Measuring and Optimizing CORBA Latency and Scalability Over High-speed Networks

Aniruddha S. Gokhale and Douglas C. Schmidt
{gokhale,schmidt}@cs.wustl.edu
Department of Computer Science
Washington University
St. Louis, MO 63130*

This paper has been submitted to IEEE Transaction on Computers. An earlier version of this paper appeared in the Proceedings of the International Conference on Distributed Computing Systems (ICDCS '97), Baltimore, MD, May 27-30, 1997.

Abstract

There is increasing demand to extend object-oriented middleware, such as OMG CORBA, to support applications with stringent quality of service (QoS) requirements. However, conventional CORBA Object Request Broker (ORB) implementations incur high latency and low scalability when used for performance-sensitive applications. These inefficiencies discourage developers from using CORBA for mission/life-critical applications such as real-time avionics, telecom call processing, and medical imaging.

This paper provides two contributions to the research on CORBA performance. First, we systematically analyze the latency and scalability of two widely used CORBA ORBs, VisiBroker and Orbix. These results reveal key sources of overhead in conventional ORBs. Second, we describe techniques used to improve latency and scalability in TAO, which is a high-performance, real-time implementation of CORBA. Although conventional ORBs do not yet provide adequate QoS guarantees to applications, our research results indicate it is possible to implement ORBs that can support high-performance, real-time applications.

Keywords: Distributed object computing, CORBA communication middleware performance, Real-time CORBA.

1 Introduction

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 [1].

Requirements for flexibility and reusability motivate the use of the *Common Object Request Broker Architecture* (CORBA) [2]. CORBA automates common network programming tasks such as object location, object activation, parameter marshaling/demarshaling, framing, and error handling. CORBA also provides the basis for defining higher layer distributed services such as naming, events, replication, transactions, and security [3].

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. 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.

*This work was supported in part by NSF grant NCR-9628218.

Networks like ATM, FDDI, and Fiber Channel now support QoS guarantees for bandwidth, latency, and reliability. However, conventional CORBA ORBs incur significant overhead when used for latency-sensitive applications over high-speed networks. If not corrected, this overhead will force developers to avoid CORBA middleware and continue to use lower-level tools like sockets. Unfortunately, lower-level tools fail to provide other key benefits of CORBA such as robustness, flexibility, and reusability, which are crucial to the success of complex latency-sensitive distributed applications [4]. Therefore, it is imperative to eliminate the sources of CORBA overhead shown in this paper.

Our earlier work [5, 6, 7, 8] has focused on measuring and optimizing the *throughput* of CORBA ORBs. This paper 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.

The research contributions of this paper include the following:

• **CORBA Latency** Section 4 measures the oneway and twoway 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. Section 5.4 describes optimizations we have developed to reduce these common sources of ORB latency.

• **CORBA Scalability** Section 4 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) lack of integration with OS and network features. Section 5.3 describes demultiplexing optimizations we have developed to increase ORB scalability.

The paper is organized as follows: Section 2 outlines the CORBA ORB reference model; Section 3 describes our CORBA/ATM testbed and experimental methods; Section 4 presents the key sources of latency and scalability overhead in conventional CORBA ORBs over ATM networks; Section 5 describes our research on developing optimizations that eliminate the latency and scalability bottlenecks in CORBA ORBs; Section 6 describes related work; and Section 7 presents concluding remarks.

2 Overview of the CORBA ORB Reference Model

CORBA Object Request Brokers (ORBs) allow clients to invoke methods on distributed objects without concern for:

• **Object location** CORBA objects can be located locally or remotely, without affecting their implementation or use;

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

• **OS platform** CORBA runs on many OS platforms, including as 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 over include TCP/IP, IPX/SPX, FDDI, ATM, Fast Ethernet, and embedded system backplanes.

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

The components in the CORBA reference model shown in Figure 1 provide this transparency.

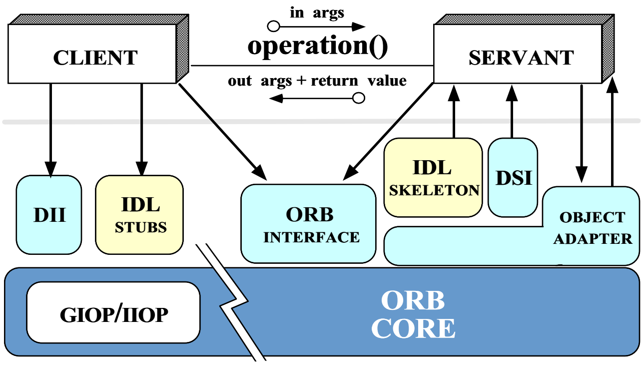

Figure 1: Components in the CORBA Reference Model

These components include the following:

• **Servant** This defines operations that implement an OMG Interface Definition Language (IDL) interface. In languages like C++ and Java that support object-oriented (OO) programming, servants are typically implemented using one or more objects.

• **Client** This is the program entity that invokes an operation on a servant, which can be remote or co-located relative to the client. Ideally, accessing a remote servant should be as simple as calling a method on a local object, *i.e.*, `obj->op(args)`.

Figure 1 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, the ORB Core is responsible for delivering the request to the servant and returning a response, if any, to the client. For servers executing remotely, CORBA 2.0-compliant [2] ORB Cores communicate via the General Inter-ORB Protocol (GIOP) and the Internet Inter-ORB Protocol (IIOP), which 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 convert object references to strings and back, as well as create argument lists for requests made through the Dynamic Invocation Interface (DII) described below.

• **OMG IDL Stubs and Skeletons** IDL stubs and skeletons serve as the "glue" between the client and servants, respectively, and the ORB Core. Stubs provide a *static invocation interface* (SII) that marshals typed 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. The transformation between OMG IDL definitions and an application programming language like C++ or Java is automated by an IDL compiler. IDL compilers eliminate common sources of network programming errors and increase opportunities for automated compiler optimizations [9].

• **Dynamic Invocation Interface (DII)** The DII allows a client to directly access the underlying request transport mechanisms provided by the ORB Core. The DII is useful when an application has no compile-time knowledge of the interface it is accessing. Unlike IDL stubs, which only allow twoway and oneway requests, the DII also allows clients to make non-blocking *deferred synchronous* calls, which separate send and receive operations.

• **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 that the server's implementation uses static skeletons or dynamic skeletons.

• **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. While current CORBA implementations are typically limited to a single Object Adapter per ORB, recent CORBA portability enhancements [10] define the Portable Object Adapter (POA), with support for multiple POAs per ORB.

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[1] 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 primary 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.

3 CORBA/ATM Testbed and Experimental Methods

This section describes our CORBA/ATM testbed and outlines our experimental methods.

3.1 Hardware and Software Platforms

Our experimental ATM/CORBA testbed is depicted in Figure 2. The experiments in this section were conducted using a FORE systems ASX-1000 ATM switch connected to two dual-processor UltraSPARC-2s running SunOS 5.5.1. The ASX-1000 is a 96 Port, OC12 622 Mbs/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 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

[1] An *object reference* is a client-side entity that behaves as a proxy on behalf of the servant residing on the server.

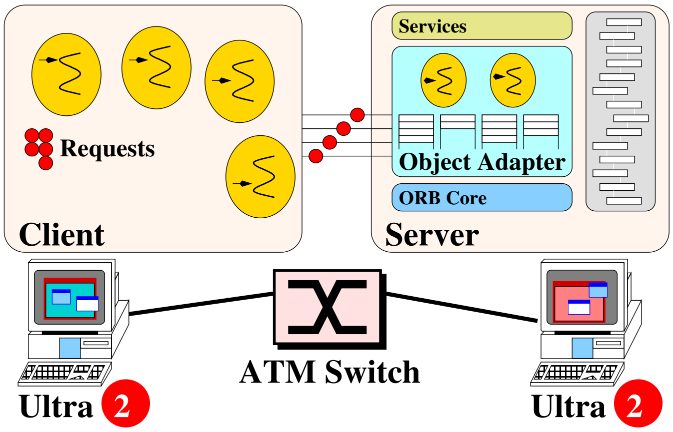

Figure 2: ATM Testbed for ORB Endsystem Performance Experiments

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.

