Optimizing the Performance of the CORBA Internet Inter-ORB Protocol Over ATM

Authors: Aniruddha Gokhale and Douglas C. Schmidt

The Internet Inter-ORB Protocol (IIOP) enables heterogeneous CORBA-compliant Object Request Brokers (ORBs) to interoperate over TCP/IP networks. The IIOP uses the Common Data Representation (CDR) transfer syntax to map CORBA Interface Definition Language (IDL) data types into a bi-canonical wire format. Due to the excessive marshaling/demarshaling overhead, data copying, and high-levels of function call overhead, conventional implementation of IIOP protocols yield poor performance over high-speed networks.

To meet the demands of emerging distributed multimedia applications, CORBA-compliant ORBs must support both interoperable and highly efficient IIOP implementations. This paper provides two contributions to the study and design of high performance CORBA IIOP implementations. First, we precisely pinpoint the key sources of overhead in the SunSoft IIOP implementation (which is the standard reference implementation of IIOP written in C++) by measuring its performance for transferring richly-typed data over a high speed ATM network. Second, we empirically demonstrate the benefits that stem from systematically applying protocol optimizations to SunSoft IIOP. These optimizations include: optimizing for the expected case; eliminating obvious waste; replacing general purpose methods with specialized,... Read complete abstract on page 2.

Follow this and additional works at: http://openscholarship.wustl.edu/cse_research

Part of the Computer Engineering Commons, and the Computer Sciences Commons

Recommended Citation

http://openscholarship.wustl.edu/cse_research/427
Optimizing the Performance of the CORBA Internet Inter-ORB Protocol Over ATM

Complete Abstract:

The Internet Inter-ORB Protocol (IIOP) enables heterogeneous CORBA-compliant Object Request Brokers (ORBs) to interoperate over TCP/IP networks. The IIOP uses the Common Data Representation (CDR) transfer syntax to map CORBA Interface Definition Language (IDL) data types into a bi-canonical wire format. Due to the excessive marshaling/demarshaling overhead, data copying, and high-levels of function call overhead, conventional implementation of IIOP protocols yield poor performance over high-speed networks. To meet the demands of emerging distributed multimedia applications, CORBA-compliant ORBs must support both interoperable and highly efficient IIOP implementations. This paper provides two contributions to the study and design of high performance CORBA IIOP implementations. First, we precisely pinpoint the key sources of overhead in the SunSoft IIOP implementation (which is the standard reference implementation of IIOP written in C++) by measuring its performance for transferring richly-typed data over a high speed ATM network. Second, we empirically demonstrate the benefits that stem from systematically applying protocol optimizations to SunSoft IIOP. These optimizations include: optimizing for the expected case; eliminating obvious waste; replacing general purpose methods with specialized, efficient ones; precomputing values, if possible; storing redundant state to speed up expensive operations; and passing information between layers. The results of applying these optimization principles to SunSoft IIOP improved its performance 1.8 times for doubles, 3.3 times for longs, 3.75 times for shorts, 5 times for chars/octets, and 4.2 times for richly-typed structs over ATM networks. Our optimized implementation is now competitive with existing commercial ORBs using the static invocation interface (SII) and 2 to 4.5 times (depending on the data type) faster than commercial ORBs using the dynamic skeleton interface (DSI). Moreover, our optimizations are fully CORBA compliant and we maintain strict interoperability with other IIOP implementations such as Visigenic’s VisiBroker and IONA’s Orbix.
Optimizing the Performance of the CORBA Internet Inter-ORB Protocol Over ATM

Aniruddha Gokhale and Douglas C. Schmidt

WUCS-97-10

February 1997

Department of Computer Science
Washington University
Campus Box 1045
One Brookings Drive
St. Louis MO 63130
Optimizing the Performance of the CORBA
Internet Inter-ORB Protocol Over ATM

Aniruddha Gokhale and Douglas C. Schmidt

gokhale@cs.wustl.edu and schmidt@cs.wustl.edu
Department of Computer Science, Washington University
St. Louis, MO 63130, USA

This paper has been submitted for publication. It can be referenced as Washington University technical report #WUCS-97-10.

Abstract

The Internet Inter-ORB Protocol (IIOP) enables heterogeneous CORBA-compliant Object Request Brokers (ORBs) to interoperate over TCP/IP networks. The IIOP uses the Common Data Representation (CDR) transfer syntax to map CORBA Interface Definition Language (IDL) data types into a bi-canonical wire format. Due to the excessive marshaling/demarshaling overhead, data copying, and high-levels of function call overhead, conventional implementations of IIOP protocols yield poor performance over high speed networks. To meet the demands of emerging distributed multimedia applications, CORBA-compliant ORBs must support both interoperable and highly efficient IIOP implementations.

This paper provides two contributions to the study and design of high performance CORBA IIOP implementations. First, we precisely pinpoint the key sources of overhead in the SunSoft IIOP implementation (which is the standard reference implementation of IIOP written in C++) by measuring its performance for transferring richly-typed data over a high speed ATM network. Second, we empirically demonstrate the benefits that stem from systematically applying protocol optimizations to SunSoft IIOP. These optimizations include: optimizing for the expected case; eliminating obvious waste; replacing general purpose methods with specialized, efficient ones; precomputing values, if possible; storing redundant state to speed up expensive operations; and passing information between layers.

The results of applying these optimization principles to SunSoft IIOP improved its performance 1.8 times for doubles, 3.3 times for longs, 3.75 times for shorts, 5 times for chars/octets, and 4.2 times for richly-typed structures over ATM networks. Our optimized implementation is now competitive with existing commercial ORBs using the static invocation interface (SII) and 2 to 4.5 times (depending on the data type) faster than commercial ORBs using the dynamic skeleton interface (DSI). Moreover, our optimizations are fully CORBA compliant and we maintain strict interoperability with other IIOP implementations such as Visigenic’s VisiBroker and IONA’s Orbix.

Keywords: Distributed object computing, CORBA, IIOP, communication middleware protocol optimizations, high-speed networks.

1 Motivation

An increasingly important class of distributed applications require stringent quality of service (QoS) guarantees. These applications include telecommunication systems (e.g., call processing and switching), avionics control systems (e.g., operational flight programs for fighter aircraft), multimedia (e.g., video-on-demand and teleconferencing), and simulations (e.g., battle readiness planning). In addition to requiring QoS guarantees, these applications must be flexible and reusable.

The Common Object Request Broker Architecture (CORBA) is a distributed object computing middleware standard defined by the Object Management Group (OMG) [15]. CORBA is intended to support the production of flexible and reusable distributed services and applications. Many implementations of CORBA are now available.

The CORBA 2.0 specification requires Object Request Brokers (ORBs) to support a standard interoperability protocol. The CORBA specification defines an abstract interoperability protocol called the General Inter-ORB Protocol (GIOP). Specialized mappings of GIOP can be defined to operate over particular transport protocols. One such mapping is called the Internet Inter-ORB Protocol (IIOP), which is the emerging standard GIOP mapping for distributed object computing over TCP/IP. The latest release of Netscape integrates IIOP into its Web browser, making IIOP the most widely available protocol for interoperability between heterogeneous ORBS.

