Principles for Optimizing CORBA Internet Inter-ORB Protocol Performance

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[1]

This paper appeared in the Proceedings of the HICSS conference, Maui, Hawaii, January 9th, 1998. It was selected as the best paper in the Software Technology Track (188 submitted, 77 accepted).

Abstract

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

This paper provides two contributions to the study and design of efficient IIOP implementations. First, we pinpoint the key sources of overhead in the SunSoft IIOP implementation, which is a publically available implementation of IIOP written in C++, and measure its performance for transferring richly-typed data over a high-speed ATM network. Second, we demonstrate the empirical benefits of systematically optimizing SunSoft IIOP by optimizing for the common case; eliminating gratuitous waste; replacing general purpose methods with specialized, efficient ones; precomputing values; storing redundant state to speed up expensive operations; passing information between layers; and optimizing for better processor cache performance.

The results of optimizing SunSoft IIOP improved its performance substantially for all data types. The resulting IIOP implementation is competitive with existing commercial ORBs using CORBA's static invocation interface and 2 to 4.5 times (depending on the data type) faster than commercial ORBs using the dynamic skeleton interface. We have integrated the optimized IIOP implementation into TAO, which is a CORBA ORB targeted for real-time systems.

[1] This work was supported in part by NSF grant NCR-9628218.

1 Motivation

An increasingly important class of distributed applications require high bandwidth and low latency. 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 meeting stringent performance and predictability requirements, 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) [21]. 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 known as the General Inter-ORB Protocol (GIOP). Specialized mappings of GIOP can be defined for particular transport protocols. One such mapping is called the Internet Inter-ORB Protocol (IIOP), which is the standard GIOP mapping IIOP is the standard for interoperability for distributed object computing over TCP/IP.

[9, 10, 11] show that the performance of conventional CORBA middleware implementations is relatively poor compared to lower-level implementations using sockets and C/C++ since the ORBs incur a significant amount of data copying, marshaling, demarshaling, and demultiplexing overhead. These results, however, focused entirely on the communication performance between *homogeneous* ORBs. They do not measure the run-time costs of interoperability between heterogeneous ORBs. In addition, earlier work on measuring CORBA performance did not present the results of optimizations to reduce key sources of ORB overhead.

In this paper, we measure the performance of SunSoft IIOP using a CORBA/ATM testbed environment similar to [9, 10, 11]. SunSoft IIOP is a freely available implementation of the `iiop` protocol. We measure the performance of SunSoft IIOP and precisely pinpoint its performance overheads. In addition, we describe the results of systemati-

cally applying seven **principle-driven optimizations** [26] that substantially improve the performance of SunSoft IIOP. These optimizations include: *optimizing for the common case; eliminating gratuitous waste; replacing general purpose methods with specialized, efficient ones; precomputing values, if possible; storing redundant state to speed up expensive operations; passing information between layers; and optimizing for the processor cache*.

The results of applying these optimization principles to SunSoft IIOP improved its performance 1.9 times for `doubles`, 3.3 times for `longs`, 4 times for `shorts`, 5 times for `chars/octets`, and 6.7 times for richly-typed `structs` over ATM networks. Our optimized version of SunSoft IIOP is now comparable to existing commercial ORBs [9, 10, 11] 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. The protocol optimizations described in this paper are based on a set of principles that have been used to improve the performance of communication protocols described in Section 4. These principles can be applied to improve the performance of other ORB implementations and distributed object computing middleware.

This paper is organized as follows. Section 2 describes the CORBA reference model, the GIOP/IIOP interoperability protocols, and the SunSoft IIOP 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. The Appendices describe CORBA IIOP in greater depth and present a detailed view of SunSoft IIOP's protocol engine structure and runtime functionality.

2 Background

2.1 Overview of CORBA

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

• **Object location:** CORBA objects can be located locally with the client or remotely on a server, 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 Win32, UNIX, MVS, and real-time embedded systems like VxWorks, Chorus, and LynxOS.

• **Communication protocols and interconnects:** The communication protocols and interconnects that CORBA can run on include TCP/IP, IPX/SPX, FDDI, ATM, Ethernet, Fast Ethernet, 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 the transparency described above.