3.2 Traffic Generators

Our earlier studies [5, 6, 7] 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 paper 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 [11]. TTCP is a widely used benchmarking tool to evaluate the performance of TCP/IP and UDP/IP networks. We extended TTCP to handle oneway and twoway operations for Orbix 2.1 and VisiBroker 2.0.

The flow of control is uni-directional in oneway operations and the client can proceed in parallel with the server. CORBA specifies "best-effort" delivery semantics for oneway operations; no application-level acknowledgment is passed back to the requester. Both Orbix and VisiBroker use TCP/IP to transmit oneway operations.

In twoway operations, the flow of control is bi-directional. After each twoway operation is invoked, the client blocks until an acknowledgment is received, *i.e.*, the operation returns. In our experiments, we defined twoway operations to return "`void`," thereby minimizing the size of the acknowledgment from the server.

We measured round-trip latency using the twoway 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 paper:

```
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();
};
```

3.3 TTCP Parameter Settings

Related work [12, 13, 14, 7] on transport protocol performance over ATM demonstrate the performance impact of parameters such as the size of socket queues, data buffers, and number of servants in a server. Therefore, our TTCP benchmarks systematically varied these parameters for each type of data as follows:

- **Socket queue size** The sender and receiver socket queue sizes used were 64 K bytes, which is the maximum on SunOS 5.5. These parameters influence the size of the TCP segment window, which has been shown [14] to significantly affect CORBA-level and TCP-level performance on high-speed networks.

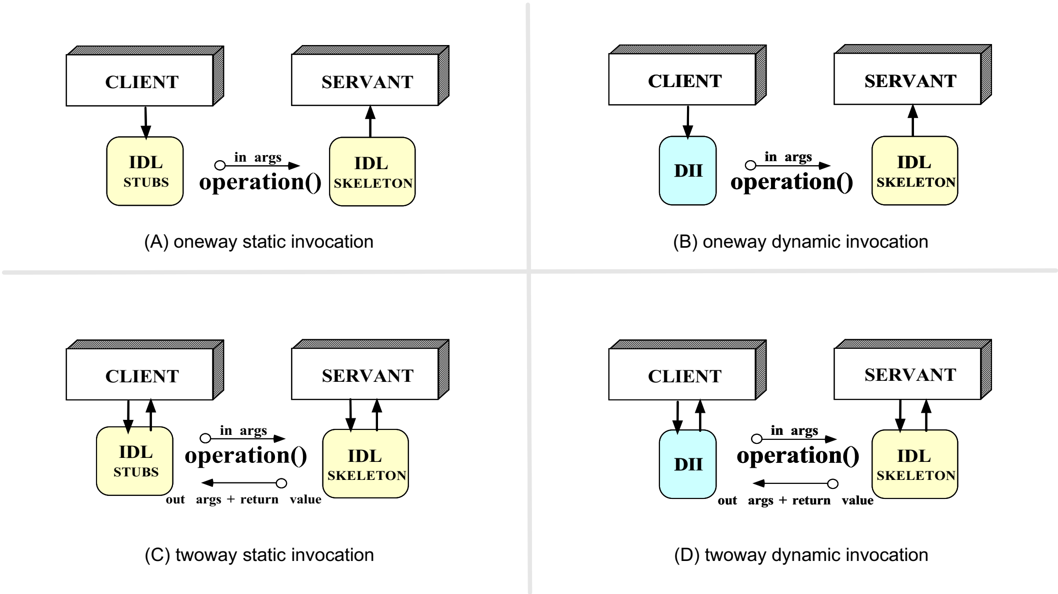

Figure 3: CORBA Operation Invocation Strategies

• **TCP "No Delay" option** Since the request sizes for our tests are relatively small, the `TCP_NODELAY` option is set on the client side. Without the `TCP_NODELAY` option, the client uses Nagel's algorithm, which buffers "small" requests until the preceding small request is acknowledged. On high-speed networks the use of Nagel's algorithm can increase latency unnecessarily. Since this paper focuses on measuring the latency of small requests, we enabled the `TCP_NODELAY` option to send packets immediately.

• **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.

3.4 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` [15] 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.

3.5 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 3.

• **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 3 (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 3 (B).

• **Twoway static invocation** The client uses the static invocation interface (SII) stubs for twoway operations defined in

IDL interfaces, as shown in Figure 3 (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 3 (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 4.4.

3.6 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.

Conventional ORBs (including Orbix and VisiBroker) demultiplex client requests to the appropriate operation of the servant implementation using the following steps shown in Figure 4:

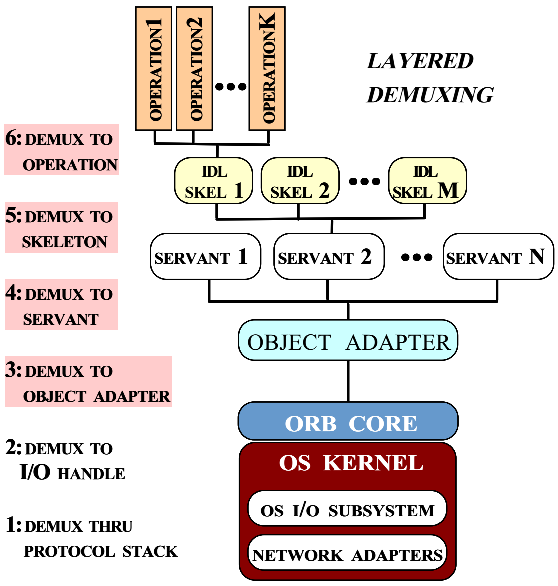

Figure 4: Layered CORBA Request Demultiplexing

• **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;

• **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 [16]. Layering increases latency by multiplying the number internal tables that must be searched as incoming client requests traverse a protocol stack. 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, [5, 17] show that $\sim$17% of the total server processing time is spent demultiplexing requests. Unless this overhead is reduced and demultiplexing is performed predictably, ORBs cannot provide consistent quality of service guarantees to applications.

Prior work [18] 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 paper, 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.

3.7 Request Invocation Algorithms

The experiments conducted for this paper 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.

Section 5 describes how we applied active demultiplexing and perfect hashing to optimized demultiplexing in our high-performance, real-time ORB called TAO [19, 18]. 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.

3.7.1 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.