[8, 9, 10] show that the performance of CORBA implementations is poor compared to that of the low level implementations using C/C++ since the ORBs incur a significant amount of data copying, marshaling, demarshaling, and demultiplexing overhead. These results, however, were restricted to measuring the performance of communication between homogeneous ORBS. They do not measure the runtime costs of interoperability between heterogeneous ORBS. In addition, earlier work on measuring CORBA performance does not pro-
vide solutions for reducing key sources of ORB overhead.

In this paper, we measure the performance of the standard reference implementation of IIOP, written by SunSoft, using a CORBA/ATM testbed environment similar to [8, 9, 10]. We measure the performance of SunSoft IIOP and precisely pinpoint the performance overhead. In addition, we describe how we applied six principle-driven optimizations [21] to substantially improve the performance of SunSoft IIOP. These optimizations include: optimizing for the expected case; eliminating obvious waste; replacing general purpose methods with specialized, efficient ones; precomputing values, if possible; storing redundant state to speed up expensive operations; and passing information between layers.

The results of applying these optimization principles to SunSoft IIOP improved its performance 1.8 times for doubles, 3.3 times for longs, 3.75 times for shorts, 5 times for chars/octet, and 4.2 times for richly-typed structs over ATM networks. Our optimized IIOP implementation is now comparable to existing commercial ORBS [8, 9, 10] using the static invocation interface (SII) and around 2 to 4.5 times (depending on the data type) faster than commercial ORBS using the dynamic skeleton interface (DSI). The optimizations and the resulting speedups reported in this paper are essential for CORBA to be adopted as the standard for implementing high-bandwidth, low-latency distributed applications.

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

This paper is organized as follows. Section 2 outlines the CORBA reference model, the GIOP/IIOP interoperability protocols, and the SunSoft IIOP reference implementation; Section 3 presents the results of our performance optimizations of SunSoft IIOP over a high-speed ATM network; Section 4 compares our research with related work; and Section 5 provides concluding remarks.

2 Overview of CORBA, GIOP/IIOP, and the SunSoft IIOP Reference Implementation

CORBA ORBS allow clients to invoke methods on target object implementations without concern for where the object resides, what language the object is written in, what OS/hardware platform it runs on, or what communication protocols and networks are used to interconnect distributed objects. To support this level of transparency, the CORBA reference model defines the components in Figure 1. These components are defined as follows:

* **Object Implementation**: This defines operations that implement a CORBA interface definition language (IDL) interface.

![Figure 1: Components in the CORBA Reference Model](image)

- **Client**: This is the program entity that invokes an operation on a (potentially remote) object implementation.
- **Object Request Broker (ORB) core**: When a client invokes a method, the ORB core is responsible for delivering the request to the object and returning a response (if any) to the client. CORBA-conformant ORBS use the GIOP and IIOP interoperability protocols described in Section 2.1.
- **ORB Interface**: The ORB interface provides standard operations that allow an ORB to decouple applications from the implementation details of the ORB.
- **CORBA IDL Stubs and Skeletons**: CORBA IDL stubs and skeletons serve as the "glue" between the client and server applications, respectively, and the ORB.
- **Dynamic Invocation Interface (DII)**: The DII allows a client to directly access the underlying request mechanisms provided by an ORB. DII is used when the ORB has no compile-time knowledge of the interface it is implementing. To retrieve this information, it has to query the interface repository.
- **Dynamic Skeleton Interface (DSI)**: The DSI is the server side's analogue to the client side's DII. The DSI allows an ORB to deliver requests to an object implementation or ORB bridge that has no compile-time knowledge of the type of the object it is implementing.
- **Object Adapter**: The Object Adapter associates an object implementation with the ORB and demultiplexes incoming requests to the appropriate method of the target object.

2.1 Overview of CORBA GIOP and IIOP

The CORBA General Inter-ORB Protocol (GIOP) defines an interoperability protocol between heterogeneous ORBS. The GIOP protocol provides an abstract interface that can be mapped onto conventional connection-oriented transport protocols. A concrete mapping of GIOP onto the TCP/IP transport protocol is known as the Internet Inter-ORB Protocol (IIOP). An ORB supports GIOP if some entity associated with the
ORB is able to send and receive GIOP messages. The GIOP specification consists of the following elements:

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

- **GIOP Message Formats:** The GIOP specification defines seven types of messages that send requests, receive replies, locate objects, and manage communication channels;

- **GIOP Transport Assumptions:** The GIOP specification describes the assumptions made regarding transport protocols that can be used to carry GIOP messages. In addition, the GIOP specification defines a connection management protocol and a set of constraints for message ordering.

The GIOP and IIOP specifications are described further in Appendix A. The remainder of this section presents an overview of the SunSoft IIOP reference implementation and describes the primary components of its implementation.

## 2.2 Overview of the SunSoft IIOP Implementation

### 2.2.1 CORBA Features Supported by SunSoft IIOP

The SunSoft IIOP implementation is widely regarded as the reference implementation of IIOP. It is freely available from `ftp://ftp.omg.org/pub/interop/`. SunSoft IIOP is written in C++ and provides most of the features of a CORBA 2.0 ORB, except for an IDL compiler, an interface repository, and a Basic Object Adapter. Therefore, users must provide stubs and skeletons that integrate with the APIs provided by SunSoft IIOP.

On the client-side, SunSoft IIOP provides a static invocation interface (SII) and a dynamic invocation interface (DII). The SII is used by the client-side stubs and server-side skeletons generated by manually or by an IDL compiler. The DII is used by applications that have no compile-time knowledge of the interface they are calling. Thus, the DII allows applications to create CORBA requests at run-time. It marshals parameters by using interfaces containing methods of CORBA pseudo-object classes and by consulting the interface repository. Since SunSoft IIOP does not provide an IDL compiler, client stubs using the SII API must be hand-crafted.

On the server-side, SunSoft IIOP supports the dynamic skeleton interface (DSI). The DSI is used by applications (such as ORB bridges [15]) that have no compile-time knowledge of the interfaces they implement. The SunSoft IIOP does not provide an interface repository or an IDL compiler that can generate skeletons. Therefore, it uses the DSI mechanism to parse incoming requests, unmarshal the parameters, and demultiplex the request to the appropriate object implementation. Servers that use the SunSoft DSI mechanism must provide it with `TypeCode`² information that it uses to interpret incoming requests and demarshal the parameters.

### 2.2.2 SunSoft IIOP Components

The SunSoft IIOP implementation is based on an interpretive marshaling/demarshaling engine. The motivation for using an interpretive design is to reduce the size of the marshaling/demarshaling engine so that it resides within a processor cache. An alternative approach is to use a compiled marshaling/demarshaling engine [11]. A compiled approach is possible when the marshaling/demarshaling engine has compile-time knowledge of the structural layout of IDL interfaces and operation parameters.