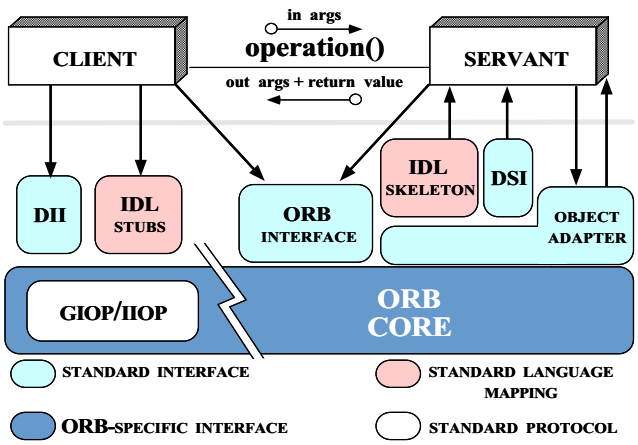


Figure 1: Components in the CORBA Reference Model

The components in CORBA include the following:

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

Client: This program entity performs application tasks by obtaining object references to servants and invoking operations on the servants. Servants can be remote or co-located relative to the client. Ideally, accessing a remote servant should be as simple as calling an operation on a local object, *i.e.,* `object->operation(args)`. Figure 1 shows the components that ORBs use to transmit requests transparently from client to servant for remote operation invocations.

ORB Core: When a client invokes an operation on a servant, the ORB Core is responsible for delivering the request to the servant and returning a response, if any, to the client. For servants executing remotely, a CORBA-compliant [21] ORB Core communicates 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. The ORB interface also creates argument lists for requests made through the dynamic invocation interface (DII) described below.

OMG IDL Stubs and Skeletons: IDL stubs and skeletons serve as the "glue" between the client and servants, respectively, and the ORB. Stubs provide a strongly-typed, *static* invocation interface (SII) that marshals application data into a common packet-level representation. Conversely, skeletons demarshal the packet-level representation back into typed data that is meaningful to an application. An IDL compiler automatically transforms OMG IDL definitions into an application programming language like C++ or Java. IDL compilers eliminate common sources of network programming errors and provide opportunities for automated compiler optimizations [7].

Dynamic Invocation Interface (DII): The DII allows a client to 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. The DII also allows clients to make *deferred synchronous* calls, which decouple the request and response portions of twoway operations to avoid blocking the client until the servant responds. In contrast, SII stubs only support twoway (*i.e.*, request/response) and oneway (*i.e.*, request only) 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 whether the server ORB 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 [22] define the Portable Object Adapter (POA), which supports multiple nested POAs per ORB.

2.2 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 protocol specification that can be mapped onto conventional connection-oriented transport protocols. An ORB is GIOP-compatible if it can send and receive GIOP messages.

The GIOP specification consists of the following elements:

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

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

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

The most common concrete mapping of GIOP onto the TCP/IP transport protocol is known as the Internet Inter-ORB Protocol (IIOP). The GIOP and IIOP specifications are described further in [21] and Appendix A. The remainder of this section presents an overview of the SunSoft IIOP reference implementation and outlines its primary components.

2.3 Overview of SunSoft IIOP

2.3.1 CORBA Features Supported by SunSoft IIOP

SunSoft IIOP is a freely available[2] implementation of an ORB Core that comprises an implementation of the IIOP version 1.0 protocol. SunSoft IIOP is written in C++ and provides many features of a CORBA ORB, except for an IDL compiler, an interface repository, and a complete Object Adapter. Therefore, users must manually generate the stubs, skeletons, and demultiplexing code that integrates with the C++ components 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. Since SunSoft IIOP does not provide an IDL compiler, client stubs using the SII API must be generated manually.

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 runtime. Requests are created and parameters marshaled using *e.g.* `Request`, `NVList`, `NamedValue`, and `TypeCode`, which are *pseudo-object* interfaces defined by CORBA. Pseudo-objects are entities that are neither CORBA primitive types nor constructed types. Operations on pseudo-object references cannot be invoked using the DII mechanism since the interface repository does not keep any information about them. In addition, only some pseudo-objects,

[2] See `ftp://ftp.omg.org/pub/interop/` for the source code.

such as TypeCodes, can be transferred as parameters to methods of an interface.

SunSoft IIOP supports dynamic skeletons via the dynamic skeleton interface (DSI). The DSI is used by applications (such as ORB bridges [21]) that have no compile-time knowledge of the interfaces they implement. Therefore, the DSI mechanism parses incoming requests, unmarshals their parameters, and demultiplexes requests to the appropriate object implementations.