3.7.2 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);
```

3.7.3 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.

3.7.4 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.

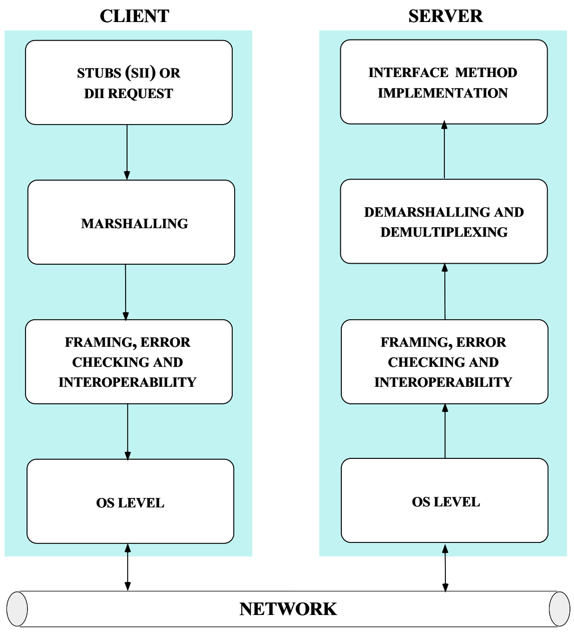

Figure 5: General Path of CORBA Requests

4 Performance Results for CORBA Latency and Scalability over ATM

This section presents the performance results from our latency and scalability experiments. Sections 4.1 and 4.2 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 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 5.

4.1 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.

4.1.1 Latency and Scalability of Parameterless Operations

Figures 6 and 7 depict the average latency for the parameterless operations using the request train variation of our request invocation algorithm. Likewise, Figures 8 and 9 depict the average latency for invoking parameterless operations using the round robin algorithm.

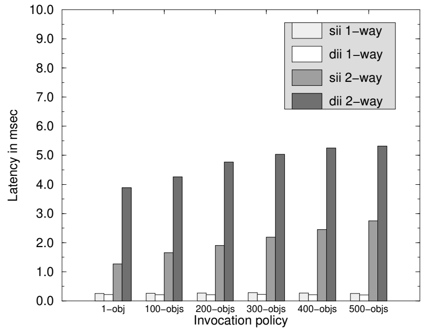

Figure 6: Orbix: Latency for Sending Parameterless Operation using request train Requests

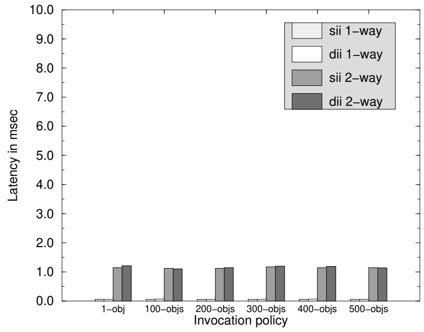

Figure 7: VisiBroker: Latency for Sending Parameterless Operation using request train Requests

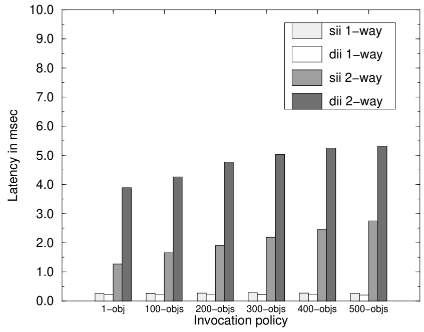

Figure 8: Orbix: Latency for Sending Parameterless Operation using Round Robin Requests

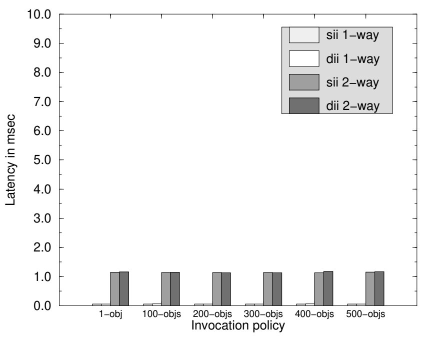

Figure 9: VisiBroker: Latency for Sending Parameterless Operation using round robin Requests

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 twoway 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.

We identified the problem with Orbix by using `truss`, which is a Solaris tool for tracing the system calls at runtime. 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 6 and 8 reveal an interesting case with Orbix's oneway latency. The oneway SII and DII latencies remain slightly less than their corresponding twoway latencies until 200 servants on the server. Beyond this, the oneway latencies exceed their corresponding twoway 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 twoway latency does not incur this flow control overhead since the sender blocks for a response after every request.

Figure 10 compares the twoway latencies obtained for sending parameterless operations for Orbix and VisiBroker with that of a low-level C++ implementation that uses sockets directly. The twoway latency comparison reveals that the Visi-

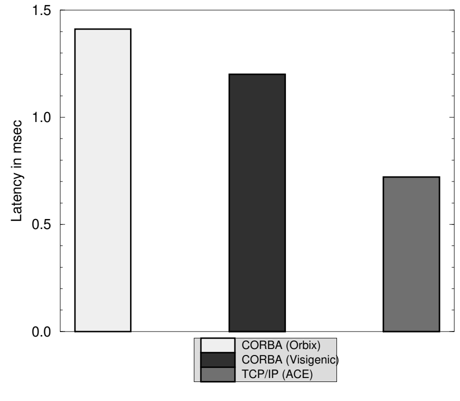

Figure 10: Comparison of Twoway Latencies

[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.

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

4.1.2 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 twoway 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 twoway 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 twoway SII latency. In Orbix DII, a new request must be created per invocation.[3]
- Twoway DII latency for Orbix is always greater than its twoway 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.

4.2 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.

[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.

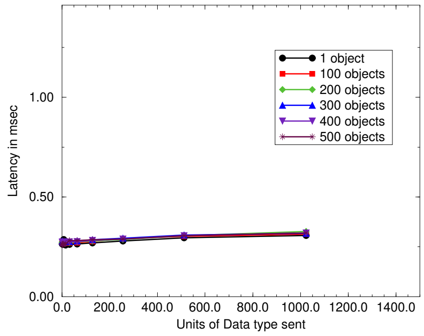

Figure 11: Orbix Latency for Sending Octets Using Oneway SII

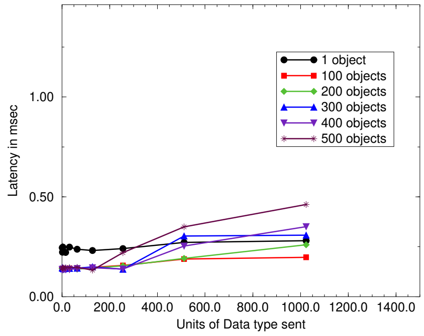

Figure 12: VisiBroker Latency for Sending Octets Using Oneway SII

4.2.1 Latency and Scalability of Parameter Passing Operations

Figures 11 through 18 depict the average latency for sending richly-typed `struct` data and untyped `octet` data using (1) the oneway operation invocation strategies (described in Section 3.5) and (2) varying the number of servants (described in Section 3.6). Similarly, Figures 19 through 26 depict the average latency for sending richly-typed `struct` data and untyped `octet` for twoway operations. These figures reveal that as the sender buffer size increases, the marshaling and data copying overhead also grows [5, 6], thereby increasing latency. These results demonstrate the benefit of using more efficient buffer management techniques and highly optimized stubs [9] 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 12 through 16, due to inefficient internal buffering strategies, there is substantial variance in latency. This jitter is

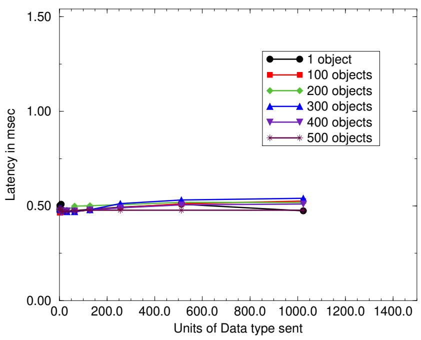

Figure 13: Orbix Latency for Sending Octets Using Oneway DII