The primary components in the SunSoft IIOP implementation are shown in Figure 2: Each component is described below:

- **The `TypeCode::traverse` method:** The `traverse` method of the CORBA::`TypeCode` class implements an IIOP `TypeCode` interpreter. All marshaling and demarshaling of parameters is performed interpretively by traversing the data structure according to the layout of the `TypeCode/request` tuple provided to `traverse`. This method is passed a pointer to a `visit` method (described below:

1. CORBA pseudo-objects are entities that are neither CORBA primitive types nor constructed types. Operations on pseudo-object references cannot be invoked using the DII mechanism since the interface repository does not keep any information about them. In addition, only some pseudo-objects (such as `TypeCode` and `Any`) can be transferred as parameters to methods of an interface.

2. `TypeCodes` are CORBA pseudo-objects that describe the format and layout of primitive and constructed IDL data types in the incoming request stream.
below), which interprets CORBA requests based on their TypeCode layout. The request part of the tuple contains the data that was received from an application (i.e., on the client-side) or from the protocol stack (i.e., on the server-side).

- **The visit method**: The TypeCode interpreter invokes the visit method to marshal or demarshal the data associated with the TypeCode it is currently interpreting. The visit method is a pointer to a method that contains the address of one of the four methods described below:
  - **The CDR::encoder method** – The encoder method of the CDR class encodes various data types from their native host representation to the CDR representation used to transmit CORBA requests over the network.
  - **The CDR::decoder method** – The decoder method of the CDR class retrieves request values in the native host representation from the incoming CDR stream.
  - **The deep_copy method** – The deep_copy method is used by the SunSoft DII mechanism to create requests and marshal parameters into the CDR stream using the TypeCode interpreter.
  - **The deep_free method** – The deep_free method is used by the DSI server to free allocated memory after incoming data has been demarshaled and passed to a server application.

- **The utility methods**: SunSoft IIOP provides several methods that perform various utility tasks. The two most important utility methods include:
  - **The calc_nested_size_and_alignment method** – This method calculates the size and alignment of various fields that comprise an IDL struct.
  - **The struct_traverse method** – This method is used by the TypeCode interpreter to traverse the fields in an IDL struct.

### 2.2.3 Tracing the Data Path of an IIOP Request

To illustrate the run-time behavior of SunSoft IIOP, we trace the path taken by requests that transmit a sequence of BinStructs (shown in Appendix B). We show how the TypeCode interpreter consults the TypeCode information as it marshals and unmarshals parameters. The same BinStruct is used in our experiments described in Section 3.2.

- **TypeCode Layout for BinStructs**: Figure 3 depicts the internal representation of TypeCode information that defines the sequence of BinStructs shown in Appendix B. The layout of the sequence of BinStructs and its parameters are described below:
  - **TypeCode value** – A CORBA TypeCode data structure contains a _kind field that indicates the TCKind value, which is an enumerated type. In our “sequence of BinStruct” example, the TCKind value is tk.sequence.
  - **TypeCode length and byte order** – The length field indicates the length of the buffer that holds the CDR representation of the TypeCode’s parameters. In this example, the first byte of the CDR buffer indicates the byte order. Here, the value 0 indicates that big-endian byte-ordering is used.
  - **Element type** – For a sequence TypeCode, the next entry in the buffer is the TypeCode kind entry for the element type that makes the sequence. In our example, this value is tk.struct.
  - **Encapsulation length and sequence bound** – The next entry is a length field indicating the length of the encapsulation that holds information about the struct’s members. The length field is followed by the encapsulation, followed by a field that indicates the bounds of the sequence. A value of 0 indicates an “unbounded” sequence (i.e., the size of the sequence is determined at run-time, not at compile-time).
  - **Encapsulation content and field layouts** – The encapsulation is made up of two string entries, which follow the designation of the encapsulation’s byte-order. Each string entry has to have a length field specifying the length of the string followed by the string values. Following this field is a number indicating the number of members in the BinStruct IDL struct. This is followed by TypeCode layouts for each field in the struct.

- **Client-side Data Path**: The client-side data path is shown in Figure 4. This figure depicts the path traced by requests through the TypeCode interpreter. The CDR::encoder method marshals the parameters from native host format into a CDR representation suitable for transmission on the network.
The client uses the do.call method, which is the SII API provided by SunSoft IIOP that uses the TypeCode interpreter to marshal the parameters and send the requests.

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

The traverse method of the TypeCode interpreter consults the TypeCode layout passed to it by an application to determine the data types that comprise a constructed data type. For each member of a constructed data type, the interpreter invokes the same visit method that invoked it. In our case, the encoder is the visit method that originally called the interpreter. This process continues recursively until all parameters have been marshaled. At this point the request is transmitted over the network via the GIOP::Invocation::invoke method.

- **Server-side Data Path:** The server-side data path shown in Figure 5 depicts the path traced by requests through the TypeCode interpreter. The CDR::decoder method unmarshals the parameters from the CDR representation into the server’s native host format. An event handler (TCP_OA) waits for incoming data. After a CORBA request is received, its GIOP type is decoded. The server’s dispatching mechanism then dispatches the request to a skeleton via a user supplied upcall.

Since SunSoft IIOP does not provide a standard CORBA Object Adapter, the server programmer must write a tcp_oa.dispatch method that dispatches the request to the correct skeleton. The request demultiplexing and dispatching process can use any of a number of strategies to demultiplex requests to skeletons. These strategies include linear searching or hashing of operation names.

The SunSoft receiver supports the DSI mechanism. Therefore, an N维尔 CORBA pseudo-object is created and populated with the TypeCode information for the parameters retrieved from the incoming request. These parameters are retrieved by calling the ServerRequest::params method. As with the client, the server’s TypeCode interpreter uses the CDR::decoder_visit method to unmarshal individual data types into parameters that are subsequently passed to the server application’s upcall method.

3 Experimental Results of CORBA IIOP over ATM

3.1 CORBA/ATM Testbed Environment

3.1.1 Hardware and Software Platforms

The experiments in this section were conducted using a Bay Networks LattisCell 10114 ATM switch connected to two dual-processor UltraSPARC-2s running SunOS 5.5.1. The LattisCell 10114 is a 16 Port, OC3 155 Mbs/port switch. Each UltraSPARC-2 contains two 168 MHz Ultra 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 [17]. Each UltraSPARC-2 has 256 Mbytes of RAM and an ENI-155s-MF ATM adapter card, which supports 155 Megabits per-sec (Mbps) SONET multimode fiber. The Maximum Transmission Unit (MTU) on the ENI ATM adaptor is 9,180 bytes. Each ENI card has 512 Kbytes of on-board memory. A maximum of 32 Kbytes is allotted per ATM virtual circuit connection for receiving and transmitting frames (for a total of 64 K). This allows up to eight switched virtual connections per card. This hardware platform is shown in Figure 6.