Servers that use the SunSoft DSI mechanism must provide it with TypeCode information that is used to interpret incoming requests and demarshal the parameters. TypeCodes are CORBA pseudo-objects that describe the format and layout of primitive and constructed IDL data types in the incoming request stream.

2.3.2 The Sunsoft IIOP Component Architecture

The components in SunSoft IIOP are shown in Figure 2. The TypeCode marshaling/demarshaling protocol engine

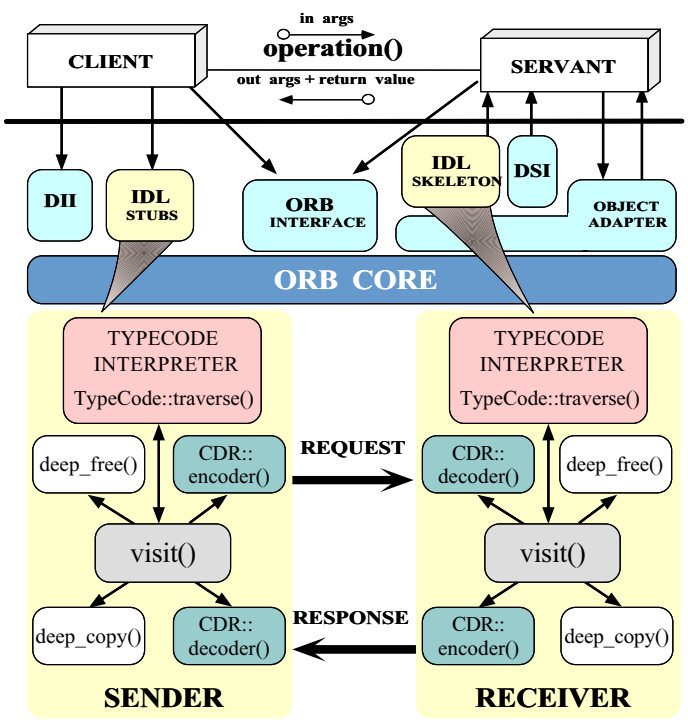


Figure 2: Components in the SunSoft IIOP Implementation

is the primary component of SunSoft IIOP. SunSoft IIOP's protocol engine uses an interpretive marshaling scheme that encodes or decodes parameter data by identifying their TypeCodes at run-time using the _kind field of each TypeCode object. A TypeCode is a pseudo-object that maintains an internal representation of the OMG IDL type information for each data type as it is marshaled and transmitted over a network.

The motivation for using an interpreter in SunSoft IIOP is to reduce the space utilization of the marshaling/demarshaling engine. Minimizing the ORB's memory footprint is important for embedded systems, such as PDAs and avionics systems. SunSoft IIOP requires less than 100 Kbytes on a real-time operating system like VxWorks. ORBs with small memory footprints can also be useful for general-purpose computing systems since this allows the protocol interpreter to fit entirely within a processor cache.

Each component of the SunSoft IIOP architecture is outlined below:

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

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

- **The CDR::encoder method:** The encoder method of the CDR class converts application data types from their native host representation into the CDR representation that transmits CORBA requests over a network.

- **The CDR::decoder method:** The decoder method of the CDR class is the inverse of the encoder method. It converts request values from the incoming CDR stream into the native host representation.

- **The deep_copy method:** The deep_copy method is used by the SunSoft DII mechanism to allocate storage and marshal parameters into the CDR stream using the TypeCode interpreter.

- **The deep_free method:** The deep_free method is used by the SunSoft DSI server to free 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, including:

- **The calc_nested_size_and_alignment method:** This method calculates the size and alignment of the fields comprising an IDL struct.

- **The struct_traverse method:** The TypeCode interpreter uses this method to recursively traverse the fields in an IDL struct.

Appendix C examines the run-time behavior of SunSoft IIOP by tracing the path taken by requests that transmit the sequence of BinStructs shown below:

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

// Richly typed data.
interface ttcp_throughput
{
  typedef sequence<BinStruct> StructSeq;
  // similarly for the rest of the types

  // Methods to send various data type sequences.
  oneway void sendStructSeq (in StructSeq ts);
  // similarly for rest of the types