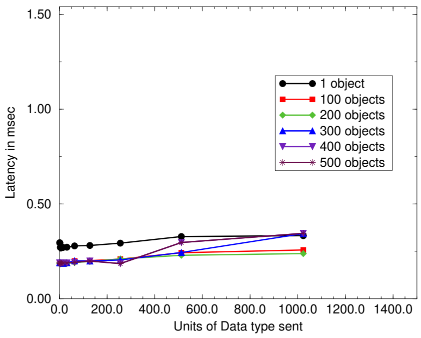

Figure 14: VisiBroker Latency for Sending Octets Using Oneway DII

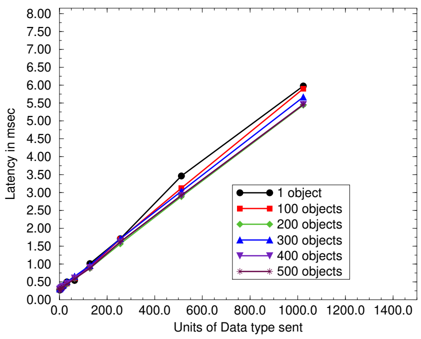

Figure 15: Orbix Latency for Sending Structs Using Oneway SII

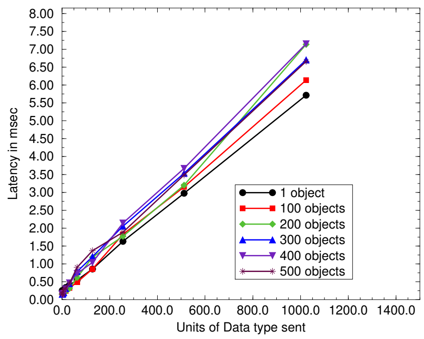

Figure 16: VisiBroker Latency for Sending Structs Using Oneway SII

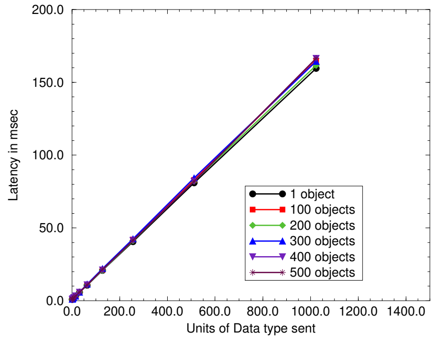

Figure 17: Orbix Latency for Sending Structs Using Oneway DII

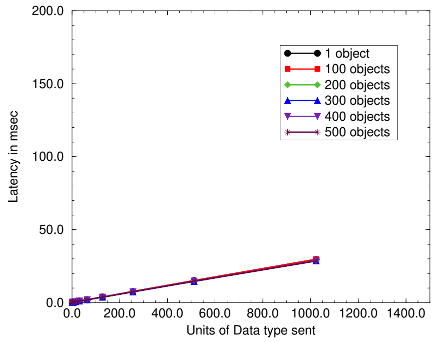

Figure 18: VisiBroker Latency for Sending Structs Using Oneway DII

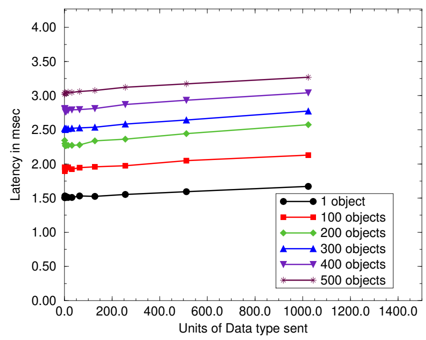

Figure 19: Orbix Latency for Sending Octets Using Twoway SII

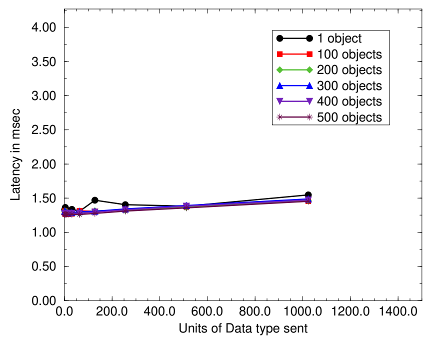

Figure 20: VisiBroker Latency for Sending Octets Using Twoway SII

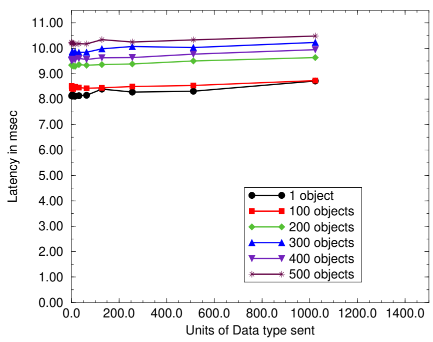

Figure 21: Orbix Latency for Sending Octets Using Twoway DII

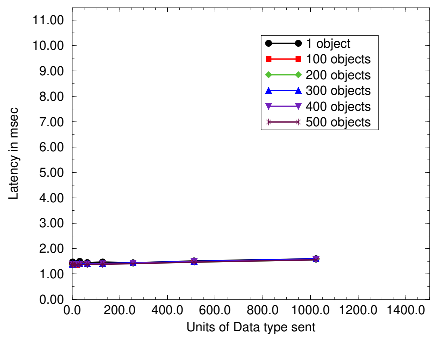

Figure 22: VisiBroker Latency for Sending Octets Using Twoway DII

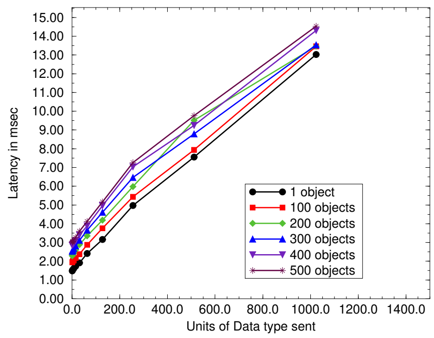

Figure 23: Orbix Latency for Sending Structs Using Twoway SII

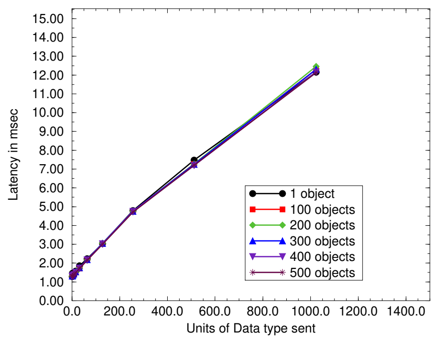

Figure 24: VisiBroker Latency for Sending Structs Using Twoway SII

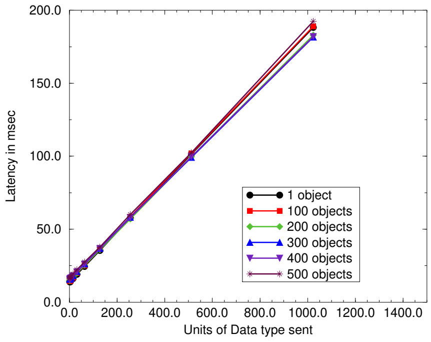

Figure 25: Orbix Latency for Sending Structs Using Twoway DII

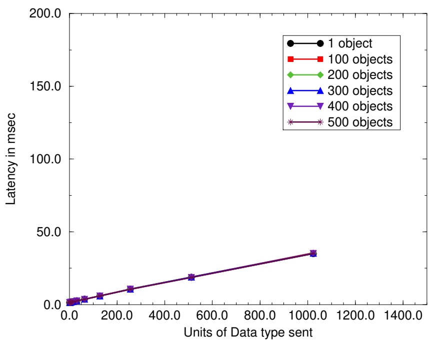

Figure 26: VisiBroker Latency for Sending Structs Using Twoway DII

generally unacceptable for real-time systems that require predictable behavior [4].

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 19 through 26 reveal that the twoway 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 23 through 26 also reveal that the latency for the Orbix twoway SII case at 1,024 data units of `BinStruct` is almost 1.2 times that for VisiBroker.

Similarly, the latency for the Orbix twoway 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, 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.

[5, 6] 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 presents our analysis of the `Quantify` results for sources of overhead that increase the latency of client request processing.

4.2.2 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 twoway 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`.

4.3 Whitebox Analysis of Latency and Scalability Overhead

Sections 4.1 and 4.2 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. The performance results reported in this section motivated the latency and scalability optimizations applied to our TAO ORB in Section 5.

Figures 27 and 28 show how Orbix and VisiBroker implement the generic SII request path shown in Figure 5. 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.

4.3.1 Orbix SII Request Flow Overhead

In Figure 27, the Orbix sender invokes the stub for the `ttcp_sequence::sendStructSeq` method. The request traverses through the `invoke` and the `send` rou-

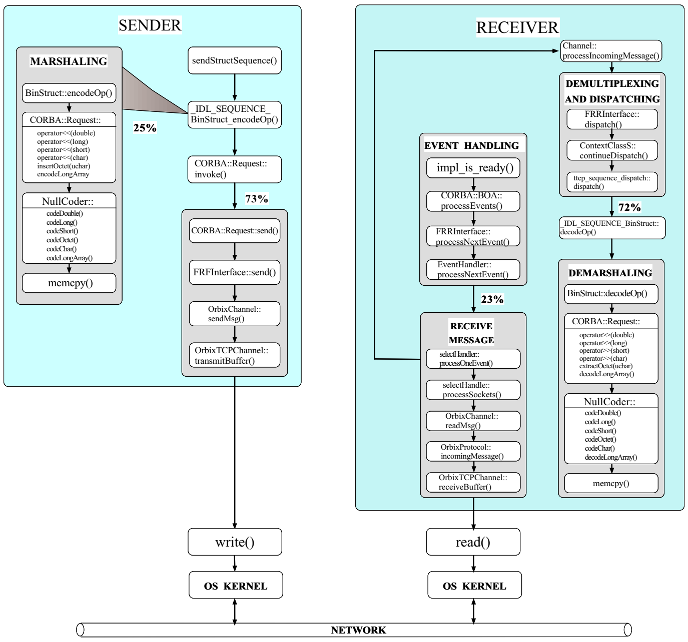

Figure 27: Request Path Through Orbix Sender and Receiver for SII

tines 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.

4.3.2 VisiBroker SII Request Flow Overhead

In Figure 28, 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

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

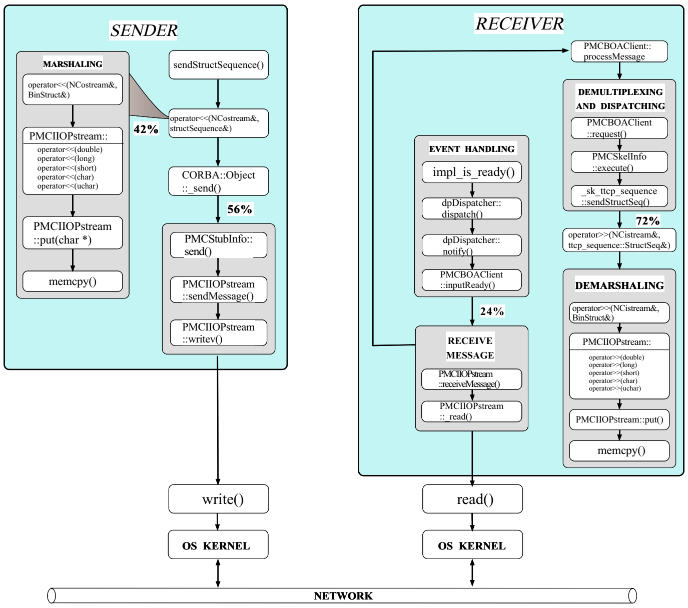

Figure 28: Request Path Through VisiBroker Sender and Receiver for SII

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

Table 1: Analysis of Servant Demultiplexing Overhead for Orbix

`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_sequence::skeleton`). 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 28), 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.

4.3.3 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 4.3.1 and 4.3.2.

Sources of Orbix demultiplexing overhead Table 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 which socket descriptor is ready

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

Table 2: Analysis of Servant Demultiplexing Overhead for VisiBroker

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 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 `~NCTransDict` 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 4.1).

- 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 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.

4.4 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 4.1, 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.