3.1.2 Traffic Generator for Throughput Measurements

Traffic for the experiments was generated and consumed by an extended version of the widely available ttcp [20] protocol benchmarking tool. We extended ttcp for use with SunSoft IIOP. We hand-crafted the stubs and skeletons for the different methods defined in the interface. Our hand-crafted stubs use the SII API (do.call method) provided by SunSoft IIOP. On the server-side, the Object Adaptor uses a callback method (supplied by the ttcp server application) to dispatch incoming requests along with parameters to the target object.

Our ttcp tool measures end-to-end data transfer throughput in Mbps from a transmitter process to a remote receiver process across an ATM network. The flow of user data for each version of ttcp is unidirectional, with the transmitter flooding the receiver with a user-specified number of data buffers. Various sender and receiver parameters may be selected at run-time. These parameters include the number of
data buffers transmitted, the size of data buffers, and the type of data in the buffers. In all our experiments the underlying socket queue sizes were enlarged to 64 Kbytes (which is the maximum supported on SunOS 5.5.1).

The following data types were used for all the tests: primitive types (short, char, long, octet, double) and a C++ struct composed of all the primitives (BinStruct). The size of the BinStruct is 32 bytes. The IIOP implementation transferred the data types using IDL sequences, which are dynamically-sized arrays. The IDL declaration is shown in the appendix. The sender side transmitted data buffer sizes of a specific data type incremented in powers of two, ranging from 1 Kbyte to 128 Kbytes. These buffers were repeatedly sent until a total of 64 Mbytes of data was transmitted.

3.1.3 Profiling Tools

The profile information for the empirical analysis was obtained using the Quantify [19] performance measurement tool. Quantify analyzes performance bottlenecks and identifies sections of code that dominate execution time. Unlike traditional sampling-based profilers (such as the UNIX gprof tool), Quantify reports results without including its own overhead. In addition, Quantify measures the overhead of system calls and third-party libraries without requiring access to source code. All data is recorded in terms of machine instruction cycles and converted to elapsed times according to the clock rate of the machine. The collected data reflect the cost of the original program's instructions and automatically exclude any Quantify counting overhead.

Additional information on the run-time behavior of the code such as system calls made, their return values, signals, number of bytes written to the network interface, and number of bytes read from the network interface are obtained using the UNIX truss utility, which traces the system calls made by an application. truss is used to observe the return values of system calls such as write and read. This indicates the number of attempts made to write a buffer to a network and read a buffer from the network.
3.2 Performance Results and Optimization Benefits

3.2.1 Methodology

This section describes the optimizations we applied to SunSoft IIOP to improve its throughput performance over ATM networks. First, we show the performance of the original SunSoft IIOP for a range of IDL data types. Next, we use Quantify to illustrate the key sources of overhead in SunSoft IIOP. Finally, we describe the benefits of the optimization principles we applied to improve the performance of SunSoft IIOP.

The optimizations described in this section are based on principles for efficiently implementing protocols. [21] describes these principles in detail and describes how they are applied in existing protocol implementations, e.g., TCP/IP. We focus on the principles in Table 1 that improve IIOP performance. When describing our optimizations, we refer to these principles and explain how their use is justified.

The SunSoft IIOP optimizations were performed in the following four steps, corresponding to the principles from Table 1:

1. Inlining to optimize for the expected case – which is discussed in Section 3.2.3;
2. Aggressive inlining to optimize for the expected case – which is discussed in Section 3.2.4;
3. Precomputing, adding redundant state, and passing information through layers – which is discussed in Section 3.2.5;
4. Eliminating obvious waste and specializing generic methods – which is discussed in Section 3.2.6.

For each step, we describe the optimizations we applied to reduce the overhead remaining from the previous steps. After each step, we show the improved throughput measurements for various data types. In addition, we compare the throughput obtained in the previous steps with that obtained in the current step. The comparisons focus on data types that exhibited the widest range of performance. First two optimization steps shown below do not significantly improve
performance. However, these steps are important since they reveal the actual sources of overhead that are alleviated by the optimizations in steps three and four reported below.

3.2.2 Performance of the Original IIOP Implementation

- **Sender-side performance:** Figure 7 illustrates the sender-side throughput obtained for sending 64 Mbytes of various data types for sender buffer sizes ranging from 1 Kbyte to 128 Kbytes, incremented in powers of two. These results indicate that different data types achieved different levels of throughput. The highest throughput results from sending doubles, whereas BinStructs displayed the worst behavior. This variation in behavior stems from the marshaling and demarshaling overhead for different data types, as well as from the use of the interpretive marshaling/demarshaling engine in SunSoft IIOP that results in a large number of recursive method calls.

Tables 2 and 3 present the results of using the Quantify profiling tool to send 64 Mbytes of various data types using a 128 Kbyte sender buffer. The sender-side results reveal that for all data types, the sender spends most of its run-time performing write system calls to the network. In addition, the table reveals that BinStructs took the most amount of time to send, whereas doubles took the least.

- **Receiver-side performance:** The receiver-side results\(^3\) for sending primitive data types indicate that most of the run-time costs are incurred by (1) the TypeCode interpreter (TypeCode::traverse), (2) the CDR methods that retrieve the value from the incoming data (e.g., get_long and get_short), (3) the method deep_free that deallocates memory, and (4) the CDR::Decoder method. For BinStructs, the receiver spends a significant amount of time traversing the BinStruct::TypeCode (struct.traverse) and calculating the size and alignments of each member in the struct.

The receiver's run-time costs affect the sender adversely by increasing the time required to perform write system calls to the network. This happens due to the transport protocol flow control enforced by the receiving side, which cannot keep pace with the sender due to the excessive presentation layer overhead.

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

\[^3\]Throughput measurements from the receiver-side were nearly identical to the sender measurements and are not presented here.

![Figure 7: Throughput for the Original IIOP Implementation](image_url)

Tables 2 and 3 illustrate the receiver is the principal performance bottleneck. Therefore, our optimizations are designed to improve receiver performance. Since receiver is the bottleneck, we only show the profile measurements for the receiver.

3.2.3 Optimization Step 1: Inlining to Optimize for the Expected Case

- **Problem:** invocation overhead for small, frequently called methods: This subsection describes an optimization to improve the performance of IIOP receivers. We applied Principle 1 from Table 1, which optimizes for the expected case. Table 3 illustrates that depending on the data type, the appropriate get method of the CDR class must be invoked to retrieve the data from the incoming stream into a local copy. Since the get methods are always invoked they are prime targets for our first optimization step.

- **Solution:** inline method calls: Our solution to reduce invocation overhead for small, frequently called methods was to designate these methods as inline, using C++ language features. The throughput measurements after inlining the get methods are shown in Figure 8. Figures 9 and 10 illustrate the effect of inlining on the throughput of doubles and BinStructs. Figures 9 and 10 also compare the new results with the original results.