};
```

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 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 Megabits per-sec (Mbps) SONET multimode fiber. The Maximum Transmission Unit (MTU) on the ENI ATM adaptor is 9,180 bytes. Each ENI card has 512 Kbytes of on-board memory. A maximum of 32 Kbytes is allotted per ATM virtual circuit connection for receiving and transmitting frames (for a total of 64 K). This allows up to eight switched virtual connections per card. The CORBA/ATM hardware platform is shown in Figure 3.

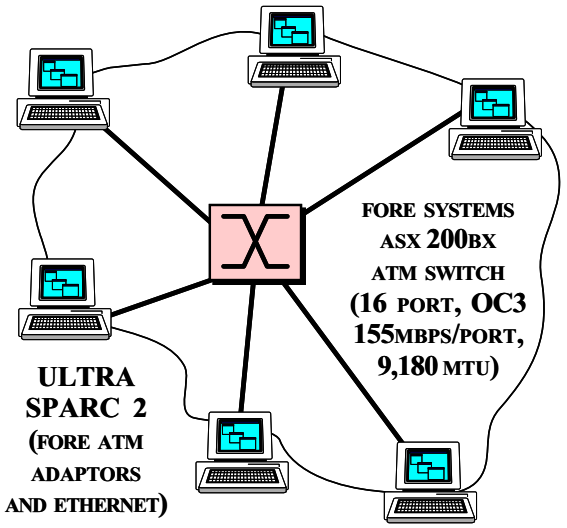


Figure 3: Hardware for the CORBA/ATM Testbed

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` [25] 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 client-side stubs use SunSoft IIOP's SII API, *i.e.*, the `do_call` method. On the server-side, the Object Adaptor uses a callback method supplied by the `ttcp` server application to dispatch incoming requests and their 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 uni-directional, with the transmitter flooding the receiver with a user-specified number of data buffers. Various sender and receiver parameters may be selected at run-time. These parameters include the 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. SunSoft IIOP transferred the data types using IDL `sequences`, which are dynamically-sized arrays. The IDL declaration is shown in Appendix B. The sender-side transmitted data buffer sizes of a specific data type incremented in powers of two, ranging from 1 Kbytes 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` [16] performance measurement tool. `Quantify` analyzes performance bottlenecks and identifies sections of code that dominate execution time. Unlike traditional sampling-based profilers (such as the UNIX `gprof` tool), `Quantify` reports results without including its own overhead. In addition, `Quantify` measures the overhead of system calls and third-party libraries without requiring access to source code.

All data is recorded in terms of machine instruction cycles and converted to elapsed times according to the clock rate of the machine. The collected data reflect the cost of the original program's instructions and automatically exclude any `Quantify` counting overhead.

Additional information on the run-time behavior of the code such as system calls made, their return values, signals,

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

3.2 Performance Results and Benefits of Optimization Principles

3.2.1 Methodology

CORBA implementations like SunSoft IIOP are representative of complex communication software. Optimizing such software is hard, particularly since seemingly minor "mistakes," such as excessive data copying or dynamic allocation, can reduce performance significantly [9]. Therefore, developing high performance and predictable ORBs requires an iterative, multi-step process. The first step involves measuring the performance of the system and pinpointing the sources of overhead. The second step involves a careful analysis of these sources of overhead and application of optimizations to remove them.