4.5 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 [20, 18]. Eliminating the presentation layer overhead requires optimized stub generators [21, 9] 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 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. Minimizing intra-ORB function calls requires sophisticated compiler optimizations such as integrated layer processing [1].

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
Inefficient lookup techniques	Use active de-layered demultiplexing and perfect hashing	Optimize demultiplexing strategies

Table 3: Optimization Principles Applied in TAO

• **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.

5 Techniques for Optimizing ORB Latency and Scalability

The performance results reported in Section 3 reveal the latency and scalability capabilities and limitations of conventional ORBs. We have used these results to guide the development of a high-performance, real-time ORB, called TAO [19, 4]. This section gives an overview of TAO and explains how the ORB optimization principles shown in Table 3 have been applied systematically to improve its latency and scalability.

5.1 Overview of TAO

TAO is an ORB endsystem that contains the network interface, operating system, communication protocol, and CORBA middleware components and features shown in Figure 29. TAO is based on the standard OMG CORBA reference model, with the following enhancements designed to overcome the shortcomings of conventional ORBs for high-performance and real-time applications:

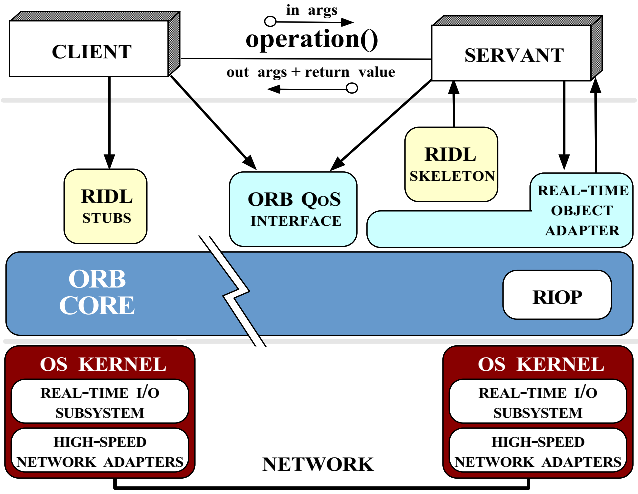

Figure 29: Components in the TAO Real-time ORB

• **Real-time IDL Stubs and Skeletons** In addition to marshaling and demarshaling of operation parameters, TAO's Real-time IDL (RIDL) stubs and skeletons are responsible for ensuring that application timing requirements are specified and enforced end-to-end [22].

• **Real-time Object Adapter** In addition to associating servants with the ORB and demultiplexing incoming requests to servants, TAO's Real-time Object Adapter (ROA) dispatches servant operations in accordance with various real-time scheduling strategies such as Rate Monotonic and Earliest Deadline First [19].

• **ORB QoS Interface** Applications use the ORB QoS interface to map real-time processing requirements to ORB endsystem/network resources. Common real-time processing requirements include end-to-end latency bounds and periodic scheduling deadlines. Common ORB endsystem/network resources include CPU, memory, network connections and storage devices.

• **Real-time I/O subsystem** TAO's real-time I/O subsystem performs admission control and assigns priorities to real-time I/O threads so that the schedulability of application components and ORB endsystem resources can be guaranteed.

• **High-speed network adapters** TAO's I/O subsystem contains a "daisy-chained" interconnect comprising a number of ATM Port Interconnect Controller (APIC) chips [23]. APIC is designed to sustain an aggregate bi-directional data rate of 2.4 Gbps. However, TAO also runs on conventional real-time interconnects such as 1553 buses, VME backplanes, and multiprocessor shared memory environments.

5.2 Overview of TAO Optimizations

We are developing TAO to overcome the following limitations with conventional ORBs:

Non-optimal demultiplexing strategies Conventional ORBs utilize inefficient and inflexible demultiplexing strategies based on layered demultiplexing, as explained in Section 4.3 and shown in Figure 30(A & B). In contrast, TAO utilizes perfect hashing and active demultiplexing in conjunction with explicit dynamic linking [18] shown in Figure 30(C & D). These strategies make it possible to adapt and configure optimal demultiplexing of client requests within ORB endsystems.

Inefficient presentation layer conversions Conventional ORBs are not optimized to generate efficient stubs and skeletons. As a result, they incur excessive marshaling and demarshaling overhead ([5, 6] and this paper in Figures 27 and 28) thereby adversely affecting latency. In contrast, TAO produces and configures multiple encoding/decoding strategies for interface definition language (IDL) descriptions. Each strategy can be configured for different time/space tradeoffs between compiled vs. interpreted OMG IDL stubs and skeletons [24], and the application's use of parameters (*e.g.*, pass-without-touching, read-only, mutable).