- **Optimization results:** After inlining, the new throughput results indicate only a marginal (i.e., 3-4%) increase in performance. Table 4 and 5 shows profiling measurements for the sender and receiver, respectively. The analysis of overhead for the sender-side receiver. The analysis of over-
Table 2: Sender-side Overhead in the Original IIOP Implementation

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Method Name</th>
<th>msec</th>
<th>Called</th>
<th>%</th>
</tr>
</thead>
<tbody>
<tr>
<td>double</td>
<td>write</td>
<td>78,051</td>
<td>512</td>
<td>93.33</td>
</tr>
<tr>
<td></td>
<td>put_longlong</td>
<td>2,250</td>
<td>8,388,608</td>
<td>2.99</td>
</tr>
<tr>
<td></td>
<td>cdr::encoder</td>
<td>1,605</td>
<td>8,388,016</td>
<td>1.92</td>
</tr>
<tr>
<td></td>
<td>TypeCode::traverse</td>
<td>1,000</td>
<td>1,024</td>
<td>1.55</td>
</tr>
<tr>
<td>long</td>
<td>write</td>
<td>134,141</td>
<td>512</td>
<td>92.92</td>
</tr>
<tr>
<td></td>
<td>put_long</td>
<td>3,799</td>
<td>16,780,238</td>
<td>2.63</td>
</tr>
<tr>
<td></td>
<td>cdr::encoder</td>
<td>3,103</td>
<td>16,781,284</td>
<td>2.47</td>
</tr>
<tr>
<td></td>
<td>TypeCode::traverse</td>
<td>2,998</td>
<td>1,024</td>
<td>1.80</td>
</tr>
<tr>
<td>short</td>
<td>write</td>
<td>265,392</td>
<td>512</td>
<td>93.02</td>
</tr>
<tr>
<td></td>
<td>put_short</td>
<td>7,993</td>
<td>33,554,432</td>
<td>2.66</td>
</tr>
<tr>
<td></td>
<td>cdr::encoder</td>
<td>6,598</td>
<td>33,559,040</td>
<td>2.31</td>
</tr>
<tr>
<td></td>
<td>TypeCode::traverse</td>
<td>5,195</td>
<td>1,024</td>
<td>1.82</td>
</tr>
<tr>
<td>octet</td>
<td>write</td>
<td>530,134</td>
<td>512</td>
<td>93.43</td>
</tr>
<tr>
<td></td>
<td>cdr::encoder</td>
<td>15,986</td>
<td>67,113,472</td>
<td>2.82</td>
</tr>
<tr>
<td></td>
<td>cdr::byte</td>
<td>10,291</td>
<td>67,118,000</td>
<td>1.83</td>
</tr>
<tr>
<td></td>
<td>TypeCode::traverse</td>
<td>10,388</td>
<td>1,024</td>
<td>1.83</td>
</tr>
<tr>
<td>BinStruct</td>
<td>write</td>
<td>588,039</td>
<td>512</td>
<td>88.65</td>
</tr>
<tr>
<td></td>
<td>get_long</td>
<td>19,846</td>
<td>44,083,504</td>
<td>2.99</td>
</tr>
<tr>
<td></td>
<td>calc_nested.size...</td>
<td>11,499</td>
<td>14,683,648</td>
<td>1.73</td>
</tr>
<tr>
<td></td>
<td>cdr::encoder</td>
<td>10,394</td>
<td>31,461,888</td>
<td>1.57</td>
</tr>
<tr>
<td></td>
<td>TypeCode::traverse</td>
<td>8,803</td>
<td>4,195,328</td>
<td>1.33</td>
</tr>
</tbody>
</table>

Table 3: Receiver-side Overhead in the Original IIOP Implementation

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Method Name</th>
<th>msec</th>
<th>Called</th>
<th>%</th>
</tr>
</thead>
<tbody>
<tr>
<td>double</td>
<td>TypeCode::traverse</td>
<td>2,598</td>
<td>1,539</td>
<td>23.88</td>
</tr>
<tr>
<td></td>
<td>cdr::get_longlong</td>
<td>2,596</td>
<td>8,388,608</td>
<td>23.80</td>
</tr>
<tr>
<td></td>
<td>deep::free</td>
<td>1,645</td>
<td>8,388,608</td>
<td>15.10</td>
</tr>
<tr>
<td></td>
<td>cdr::decoder</td>
<td>1,551</td>
<td>8,395,797</td>
<td>14.21</td>
</tr>
<tr>
<td></td>
<td>read</td>
<td>1,146</td>
<td>1,866</td>
<td>10.51</td>
</tr>
<tr>
<td>long</td>
<td>TypeCode::kind</td>
<td>799</td>
<td>8,389,120</td>
<td>7.31</td>
</tr>
<tr>
<td></td>
<td>cdr::get_longlong</td>
<td>5,194</td>
<td>1,539</td>
<td>25.31</td>
</tr>
<tr>
<td></td>
<td>cdr::get_PTT</td>
<td>4,956</td>
<td>16,783,379</td>
<td>22.40</td>
</tr>
<tr>
<td></td>
<td>deep::free</td>
<td>3,295</td>
<td>16,784,241</td>
<td>16.06</td>
</tr>
<tr>
<td></td>
<td>cdr::decoder</td>
<td>3,099</td>
<td>16,784,405</td>
<td>15.10</td>
</tr>
<tr>
<td></td>
<td>read</td>
<td>1,682</td>
<td>2,274</td>
<td>8.40</td>
</tr>
<tr>
<td>short</td>
<td>TypeCode::traverse</td>
<td>1,598</td>
<td>16,777,728</td>
<td>7.79</td>
</tr>
<tr>
<td></td>
<td>cdr::get_short</td>
<td>10,387</td>
<td>1,539</td>
<td>27.22</td>
</tr>
<tr>
<td></td>
<td>deep::free</td>
<td>9,188</td>
<td>33,554,432</td>
<td>24.07</td>
</tr>
<tr>
<td></td>
<td>cdr::decoder</td>
<td>9,194</td>
<td>33,559,615</td>
<td>17.27</td>
</tr>
<tr>
<td></td>
<td>TypeCode::kind</td>
<td>3,196</td>
<td>33,554,944</td>
<td>8.37</td>
</tr>
<tr>
<td>octet</td>
<td>TypeCode::traverse</td>
<td>20,773</td>
<td>1,539</td>
<td>29.30</td>
</tr>
<tr>
<td></td>
<td>cdr::decoder</td>
<td>13,984</td>
<td>67,118,003</td>
<td>19.71</td>
</tr>
<tr>
<td></td>
<td>deep::free</td>
<td>13,182</td>
<td>67,109,889</td>
<td>18.59</td>
</tr>
<tr>
<td></td>
<td>cdr::get_byte</td>
<td>10,787</td>
<td>67,118,113</td>
<td>15.22</td>
</tr>
<tr>
<td></td>
<td>TypeCode::kind</td>
<td>6,391</td>
<td>67,109,376</td>
<td>9.02</td>
</tr>
<tr>
<td>BinStruct</td>
<td>get_long</td>
<td>35,021,118</td>
<td>85,921,421</td>
<td>27.65</td>
</tr>
<tr>
<td></td>
<td>struct_traverse</td>
<td>23,041,20</td>
<td>29,270,880</td>
<td>18.31</td>
</tr>
<tr>
<td></td>
<td>calc_nested.size...</td>
<td>15,154,42</td>
<td>4,199,304</td>
<td>11.94</td>
</tr>
<tr>
<td></td>
<td>struct_traverse</td>
<td>10,406,70</td>
<td>33,561,621</td>
<td>8.21</td>
</tr>
<tr>
<td></td>
<td>TypeCode::traverse</td>
<td>10,401</td>
<td>6,092,995</td>
<td>8.20</td>
</tr>
<tr>
<td></td>
<td>deep::free</td>
<td>6,492</td>
<td>14,681,080</td>
<td>5.11</td>
</tr>
<tr>
<td></td>
<td>cdr::get_byte</td>
<td>6,394</td>
<td>33,566,720</td>
<td>5.94</td>
</tr>
<tr>
<td></td>
<td>cdr::decoder</td>
<td>3,399</td>
<td>21,153,513</td>
<td>2.88</td>
</tr>
</tbody>
</table>

