Packet Demultiplexing for Multiple Endpoints and Large Messages. In *Proceedings of the Winter Usenix Conference*, January 1994.

Vita

Aniruddha Suresh Gokhale

Washington University
Department of Computer Science
Campus Box 1045
One Brookings Drive
St. Louis, MO 63130-4899, USA

Phone: (314) 935-4215
FAX: (314) 935-7302
Email: gokhale@cs.wustl.edu
URL: http://www.cs.wustl.edu/~gokhale/

Research Interests

- Distributed Object Computing
- Performance evaluation and optimizations of object-oriented middleware such as CORBA, DCOM, and Java RMI
- High-performance, real-time communication systems and protocols
- Object-oriented software engineering with design patterns and frameworks

Educational Background

- **Ph.D. Computer Science**, expected Spring 1998, Washington University, St. Louis, Missouri, USA.
  Dissertation: "Design Principles and Optimizations for High-performance, Real-time CORBA Implementations."
  Advisor: Dr. Douglas C. Schmidt

- **M.S. Computer Science**, August 1992, Arizona State University, Tempe, Arizona, USA.
  Thesis: "Automatic Test Suite Generation for Protocol Testing."
  Advisor: Dr. Arunabha Sen

- **B.E. Computer Engineering**, June 1989, University of Poona, Pune, Maharashtra, India.
  Senior level project: "Lisp Interpreter in C using Recursive Descent Parsing."
  Advisors: Prof. Shashikant Bhandari and Prof. Shirish Joshi

Experience

Research Experience

- **8/95 – present: Graduate Research Assistant**, Washington University, St. Louis, MO.
  Worked with Dr. Douglas C. Schmidt on research involving measuring the performance of CORBA implementations and improving their efficiency and predictability via systematic optimizations.
- **8/93 – 8/95: Graduate Research Assistant**, Washington University, St. Louis, MO.
  Worked with Dr. Ron Cytron and Dr. George Varghese on research involving use of compiler code generation and optimization techniques for rapid prototyping of efficient protocol implementations.
- **8/91 – 7/92: Graduate Research Assistant**, Arizona State University, Tempe, AZ.
  Worked with Dr. Arunabha Sen on research involving Protocol Conformance testing. This work involved automatically generating tests suites from Estelle specifications of protocols to test for conformance to their specifications.

Teaching Experience

- Delivered guest lectures on CORBA architecture and performance of CORBA implementations for graduate level courses taught by Dr. Douglas C. Schmidt at Washington University, St. Louis, MO.
- **8/92 – 5/93: Graduate Teaching Assistant**, Washington University, St. Louis, MO.
  Assisted in grading and consulting for a senior level course on Operating Systems for the Fall 1992 and Spring 1993 semesters.
- **1/92 – 5/92: Graduate Teaching Assistant**, Arizona State University, Tempe, AZ.
  Assisted in grading and consulting for a junior level course on Data Structures.
- **8/89 – 7/90: Lecturer**, Maharashtra Institute of Technology, Pune, India.
  Responsibilities included:
  - Teaching a senior level course in Introduction to Computer Networks.
  - Teaching a junior level course in Digital Logic Design
  - Conducting Software Laboratory sessions
  - Overseeing the administration and maintenance of the PC laboratory

Software Development Experience

- **8/95 – present: TAO High Performance, Real-time ORB Implementation**, Washington University, St. Louis, MO.
  As part of my Ph.D. dissertation, I have implemented major portions The ACE ORB (TAO) (`http://www.cs.wustl.edu/~gokhale/research.html`). TAO is a high-performance, real-time implementation of CORBA. TAO's implementation is heavily influenced by the optimizations and components developed for my Ph.D. research, including:
  - High-performance IIOP protocol marshaling engine
  - Real-time Object Adapter demultiplexing engine
  - OMG IDL compiler implementation
  - Performance optimizations and measurements
- **8/96 – 12/96: Bellcore's DCE to CORBA Migration Tool**, Washington University, St. Louis, MO.
  Worked with Dr. Douglas Schmidt and Stan Moyer of Bellcore, USA in the production of a OSF DCE RPC to OMG CORBA migration tool.
- **5/93 – 12/93, 5/94 – 7/95: DICOM Protocol Implementation**, Electronic Radiology Laboratory, Washington University Medical School, St. Louis, MO.
  Worked in a team headed by Dr. G. James Blaine on the implementation of the Digital Imaging and Communications in Medicine (DICOM) v3.0 protocol.
- **5/91 – 8/92: Automatic Test Suite Generator**, Arizona State University, Tempe, AZ.
  As part of my Master's thesis, I developed a tool in C++ that modified NIST's PET tool for the Estelle formal description technique and generated test suites to test protocols for conformance testing. This work was done with my advisor Dr. Arunabha Sen.