Excessive data copying and intra-ORB calls Conventional ORBs are not optimized to reduce the overhead of data copies. In addition, these ORBs suffer from excessive intra-ORB function call overhead as shown in Section 4.3. In contrast, TAO uses advanced compiler techniques, such as program flow analysis [25, 26] and integrated layer processing (ILP) [1] to automatically omit unnecessary data copies between the CORBA infrastructure and applications. In addition, ILP reduces the overhead of excessive intra-ORB function calls. Most importantly, this streamlining can be performed without requiring modifications to the standard CORBA specification.

Lack of integration with advanced OS and Network features Conventional ORBs do not fully utilize advanced OS and network features, such as real-time threads, high-speed network interfaces, and I/O subsystems with QoS support. In contrast, TAO integrates a high-performance I/O subsystem and the APIC network adapter with its ORB Core and optimized Object Adapter to produce a real-time ORB endsystem [19] that interoperates seamlessly with IIOP-compliant ORBs.

Non-optimized buffering algorithms used for network reads and writes Conventional ORBS utilize non-optimized internal buffers for writing to and reading from the network, as shown in Section 4.3. This causes the ORBs to spend a significant amount of time doing reads and writes affecting latency adversely. In contrast, TAO utilizes optimal buffer choices to reduce this overhead.

The remainder of this section is organized as follows: Section 5.3 describes the demultiplexing strategies supported by TAO and shows how these strategies can provide predictable and consistent low delay to latency-critical applications; Section 5.4 describes the principles we used to optimize TAO's end-to-end latency.

5.3 Increasing ORB Scalability via Demultiplexing Optimizations

The results of our measurements in Section 4.3.3 revealed that the Object Adapter's demultiplexing strategy has a significant impact on ORB endsystem scalability. As a result, we designed TAO's Object Adapter to support multiple demultiplexing strategies [27]. Section 5.3.2 presents the results of experiments using the following four demultiplexing strategies available in TAO: (A) linear search, (B) dynamic hashing, (C) perfect hashing, and (D) de-layered active demultiplexing shown in Figure 30:

Linear search The linear search demultiplexing strategy is a two-step layered demultiplexing strategy (shown in Figure 30(A)). In the first step, the Object Adapter uses the object key to linearly search through the *servant map*, which associates object keys to servants maintained by an Object Adapter, to locate the right servant and its skeleton. Each entry in the servant map maintains a pointer to its associated skeleton. In turn, the skeleton maintains an operation map defined by the IDL interface. In the second step, the Object Adapter uses the operation name to linearly search the operation map 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, and (2) to contrast our optimizing demultiplexing strategies with strategies used in conventional ORBs (such as Orbix) that use linear search for their operation demultiplexing.

Dynamic hashing The dynamic hashing strategy is another two-step layered demultiplexing strategy (shown in Figure 30(B)). 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 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 servant and the operation hash tables contained twice as many array elements as the number of servants and operations, respectively.

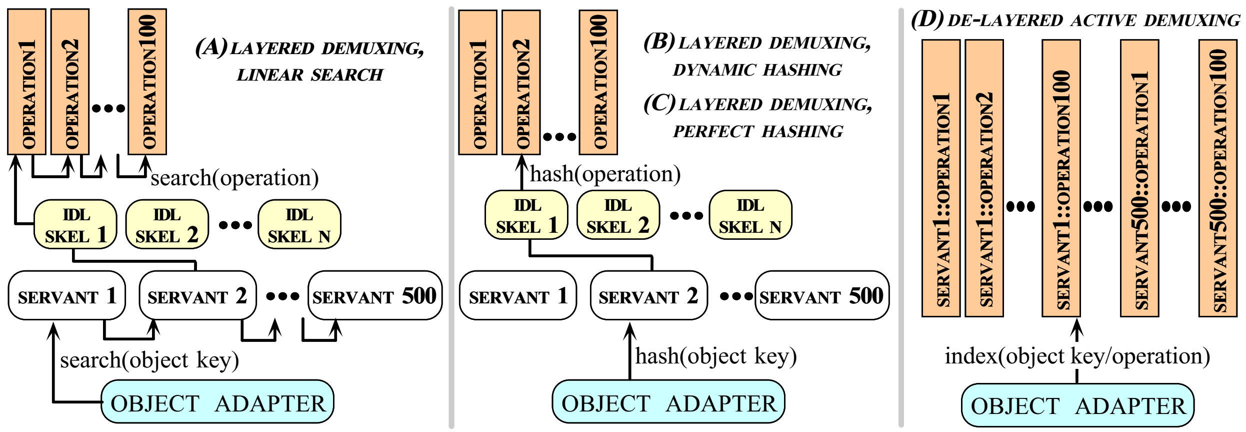

Figure 30: Demultiplexing Strategies

Perfect hashing The perfect hashing strategy is also a two-step layered demultiplexing strategy (shown in Figure 30(C)). 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 [4]), the servants and operations can be configured statically. In this scenario, it is possible to use perfect hashing to hash the servant and operations. For our experiment, we used the GNU `gperf` [28] 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 Servant_Hash
{
  // ...
  static u_int hash (const char *str, int len);
};

u_int
Servant_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]];
}
```