Table 4: Sender-side Overhead After Applying the First Optimization (lining)

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Method Name</th>
<th>msec</th>
<th>Called</th>
<th>%</th>
</tr>
</thead>
<tbody>
<tr>
<td>double</td>
<td>write</td>
<td>86,190</td>
<td>512</td>
<td>93.08</td>
</tr>
<tr>
<td></td>
<td>cdr::encoder</td>
<td>2,254</td>
<td>8,383,216</td>
<td>3.78</td>
</tr>
<tr>
<td></td>
<td>TypeCode::traverse</td>
<td>1,000</td>
<td>1,024</td>
<td>1.51</td>
</tr>
<tr>
<td>BinStruct</td>
<td>write</td>
<td>436,694</td>
<td>512</td>
<td>85.00</td>
</tr>
<tr>
<td></td>
<td>calc_nested.size...</td>
<td>13,195</td>
<td>14,683,648</td>
<td>2.90</td>
</tr>
<tr>
<td></td>
<td>cdr::encoder</td>
<td>14,213</td>
<td>31,461,888</td>
<td>2.77</td>
</tr>
</tbody>
</table>

3.2.4 Optimization Step 2: Aggressive Inlining to Optimize for the Expected Case

- Problem: lack of C++ compiler support for aggressive inlining: The second optimization step continues with Principle 1 (optimizing for the expected case). Our Quantify results in Section 3.2.3 reveal that supplying the inline keyword to the compiler does not always work since the compiler occasionally ignores this "hint." Likewise, Table 5 reveals that inlining some methods may "uninline" others.

4The cdr::ptr::align::binary is a static method that aligns a given address at the specified alignment.

- Solution: replace inline methods with preprocessor macros: In our second optimization step, therefore, we employ a more aggressive inlining strategy. Methods like cdr::ptr::align::binary, which became unlined, are now forcibly inlined by defining them as preprocessor macros instead of as C++ inline methods.

The Sun C++ compiler did not include certain methods (such as cdr::ptr::align::binary) due to their size. For instance, the code in method cdr::ptr::align::binary swaps 16 bytes in an manually un-rolled loop if the arriving data was in a different byte order. This increases the size of the code, which caused the C++ compiler to ignore the inline keyword.

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

- **Optimization results:** The results of our aggressive inlining are shown in Figure 11. A comparison of throughput obtained for this optimization step with that of step 1 is shown in Figure 12 (for doubles) and Figure 13 (for BinStructs). The results reveal that the performance for sending doubles remains almost the same as that obtained in Section 3.2.3. The results for sending BinStructs reveal that performance degrades and becomes comparable to the unoptimized original SunSoft iop implementation.

To understand this behavior, we observe the profiling measurements shown in Table 6 for the receiver. The sender once again spends most of its run-time doing network writes. The receiver-side Quantify profile output reveals that aggressive inlining has indeed worked. But this inlining increases the code size for other methods (such as calc_nested.size and alignment, struct.traverse, CDR::decoder), thereby increasing their run-time costs. As shown in Figures 4 and 5, these methods are called a large number of times (indicated in Table 6).

Certain SunSoft iop Methods such as CDR::decoder and TypeCode::traverse are large and general-purpose.Inlining the methods described above causes further code bloat for these methods. Therefore, calling each other a large number of times results in a very high method call overhead. In addition, due to their large size, it is unlikely that code for both these methods can be resident in the processor cache at the same time, which further degrades performance.

In summary, although the second optimization step does not improve performance it is a necessary step since it reveals the actual sources of overhead in the code, as explained in Sections 3.2.5 and 3.2.6.

### 3.2.5 Optimization Step 3: Precomputing, Adding Redundant State, and Passing Information Through Layers

- **Problem: too many method calls:** The aggressive inlining optimization in Section 3.2.4 caused a slight degradation in performance, due primarily to processor cache effects. Table 6 reveals that for sending structs, the high cost methods are calc_nested.size and alignment, CDR::decoder, and struct.traverse. These methods are invoked a substantial number of times (29,367,801, 33,554,437, and 4,194,303 times, respectively) to process incoming requests.

To understand why so many method calls were made, we analyzed the calls to the method struct.traverse. The TypeCode interpreter invoked the struct.traverse method 2,097,152 times for data transmissions of 64 Mbytes in sequences of 32 bytes BinStructs. In addition, the TypeCode interpreter calculated the size of BinStruct, which called struct.traverse internally for every BinStruct. This accounted for an additional 2,097,152 calls.

Although inlining did not improve performance, it set the stage to answer the key question of why these high cost methods were invoked so frequently. As shown in Figure 5, and in the explanation in Section 2.2, we recognized that to demarshal an incoming sequence of BinStructs, the receiver’s TypeCode interpreter (TypeCode::traverse) must traverse each of its members using the method struct.traverse. As each member is traversed, this method determines the member’s size and alignment requirements using the calc_nested.size and alignment method. Each call to the calc_nested.size and alignment method can invoke the CDR::decoder method, which may invoke the traverse method.

Close scrutiny of the CORBA request datapath shown in Figure 5 reveals that the struct.traverse method calculates the size and alignment requirements each time it is invoked. As shown above, this yields a very large number of method calls for large streams of data.

- **Solution 1: reduce obvious waste by precomputing values and storing additional state:** At this point, we made a crucial observation: for incoming sequences, the TypeCode of each element is the same. With this observation, we can pinpoint the obvious waste (Principle 2 from Table I), which involves calculating the size and alignment requirements of each member (see Figures 4 and 5). In our experiments, the methods calc_nested.size and alignment and
struct_traverse are expensive. Therefore, optimizing them is crucial.