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

The optimizations described in this section are based on seven core principles for implementing protocols efficiently. [26] describes a collection of optimization principles in detail and illustrates how they have been applied in existing protocol implementations, *e.g.,* TCP/IP. This paper focuses on the principles in Table1 that we applied systematically to improve the performance of SunSoft IIOP. We focused on

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

Table 1: Summary of Principles for Efficient Protocol Implementations

these principles since we found them strategic to improve SunSoft IIOP performance. When describing our optimizations, we refer to these principles and explain how their use is

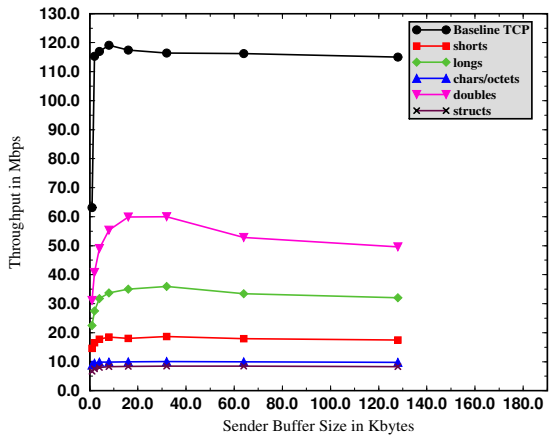


Figure 4: Throughput for the Original SunSoft IIOP Implementation

justified. Thus, we describe a *measurement-based, principle-driven* methodology to improve the performance of SunSoft IIOP.

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

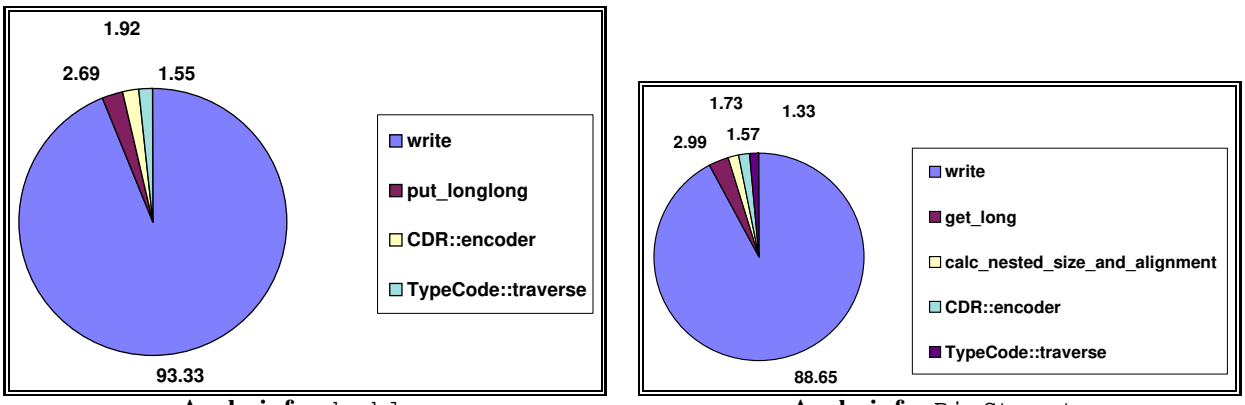
1. Aggressive inlining to optimize for the common case: which is discussed in Section 3.2.3;

2. Precomputing, adding redundant state, passing information through layers, eliminating gratuitous waste, and specializing generic methods: which is discussed in Section 3.2.4;

3. Optimizing for the processor cache: which is discussed in Section 3.2.5.

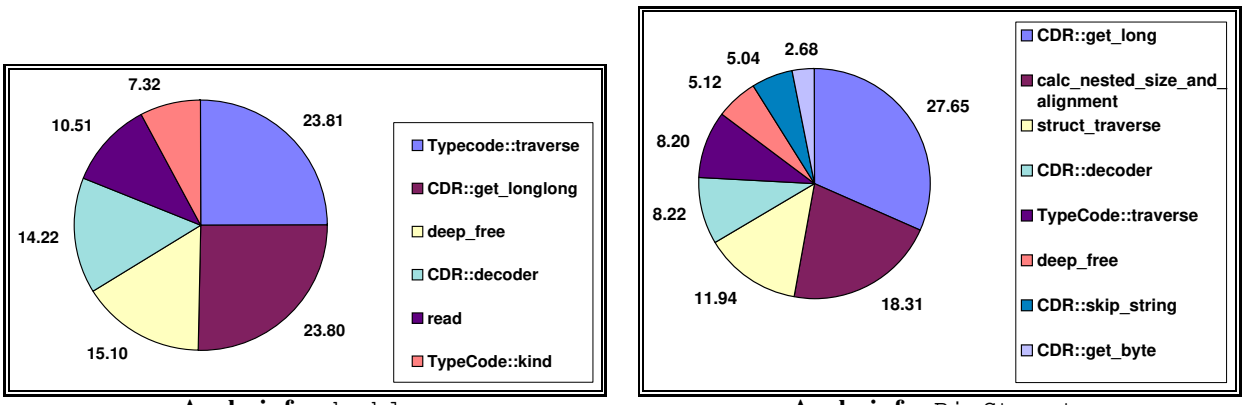
The order in which we applied the principles was based on the most significant sources of overhead identified empirically at that step and the principle(s) that most effectively reduced the overhead. For each step, we describe the principles and specific optimization techniques that were applied to reduce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

The comparisons focus on data types that exhibited the widest range of performance, *i.e.*, `double` and `BinStruct`. As shown below, the first optimization step does not significantly improve performance. However, this step is necessary since it reveals the actual sources of overhead, which are then alleviated by the optimizations in subsequent steps.



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

Figure 5: Sender-side Overhead in the Original IIOP Implementation



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

Figure 6: Receiver-side Overhead in the Original IIOP Implementation

3.2.2 Performance of the Original IIOP Implementation

Sender-side performance: Figure 4 illustrates the sender-side throughput obtained by sending 64 Mbytes of various data types for buffer sizes ranging from 1 Kbytes to 128 Kbytes (incremented by powers of two). The figure compares SunSoft IIOP with a hand-optimized baseline implementation that uses TCP/IP and sockets. These results indicate that different data types achieved substantially different levels of throughput.

The highest ORB throughput results from sending `doubles`, whereas `BinStructs` displayed the worst behavior. This variation in behavior stems from the marshaling and demarshaling overhead for different data types. In addition, the use of the interpretive marshaling/demarshaling engine in SunSoft IIOP incurs a large number of recursive method calls.