Other work experience

- **8/90 – 12/91 Student Worker**, Department of Sociology, Arizona State University, Tempe, AZ.
  Job responsibilities included assisting in office work that included printing, collating, and stapling handouts and examinations for Sociology classes, and sometimes proctoring examinations.

Publications

Refereed Journal Publications

1. Aniruddha Gokhale and Douglas C. Schmidt, "Measuring and Optimizing CORBA Latency and Scalability Over High-speed Networks," IEEE Computer Society's Journal of Transactions on Computers. Publication date – April 1998.

2. Douglas C. Schmidt, Aniruddha Gokhale, Tim Harrison, and Guru Parulkar, "A High-performance Endsystem Architecture for Real-time CORBA," *IEEE Communications Magazine*, Vol. 14, No. 2, February, 1997.

Refereed Conference Publications

1. Douglas C. Schmidt, Sumedh Mungee, and Aniruddha Gokhale, "Alleviating Priority Inversion and Non-determinism in Real-time CORBA ORB Architectures," IEEE Real-Time Applications Symposium (RTAS) 1998, Denver, CO, June 1998.

2. Aniruddha Gokhale and Douglas C. Schmidt, "Optimizing the Performance of the CORBA Internet Inter-ORB Protocol Over ATM," Proceedings of the 31st Hawaii International Conference on System Systems (HICSS), Hawaii, January, 1998. Received **Best Paper Award** in the Software Technology Track (188 submitted, 77 accepted).

3. Aniruddha Gokhale and Douglas C. Schmidt, "Evaluating the Performance of Demultiplexing Strategies for Real-time CORBA," Proceedings of GLOBECOM '97 conference, IEEE, Phoenix, AZ, November, 1997.

4. Aniruddha Gokhale and Douglas C. Schmidt and Stan Moyer, "Tools for Automating the Migration from DCE to CORBA,"Proceedings of ISS 97: World Telecommunications Congress, IEEE Toronto, Canada, September, 1997.

5. Aniruddha Gokhale and Douglas C. Schmidt, "Evaluating Latency and Scalability of CORBA Over High-Speed ATM Networks," Proceedings of the International Conference on Distributed Computing Systems '97, IEEE, Baltimore, Maryland, May 27–30, 1997.

6. Aniruddha Gokhale and Douglas C. Schmidt, "Performance of the CORBA Dynamic Invocation Interface and Internet Inter-ORB Protocol over High-Speed ATM Networks," Proceedings of GLOBECOM '96, IEEE, London England, November, 1996.

7. Aniruddha Gokhale and Douglas C. Schmidt, "Measuring the Performance of Communication Middleware on High-Speed Networks," Proceedings of SIGCOMM '96, ACM, San Francisco, August 28-30th, 1996.

Refereed Workshop Publications

1. Aniruddha Gokhale, Tim Harrison, Douglas C. Schmidt, and Guru Parulkar, "Operating System Support for Real-time CORBA," *Proceedings of the 5th International Workshop on Object-Orientation in Operating Systems*: IWOOOS 1996 workshop, October 27–28, 1996, Seattle, Washington.

Poster Sessions

1. Aniruddha Gokhale and Douglas C. Schmidt, "Design Principles and Optimizations for High Performance ORBs," ACM, *OOPSLA 97*, Poster Session, Oct 1997, Atlanta, GA, USA.

2. Aniruddha Gokhale, "Optimizations for High Performance ORBs," ACM, *OOPSLA 96*, Poster Session, Oct 1996, San Jose, CA, USA.

Technical Reports

1. Aniruddha Gokhale and Douglas C. Schmidt, "Optimizing the Performance of the CORBA Internet Inter-ORB Protocol Over ATM," Washington University, Computer Science technical report #WUCS-97-10.

2. Aniruddha Gokhale, George Varghese. and Ron Cytron, "Design of a Tool for Rapid Prototyping of Protocols," Washington University, Computer Science, Technical report WUCS-95-30.