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

- **Solution 2:** converts generic methods into special-purpose, efficient ones: To further reduce method call overhead and to decrease the potential for processor cache misses, we moved the struct_traverse logic for handling t.k.structs into the traverse method. This optimization illustrates an application of Principle 3, which converts generic methods into special-purpose, efficient ones. We chose to keep the traverse method generic, yet make it efficient since we want our demarshaling engine to remain in the cache.

- **Analysis and remedies:** Finally, we infer the following drawbacks and suggest solutions to them:

  1. *Avoiding duplication:* Whenever the cdr::decode is demarshaling a sequence of structs, it must retrieve the TypeCode and size of the element type. When the decoder invokes the TypeCode interpreter, the same information is retrieved once again. This duplication is wasteful and must be eliminated (while maintaining correctness).

    To solve this problem, we used C++'s "default parameter" feature to pass the information from the decoder to the traverse method. This is an example of Principle 6, which recommends passing information between layers. To accomplish this, we modified the signature of the traverse method to use two additional void* parameters at the end. Each parameter has a default value of NULL, so it does not affect the existing implementation.

    For the t.k.sequence case, the decoder passes a pointer to the element's TypeCode and a pointer to the size information via the additional parameters of method traverse. Inside the traverse logic for t.k.sequences, we check if the two parameters are non-NULL and retrieve the necessary information accordingly. Note that this approach is also reentrant, which is important since our IOIP implementation must run in multi-threaded applications.

  2. *Avoid unnecessary function calls:* Whenever the TypeCode interpreter is passed a sequence of structs, it invokes the cdr::decode to demarshaling the struct. However, this method simply reinvokes the interpreter without processing the struct. This additional method call overhead is wasteful and unnecessary.

    To handle this second problem, we inserted a special-case check inside the traverse method for t.k.sequences to see if the element type is a struct or an array. If it is, we invoke the traverse method directly to handle the element. Although this is still a recursive call, performance improves since the code for traverse is likely to remain in the instruction cache.

- **Optimization results:** The throughput measurements recorded after incorporating these optimizations are shown in Figure 14. Figures 15 and 16 illustrate the benefits of the optimizations from step 3 by comparing the throughput obtained for doubles and BinStructs, respectively, with those from the previous optimization steps.

  The results illustrate that there is no change in the performance of any of the primitive types. This is expected since no wasteful (recursive) method call overhead is incurred for these types. However, for sending BinStructs, the throughput is now twice the performance of the previous results. This increase is primarily due to eliminating the overhead of the calc_nested.size_and_alignment method, which accounts for almost 30% of the runtime costs prior to these optimizations.

  Table 7 depicts the profiling measurements for the receiver. The sender continues to spend the most of its execution time performing network write calls. The methods in the receiver that account for most of the execution time for doubles include traverse, decoder, and
3.2.6 Optimization Step 4: Eliminating Obvious Waste and Specializing Generic Methods

- **Problem: expensive no-ops for memory deallocation:**
  The optimizations described in Section 3.2.5 improve performance for sending BinStructs. Table 7 reveals the overhead of the `deep_free` method is significant for primitive data types. This method is similar to the `decoder` method that traverses the `TypeCode` and deallocates dynamic memory. The `deep_free` method has the same type signature as the `decoder` method. Therefore, it can use the `traverse` method to traverse the data structure and deallocate memory.

  Close scrutiny of the `deep_free` method indicates that memory must be freed for constructed data structures (such
as IDL sequences and structs). We observed that for sequences, if the element type is a primitive type, the deep_free method simply deallocates the buffer containing the sequence.

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

- **Solution 1: eliminate obvious waste:** To optimize this, we changed the deletion strategy for sequences so that the element's TypeCode is checked first. If it is a primitive type, the traversal is not done and the memory is deallocated directly.

- **Solution 2: specialize generic methods:** In addition to eliminating obvious waste, we also optimized the traverse method by reducing the number of calls it made to method other than itself. The goal is to improve processor cache affinity. This improvement in the traverse method was achieved by incorporating the encoder, decoder, and deep_free logic in the traverse method itself.

- **Optimization results:** The throughput measurements for the fourth optimization step is shown in Figure 3.2.6. The throughput comparison for this case with previous cases is shown in Figures 18 and 19. These results indicate that this final optimization step improves the performance of the original IOP implementation by a factor of 1.7 times for doubles, 2.56 times for longs, 3.75 times for shorts, 5 times for chars/octets, and around 4.75 times for BinStructs.
Figure 15: Throughput Comparison for Doubles After Applying the Third Optimization (precomputing and adding redundant state)

Figure 17: Throughput After the Applying the Fourth Optimization (getting rid of waste)

Figure 16: Throughput Comparison for Structs After Applying the Third Optimization (precomputing and adding redundant state)

Figure 18: Throughput Comparison for Doubles After Applying the Fourth Optimization (getting rid of waste)
Tables 8 and 9 illustrate the remaining high cost sender-side and receiver-side methods, respectively. The tables indicate that for primitive types, the cost of writing to the network and reading from the network becomes the primary contributor to the run-time costs. For BinStructs, the TypeCode interpreter still remains the dominant factor — accounting for almost 92% of the receiver side run-time costs. To further reduce the overhead of the interpreter, we are applying more sophisticated compiler-based optimizations. These include using flow analysis to identify dependencies in the code. Having dealt with the dependencies, it would be possible to obtain an optimal ILP (Integrated Layer Processing) implementation of the interpreter. An ILP-based implementation will reduce the excessive data manipulation operations which is essential for RISC based architectures.

3.3 Maintaining CORBA Compliance and Interoperability

The optimizations presented in the previous sections are useful only to the extent that we maintain CORBA compliance and can interoperate with IIOP-based ORBs. This subsection presents the throughput results obtained for running the same experiment with Visigenic’s VisiBroker for C++ client and the SunSoft IIOP server. Figures 20 and 21 illustrate the throughput measurements for the original SunSoft IIOP server and our highly optimized implementation, respectively.

4 Related Work

This section describes results from existing work on protocol optimization based on one or more of the principles mentioned in Table 1. In addition, we discuss related work on CORBA performance measurements and presentation layer marshaling.

4.1 Related Work Based on Optimization Principles

[4] describes a technique called header prediction that predicts the message header of incoming TCP packets. This technique is based on the observation that many members in the header remaining constant between consecutive packets. This observation led to the creation of a template for the expected packet header. The optimizations reported in [4] are based on Principle 1, which optimizes for the expected case and Principle 3, which is precompute, if possible. We present
4.2 Related Work Based on CORBA Performance Measurements

[8, 9, 10] show that the performance of CORBA implementations is relatively poor, compared to that of low-level implementations using C/C++. The primary source of ORB-level overhead stems from marshaling and demarshaling. [8] measures the performance of the static invocation interface. [9] measures the performance of the dynamic invocation interface and the dynamic skeleton interface. [10] measures performance of CORBA implementations in terms of latency and support for very large number of objects. However, these results were restricted to measuring the performance of communication between homogeneous ORBs. These tests do not measure the run-time costs of interoperability between ORBS from different vendors. In addition, these papers do not provide solutions to reduce these overheads. In contrast, we have provided solutions that significantly improve performance by reducing marshaling/demarshaling overhead.