TAO uses the code shown above 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 servant map by using the perfect `hash` function shown above. The `hash` function uses an automatically generated servant map (`asso_values`) to return a unique hash value for each object key.

Active demultiplexing The fourth demultiplexing strategy is called *active demultiplexing* (shown in Figure 30(D)). In this strategy, the client includes a handle to the servant in the servant map and the operation table in the CORBA request header. This handle is configured into the client when the servant 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 servant and its associated operation in a single step. It is possible to implement active demultiplexing in a de-layered manner, so only one $O(1)$ table lookup is required to associate the incoming request with its servant operation.

5.3.1 Parameter Settings for Demultiplexing Experiments

This section describes the parameter settings for analyzing the behavior of each demultiplexing scheme described above.

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 this demultiplexing overhead and to evaluate the efficiency of different demultiplexing strategies, we benchmarked various numbers of servants on the server, ranging from 1, 100, 200, 300, 400, to 500.

Number of operations defined by the interface In addition to the number of servants, demultiplexing 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. Section 5.4 describes our latency optimizations that reduce the overhead of presentation layer conversions.

5.3.2 Performance Results

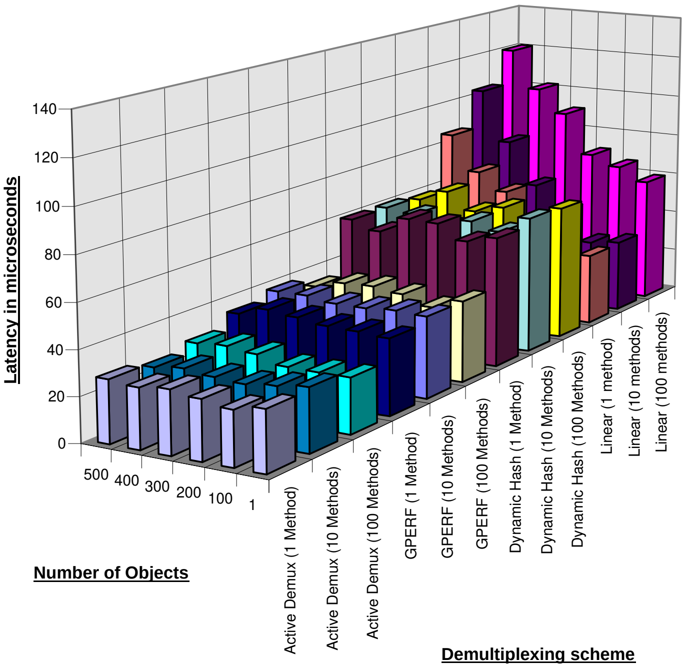

Figure 31: Demultiplexing Overhead for the random invocation Strategy

Figures 31 and 32 illustrate the performance of the four demultiplexing strategies for the random and worst-case invocation strategies described in Sections 3.7.3 and 3.7.4, respectively. These figures reveal that in both cases, the 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 servants 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 active demultiplexing strategy performs slightly better than the perfect hash-based strategy for both invocation strategies.

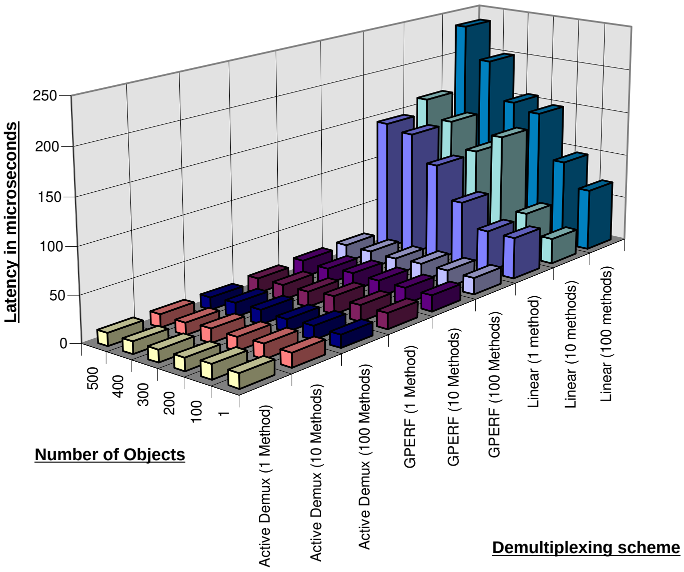

Figure 32: Demultiplexing Overhead for the worst-case invocation Strategy

5.3.3 Analysis of the Demultiplexing Strategies

The performance results and analysis presented in Sections 5.3.2 reveals 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 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, servants implementing an interface can be created dynamically. In this case, the perfect hashing strategy cannot generally be used for servant lookup.[5] 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 active demultiplexing outperforms the other demultiplexing strategies. However, it requires the client to possess a handle for each servant and its associated operations in a servant map and operation map, respectively. Therefore, active demultiplexing requires either (1) preconfiguring

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

the client with this knowledge or (2) defining a protocol for dynamically managing handles to add and remove servants correctly and securely.[6]

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 [19, 4].

5.4 Reducing ORB Latency with IIOP Optimizations

To expedite the research goals of the TAO project, and to avoid re-inventing existing components, we based TAO on SunSoft IIOP, which is a freely available reference implementation of the Internet Inter-ORB Protocol (IIOP). SunSoft IIOP is written in C++ and provides many features of a CORBA 2.0 ORB. However, it performs poorly over high-speed networks [8].

The source of overhead in SunSoft IIOP include many factors reported in Section 4. In particular, it has (1) high invocation overhead for small, frequently called methods, (2) repeated computation of invariant values, (3) excessive memory management and data copying overhead, and (4) inefficient, large functions that overflow the process cache. To alleviate this overhead, we applied a set of *principle-based optimizations* [29] to SunSoft IIOP in order to improve the performance of TAO.

Table 3 summarizes the optimization principles used in TAO. These principles 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, (7) optimizing for processor cache affinity, and (8) optimizing demultiplexing strategies*. The results of applying these optimization principles to SunSoft IIOP improved its latency substantially for all data types by alleviating the following source of overhead:

• **High invocation overhead for small, frequently called methods** TAO IIOP solves this problem using aggressive inlining based on *inline* functions and *macros*. This optimization uses the principle of *optimizing for the common case*.

• **Repeated computation of values that do not change** TAO IIOP solves this problem by computing the values once and storing them in additional storage. This optimization is based on the principles of *precomputing* and *using additional storage*.

• **Wasteful memory management** The implementation of the memory management system in SunSoft IIOP is overly generic. This causes it to interpretively deallocate primitive types, which instead can be freed wholesale. TAO IIOP remedies this problem by not interpreting buffers holding primitive data types. This optimization is based on the principle of *eliminating gratuitous waste*.

• **Inefficient, large functions that overflow the process cache** TAO IIOP streamlines the inefficient, large, and generic functions in SunSoft IIOP into smaller, efficient, and special-purpose functions. This optimization *improves cache affinity*.

Figures 33 and 34 indicate that the latency for the optimized IIOP version improves as the size of data transferred increases. For 1,024 units of data sent, the optimized IIOP version performs ~1.5 to 2.0 times better that the non-optimized IIOP version for all the primitive data types. For `BinStructs`, TAO's latency is ~4 times lower that the original SunSoft IIOP version.

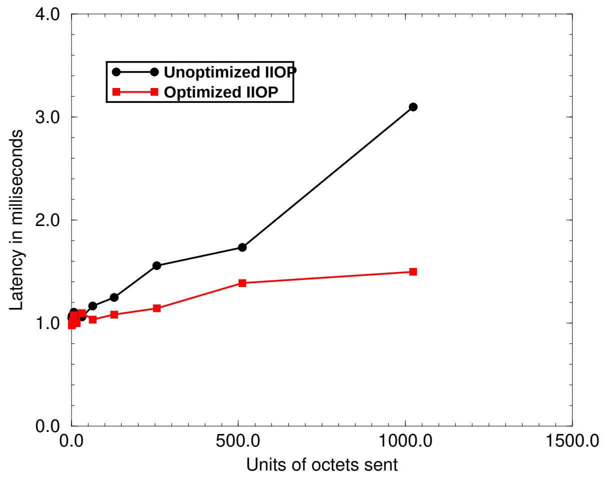

Figure 33: Improvement in Latency for Octets

The optimized TAO implementation of IIOP is now competitive with existing commercial ORBs using CORBA's static invocation interface. Moreover, TAO's dynamic invocation implementation is 2 to 4.5 times (depending on the data type) faster than commercial ORBs.

6 Related Work

Existing research on measuring latency in high performance networking has focused extensively on enhancements to

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

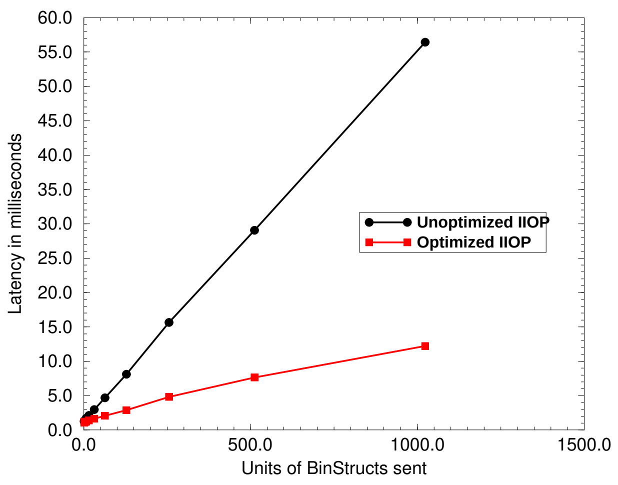

Figure 34: Improvement in Latency for Structs

TCP/IP. None of the systems described below are explicitly targeted for the requirements and constraints of communication middleware like CORBA. In general, less attention has been paid to integrating the following topics related to communication middleware:

6.1 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. [12, 13, 14] present results on performance of TCP/IP (and UDP/IP [12]) 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. [12] shows that UDP performs better than TCP over ATM networks, which is attributed to redundant TCP processing overhead on highly-reliable ATM links. [12] 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.