Detailed analysis using `Quantify` reveals that the sender spends ~90% of its run-time performing `write` system calls to the network. This overhead stems from the transport protocol flow control enforced by the receiving side, which cannot keep pace with the sender due to excessive presentation layer overhead.

Receiver-side performance: The `Quantify` analysis for the receiver-side are shown in Figure 6. The receiver-side results[3] for sending primitive data types indicate that most of the run-time costs are incurred by the following methods:

1. **The TypeCode interpreter:** *i.e.*, the `traverse` method in class `TypeCode`.

2. **The CDR methods that retrieve the value from the incoming data:** *e.g.*, `get_long` and `get_short`.

3. **The deep_free() method:** which deallocates memory.

4. **The CDR::decoder() method:** The receiver spends a significant amount of time traversing the `BinStruct` `TypeCode` (`struct_traverse`) and calculating the size and alignment of each member in the `struct`.

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

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

The remainder of this section describes the various optimization principles we applied to SunSoft IIOP, as well as the motivations and consequences of applying these optimizations.

Figures 6 illustrate the receiver is the principal performance bottleneck. Therefore, our initial set of optimizations are designed to improve receiver performance. Likewise, since the receiver is the bottleneck, we show the `Quantify` profile measurements only for it.

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

Problem: high 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 common case*. Figure 6 illustrates that the appropriate `get` method of the CDR class must be invoked to retrieve the data from the incoming stream into a local copy. For instance, depending on the data type, methods like `CDR::get_long` or `CDR::get_longlong` are called millions of times to decode the 64 Mbytes of data. Since these `get` methods are invoked quite frequently they are prime targets for our first optimization step.

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

Problem: lack of C++ compiler support for aggressive inlining: Our intermediate `Quantify` results after inlining reveal that supplying the `inline` keyword to the compiler does not always work since the compiler occasionally ignores this "hint." Likewise, inlining some methods may cause others to become "uninlined."

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

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

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

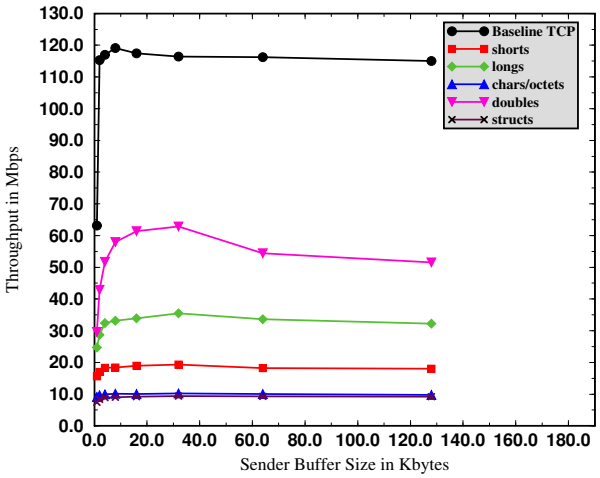


Figure 7: Throughput After Applying the First Optimization (inlining)