4.3 Related Work Based on Interpretive and Compiled Forms of Marshaling

[11] describes the tradeoffs of using compiled and interpreted marshaling schemes. Although compiled stubs are faster, they are also larger. In contrast, interpretive marshaling is slower, but smaller in size. [11] describes a hybrid scheme that combines compiled and interpretive marshaling to achieve better performance. This work was done in the context of the ASN.1/BER encoding [12].

According to the SunSoft IIOP implementors, interpretive marshaling is preferable since it decreases code size and increases the likelihood of remaining in the processor cache. Our measurements do not compare the SunSoft IIOP interpretive marshaling scheme with a compiled marshaling scheme. As explained in Section 5, we are currently implementing a CORBA IDL compiler that can generate compiled stubs and skeletons. Our goal is to generate efficient stubs and skeletons by extending optimizations provided in USC [16].

5 Concluding Remarks

This paper illustrates the benefits of applying principle-based optimizations [21] that improve the performance of the SunSoft CORBA Inter-ORB Protocol (IIOP) reference implementation. The principles that directed our optimizations included: (1) optimizing for the expected case, (2) eliminating obvious 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, and (6) passing information between layers.

The results of applying these optimization principles to SunSoft IIOP improved its performance 1.8 times for doubles, 3.3 times for longs, 3.75 times for shorts, 5 times for chars/octets, and 4.2 times for richly-typed
structs over ATM networks. Our optimized implementation is now competitive with existing commercial ORBs [8, 10] using the static invocation interface (SII) and 2 to 4.5 times (depending on the data type) faster than commercial ORBs using the dynamic skeleton interface (DSI) [9]. In addition, we show that our optimized implementation of IIOP interoparates seamlessly with Visigenic’s VisiBroker for C++ ORB which is a commercially available ORB.

In addition to our optimizations, we are currently enhancing the SunSoft IIOP implementation to form a complete high-performance ORB [18]. This involves extending the SunSoft CORBA IDL compiler to generate optimized stubs and skeletons from IDL interfaces. These generated stubs and skeletons will transform C++ methods into/from CORBA requests via our optimized IIOP implementation. In addition, we are incorporating an Object Adapter into our IIOP implementation that supports de-layered request demultiplexing. This ORB will be used to compare the impact of using compiled marshaling stubs and skeletons vs. the interpretive scheme currently implemented in SunSoft IIOP. We plan to measure the tradeoffs of using the two marshaling schemes to achieve an optimal hybrid solution [11].

References


A The CORBA General Inter-ORB Protocol and Internet Inter-ORB Protocol

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

A.1 Common Data Representation (CDR)

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

- **Variable byte ordering**: The sender encodes the data using its native byte-order. Special flag values in the encoded stream indicate the byte order used. Thus, only receivers with byte ordering different from the sender are required to perform byte swapping to retrieve correct binary values.

- **Aligned Types**: Primitive OMG IDL data types are aligned on their "natural" boundaries within GIOP messages, as shown in Table 10. Constructed data types (such as IDL sequence and struct) have no additional alignment restrictions beyond those of their primitive types i.e., the size and alignment of the constructed type will depend on the size and alignment of the primitives that make up constructed type.

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

- **CDR Encapsulations**: A CDR encapsulation is a sequence of octets. Encapsulations are typically used to marshal parameters of the following types:
  - **Complex TypeCodes** — see Table 11.
  - **IIOP protocol profiles inside interoperable object references (IOR)** — A protocol profile provides information about the transport protocol that enables client applications to talk to the servers. In the IIOP profile (Figure 22), this information consists of the host name and port number on which the server is listening, and the object_key of the target object implemented by that server. An IOR (see Figure 23) represents a complete information about an object. This information includes the type it represents, the protocols it supports, whether it is null or not, and any ORB related services that are available.

### Table 10: Alignment for OMG Primitive Types in Bytes

<table>
<thead>
<tr>
<th>Type</th>
<th>Alignment</th>
</tr>
</thead>
<tbody>
<tr>
<td>char</td>
<td>1</td>
</tr>
<tr>
<td>octet</td>
<td>1</td>
</tr>
<tr>
<td>short</td>
<td>2</td>
</tr>
<tr>
<td>long</td>
<td>4</td>
</tr>
<tr>
<td>unsigned short</td>
<td>2</td>
</tr>
<tr>
<td>unsigned long</td>
<td>4</td>
</tr>
<tr>
<td>float</td>
<td>4</td>
</tr>
<tr>
<td>double</td>
<td>8</td>
</tr>
<tr>
<td>boolean</td>
<td>1</td>
</tr>
<tr>
<td>enum</td>
<td>4</td>
</tr>
</tbody>
</table>

### Table 11: TypeCode Enum Values, Parameter List Types, and Parameters

- **Service-specific contexts** — The COSS [14] specification defines a number of services such as the transaction service. For interoperability, it may be required to pass service-specific information via opaque parameters. This is achieved using the service-specific context.

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

### A.2 GIOP Message Formats

The GIOP specification defines seven types of messages. Each message is assigned a unique value. The originator of a giop message can be a client and/or a server. Table 12 depicts the seven types of messages and the permissible originators of these messages.

A GIOP message begins with a GIOP header (Figure 24), followed by one of the message types (Figure 25), and finally the body of the message, if any.
A.3 GIOP Message Transport

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

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

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

A.4 Internet Inter-ORB Protocol (IIOP)

The IIOP is a specialized mapping of GIOP onto the TCP/IP protocols. ORBs that use IIOP can communicate with other ORBs that publish their TCP/IP addresses as interoperable object references (IORs). IIOP IOR profiles are shown in Figure 22. Figure 23 shows the format of an IOR. When IIOP is used, the profile data member of the TaggedProfile structure holds the profile data of IIOP shown in Figure 22.
B TTCP IDL description

The following CORBA IDL interface was used in our experiments to measure the throughput of SunSoft IIOP:

```idl
typedef sequence<BinStruct> StructSeq;

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

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

```idl
// BinStruct is 32 bytes (including padding).
struct BinStruct
{
    short s; char c; long l;
    octet o; double d; octet pad[8]
};
```

```idl```
// Richly typed data.
interface tcp_throughput
{
    typedef sequence<short> ShortSeq;
    typedef sequence<long> LongSeq;
    typedef sequence<double> DoubleSeq;
    typedef sequence<char> CharSeq;
    typedef sequence<octet> OctetSeq;
}