Submitted for Publication

1. Aniruddha Gokhale and Douglas C. Schmidt, "Optimizing a CORBA IIOP Protocol Engine for Minimal Footprint Multimedia Systems," IEEE Journal on Selected Areas in Communication, Special Issue on Service Enabling Platforms for Networked Multimedia Systems

Presentations

1. Aniruddha Gokhale and Douglas C. Schmidt, "Evaluating the Performance of Demultiplexing Strategies for Real-time CORBA," Proceedings of GLOBECOM '97 conference, IEEE, Phoenix, AZ, November, 1997.

2. Aniruddha Gokhale and Douglas C. Schmidt, "Design Principles and Optimizations for High Performance ORBs," ACM, *OOPSLA 97*, Poster Session, Oct 1997, Atlanta, GA, USA.

3. Aniruddha Gokhale and Douglas C. Schmidt and Stan Moyer, "Tools for Automating the Migration from DCE to CORBA,"Proceedings of ISS 97: World Telecommunications Congress, IEEE, Toronto, Canada, September, 1997.

4. Aniruddha Gokhale and Douglas C. Schmidt, "Evaluating Latency and Scalability of CORBA Over High-Speed ATM Networks," Proceedings of the International Conference on Distributed Computing Systems '97, IEEE, Baltimore, Maryland, May 27–30, 1997.

5. Aniruddha Gokhale and Douglas C. Schmidt, "Performance of the CORBA Dynamic Invocation Interface and Internet Inter-ORB Protocol over High-Speed ATM Networks," Proceedings of GLOBECOM '96, IEEE, London England, November, 1996.

6. Aniruddha Gokhale, "Optimizations for High Performance ORBs," ACM, *OOPSLA 96*, Poster Session, Oct 1996, San Jose, CA, USA.

7. Aniruddha Gokhale, "Optimizations for High Performance ORBs," ACM, *OOPSLA 96*, Doctoral Symposium, Oct 1996, San Jose, CA, USA.

Awards and Honors

- First author of paper receiving the Best Paper Award in the Software Technology Track at the HICSS '98 conference, January 1998, Hawaii, USA (188 submitted, 77 accepted).
- Selected amongst the top six papers submitted to the ICDCS '97 Conference, May 97, Baltimore, MD for further review for the IEEE Computer Society's Journal of Transactions on Computers.
- Selection for ACM OOPSLA '96 Doctoral Symposium, October 1996, San Jose, CA.
- IEEE travel grant award for attending and presenting a paper at IEEE Globecom 96, London, UK.
- Tilak Maharashtra Vidyapeeth, Pune, India award for securing second highest scores in state-wide mathematics examination, Spring 1983.

Professional Activities

- **Professional Society Memberships:** – Student member of IEEE, IEEE Communications Society, and IEEE Computer Society.

Citizenship, Nationality, and Visa Status

- **Citizenship and Nationality** – India
- **Immigration Status in USA** – F-1 Student Visa

References

1. Dr. Douglas C. Schmidt
   Washington University
   Campus Box 1045
   Department of Computer Science
   One Brookings Drive
   St. Louis, MO 63130-4899
   Phone: (314) 935-4215
   FAX: (314) 935-7302
   Email: schmidt@cs.wustl.edu
   URL: http://www.cs.wustl.edu/∼schmidt/

2. Dr. George Varghese
   Washington University
   Campus Box 1045
   Department of Computer Science
   One Brookings Drive
   St. Louis, MO 63130-4899
   Phone: (314) 935-4963
   FAX: (314) 935-7302
   Email: varghese@askew.wustl.edu
   URL: http://dworkin.wustl.edu/∼vargehse/

3. Dr. Ron Cytron
   Washington University
   Campus Box 1045
   Department of Computer Science
   One Brookings Drive
   St. Louis, MO 63130-4899
   Phone: (314) 935-7527
   FAX: (314) 935-7302
   Email: cytron@cs.wustl.edu
   URL: http://www.cs.wustl.edu/∼cytron/

4. Dr. Arunabha Sen
   Arizona State University
   Department of Computer Science and Engineering
   College of Engineering and Applied Sciences
   Box 875406
   Tempe, AZ 85287-5406
   Phone: (602) 965-6153
   FAX: (602) 965-2751

Email: arunabha.sen@asu.edu
URL: http://www.eas.asu.edu:80/∼csedept/people/faculty/sen.html

August 1998