Optimization results: The throughput measurements after aggressive inlining are shown in Figure 7. Figures 8 and 9 illustrate the effect of inlining on the throughput of `doubles` and `BinStructs`. Figures 8 and 9 also compare the new results with the original results. After aggressive inlining, the new throughput results indicate only a marginal (*i.e.*, 4%) increase in performance. Figures 11 and 2 show profiling measurements for the sender and receiver, respectively. As before, the analysis of overhead for the sender-side reveals that most run-time overhead stems from `write` calls to the network.

Figure 10 illustrates the effect of aggressive inlining on the throughput of `doubles` and `BinStructs`. After aggressive inlining, the new throughput results indicate only a marginal (*i.e.*, 4%) increase in performance. Figure 2 shows profiling measurements for the receiver. The analysis of overhead for the sender-side reveals that, as before, most run-time overhead stems from `write` calls to the network.

The receiver-side `Quantify` profile output reveals that aggressive inlining does force operations to be inlined. However, this inlining increases the code size for other methods such as `struct_traverse`, `CDR::decoder`, and `calc_nested_size_and_alignment`, thereby increasing their run-time costs. As shown in Figures 39 and 40, these methods are called a large number of times (indicated in Figure 2).

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

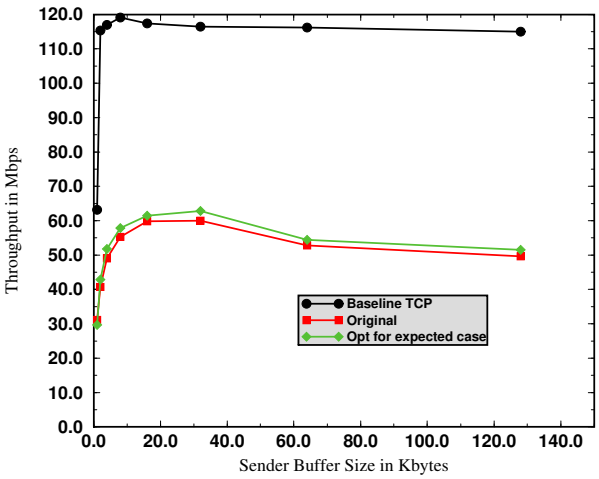


Figure 8: Throughput Comparison for Doubles After Applying the First Optimization (inlining)

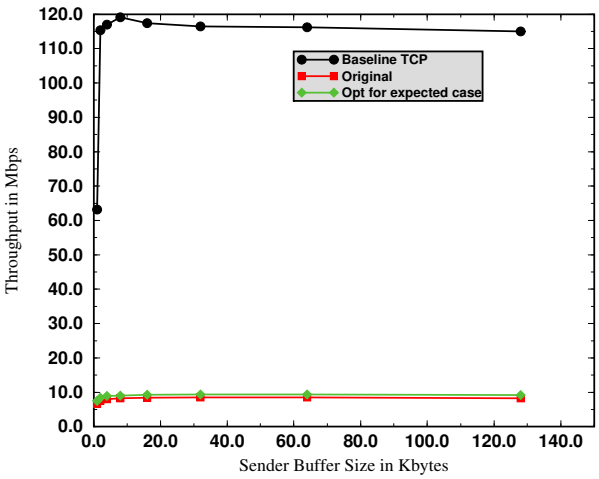


Figure 9: Throughput Comparison for Structs After Applying the First Optimization (inlining)

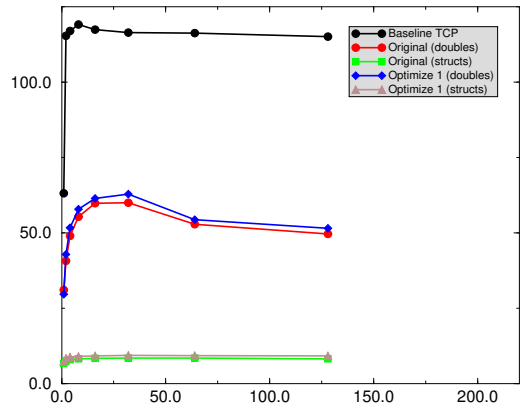


Figure 10: Throughput Comparison for Doubles and Structs After Applying the First Optimization (inlining)

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

Table 2: Receiver-side Overhead After Applying the First Optimization (aggressive inlining)

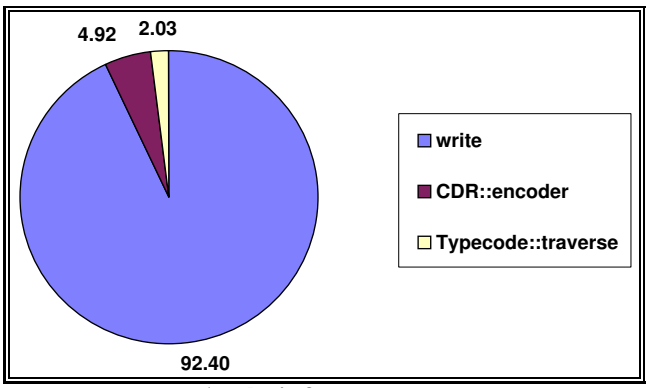
plains why inlining does not result in significant performance improvement.

In summary, although our first optimization step did not improve performance dramatically, it helped to reveal the actual sources of overhead in the code, as explained in Section 3.2.4.

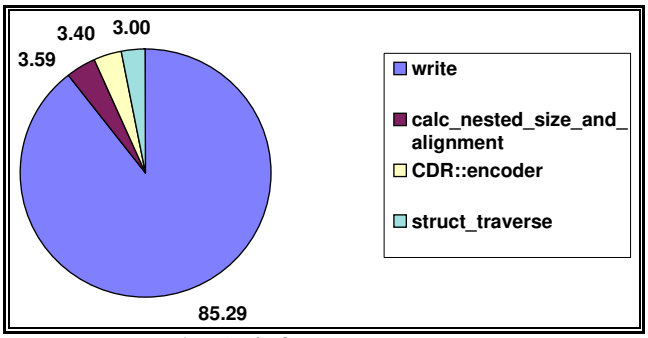
3.2.4 Optimization Step 2: Precomputing, Adding Redundant State, Passing Information Through Layers, Eliminating Gratuitous Waste, and Specializing Generic Methods

Problem: too many method calls: The aggressive inlining optimization in Section 3.2.3 did not cause substantial improvement in performance due to processor cache effects. Figure 2 reveals that for sending `structs`, the high cost methods are `calc_nested_size_and_alignment`, `CDR::decoder`, and `struct_traverse`. These methods are invoked a substantial number of times (29,367,801, 33,554,437, and 4,194,303 times, respectively) to process incoming requests.

To see why these methods were invoked so frequently, we analyzed the calls to `struct_traverse`. The `TypeCode` interpreter invoked `struct_traverse`



Analysis for `doubles`



Analysis for `BinStructs`

Figure 11: Sender-side Overhead After Applying the First Optimization (aggressive inlining)