[30] 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. [31] 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. [32] discusses the `TCP_NODELAY` option, which allows TCP to send small packets as soon as possible to reduce latency.

Earlier work [5, 6] using untyped data and typed data in a similar CORBA/ATM testbed as the one in this paper 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.

6.2 Presentation Layer and Data Copying

The presentation layer is a major bottleneck in high-performance communication subsystems [1]. This layer transforms typed data from higher-level representations to lower-level representations (marshaling) and vice versa (demarshaling). 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 description language to other forms such as a network wire format.

Eliminating the overhead of presentation layer conversions requires highly optimized stub compilers (*e.g.*, Universal Stub Compiler [21]) and the Flick IDL compiler [9]. The generated stub code must make an optimal tradeoff between compiled code (which is efficient, but large in size) and interpreted code (which is slow, but compact) [24].

Our earlier results [5, 6] have presented detailed measurements of presentation layer overhead for transmitting richly-typed data. Our results for sending `structs` reveal that with increasing sender buffer sizes, the marshaling overhead increases, thereby increasing the latency. We are designing an IDL compiler that will adapt according to the run-time access characteristics of various data types and operations. The run-time usage of a operation or data type can be used to dynamically link in either the compiled or an interpreted version of marshaling code.

6.3 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) [1, 33, 34] and *Integrated Layer Processing* (ILP) [1, 35, 36].

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. [37] 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.

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.

6.4 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 4). Layered multiplexing and demultiplexing is generally disparaged for high-performance communication systems [16] due to the additional overhead incurred at each layer.[20] describes a fast and flexible message demultiplexing strategy based on dynamic code generation. [18] 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 [18] that can select optimal demultiplexing strategies based on compile-time and run-time analysis of CORBA IDL interfaces.

7 Concluding Remarks

An important class of applications (such as avionics, distributed interactive simulation, and telecommunication systems) require scalable, low-latency communication. However, the results in this paper 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 [19] 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* [8], 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`.

Acknowledgments

We thank the anonymous reviewers for providing us with many useful suggestions for improvement. We also thank Chris Cleeland for implementing TAO's ORB Core and Sumedh Mungee for helping with the performance tests. In addition, we thank IONA and Visigenic for their help in supplying the CORBA ORBs used for these tests. Both companies are currently working to eliminate the latency overhead and scalability limitations described in this paper. We expect their forthcoming releases to perform much better over high-speed ATM networks.

References

[1] 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)*, Philadelphia, PA, Sept. 1990, ACM, pp. 200–208.

[2] Object Management Group, *The Common Object Request Broker: Architecture and Specification*, 2.0 edition, July 1995.

[3] Object Management Group, *CORBAServices: Common Object Services Specification, Revised Edition*, 95-3-31 edition, Mar. 1995.

[4] 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*, Atlanta, GA, October 1997, ACM.

[5] Aniruddha Gokhale and Douglas C. Schmidt, "Measuring the Performance of Communication Middleware on High-Speed Networks," in *Proceedings of SIGCOMM '96*, Stanford, CA, August 1996, ACM, pp. 306–317.

[6] 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*, London, England, November 1996, IEEE, pp. 50–56.

[7] 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*, vol. 9, no. 4, November/December 1996.

[8] Aniruddha Gokhale and Douglas C. Schmidt, "Principles for Optimizing CORBA Internet Inter-ORB Protocol Performance," in *Hawaiian International Conference on System Sciences*, January 1998.

[9] 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.

[10] Object Management Group, *Specification of the Portable Object Adapter (POA)*, OMG Document orbos/97-05-15 edition, June 1997.

[11] USNA, *TTCP: a test of TCP and UDP Performance*, Dec 1984.

[12] 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.

[13] 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*, vol. 8, no. 1, pp. 43–48, February 1995.

[14] 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*, Minneapolis, MN, Oct. 1994, IEEE, pp. 176–185.

[15] PureAtria Software Inc., *Quantify User's Guide*, PureAtria Software Inc., 1996.

[16] David L. Tennenhouse, "Layered Multiplexing Considered Harmful," in *Proceedings of the* 1^{st} *International Workshop on High-Speed Networks*, May 1989.

[17] 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.

[18] 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.

[19] Douglas C. Schmidt, David L. Levine, and Sumedh Mungee, "The Design and Performance of Real-Time Object Request Brokers," *Computer Communications*, 1997, to appear.

[20] 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*, Stanford University, California, USA, August 1996, pp. 53–59, ACM Press.

[21] 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, Aug. 1994.

[22] Victor Fay Wolfe, Lisa Cingiser DiPippo, Roman Ginis, Michael Squadrito, Steven Wohlever, Igor Zykh, and Russel Johnston, "Real-Time CORBA," in *Proceedings of the Third IEEE Real-Time Technology and Applications Symposium*, Montréal, Canada, June 1997.

[23] Zubin D. Dittia, Guru M. Parulkar, and Jr. Jerome R. Cox, "The APIC Approach to High Performance Network Interface Design: Protected DMA and Other Techniques," in *Proceedings of INFOCOM '97*, Kobe, Japan, April 1997, IEEE.

[24] 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.

[25] Jong-Deok Choi, Ron Cytron, and Jeanne Ferrante, "Automatic Construction of Sparse Data Flow Evaluation Graphs," in *Conference Record of the Eighteenth Annual ACE Symposium on Principles of Programming Languages*. ACM, January 1991.

[26] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck, "Efficiently Computing Static Single Assignment Form and the Control Dependence Graph," in *ACM Transactions on Programming Languages and Systems*. ACM, October 1991.

[27] Douglas C. Schmidt and Chris Cleeland, "Applying Patterns to Develop Extensible and Maintainable ORB Middleware," *Communications of the ACM, to appear*, Dec. 1997.

[28] Douglas C. Schmidt, "GPERF: A Perfect Hash Function Generator," in *Proceedings of the* 2^{nd} *C++ Conference*, San Francisco, California, April 1990, USENIX, pp. 87–102.

[29] George Varghese, "Algorithmic Techniques for Efficient Protocol Implementations ," in *SIGCOMM '96 Tutorial*, Stanford, CA, August 1996, ACM.

[30] Jonathan Kay and Joseph Pasquale, "The Importance of Non-Data Touching Processing Overheads in TCP/IP," in *Proceedings of SIGCOMM '93*, San Francisco, CA, September 1993, ACM, pp. 259–269.

[31] Christos Papadopoulos and Gurudatta Parulkar, "Experimental Evaluation of SUNOS IPC and TCP/IP Protocol Implementation," *IEEE/ACM Transactions on Networking*, vol. 1, no. 2, pp. 199–216, April 1993.

[32] 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, 1989.

[33] 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.

[34] 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.

[35] M. Abbott and L. Peterson, "Increasing Network Throughput by Integrating Protocol Layers," *ACM Transactions on Networking*, vol. 1, no. 5, October 1993.

[36] 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.

[37] 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.