2,097,152 times for data transmissions of 64 Mbytes in `sequences` of 32-byte `Binstructs`. In addition, the `TypeCode` interpreter calculated the size of `BinStruct` (using the `calc_nested_size_and_alignment` function), which called `struct_traverse` internally for every `BinStruct`. This accounted for an additional 2,097,152 calls.

Although inlining did not improve performance substantially, it helped to answer a key performance question: *why were these high cost methods invoked so frequently*? Based on our detail analysis of the SunSoft IIOP implementation (shown in Figure 40 and in the explanation in Section 2.3), we recognized that to demarshal an incoming `sequence` of `BinStructs`, the receiver's `TypeCode` interpreter method `TypeCode::traverse` must traverse each of its members using the method `struct_traverse`. As each member is traversed, the `calc_nested_size_and_alignment` method determines the member's size and alignment requirements. Each call to the `calc_nested_size_and_alignment` method can invoke the `CDR::decoder` method, which may invoke the `traverse` method.

Close scrutiny of the CORBA request datapath shown in Figure 40 reveals that the `struct_traverse` method calculates the size and alignment requirements every time it is invoked. As shown above, this yields a substantial number of method calls for large amounts of data.

Several solutions to remedy this problem are outlined below:

Solution 1: reduce gratuitous waste by precomputing values and storing additional state: The first solution is based on the following two observations. First, for incoming `sequences`, the `TypeCode` of each element is constant. Second, each `BinStruct` in the IDL `sequence` has the same fixed size. These observations enabled us to pinpoint a key source of *gratuitous waste* (Principle 2 from Table 1). In this case, the gratuitous waste involves recalculating the size and alignment requirements of each element of the `sequence`. In our experiments, the methods `calc_nested_size_and_alignment` and `struct_traverse` are expensive. Therefore, it is crucial to optimize them.

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

Solution 2: convert generic methods into special-purpose, efficient ones: To further reduce method call overhead, and to decrease the potential for processor cache misses, we moved the `struct_traverse` logic for handling `structs` into the `traverse` method. In addition, we introduced the `encoder`, `decoder`, `deep_copy`, and `deep_free` logic into the `traverse` method. This optimization illustrates an application of Principle 3 (*Convert generic methods into special-purpose, efficient ones*).

We chose to keep the `traverse` method generic, yet make it efficient since we want our demarshaling engine to remain in the cache. However, this scheme may not result in the best cache hit performance for machine architectures with small caches since the `traverse` method is excessively large. Section 3.2.5 describes optimizations we used to improve processor cache performance.

Problem: expensive no-ops for memory deallocation: Figure 2 reveals that the overhead of the `deep_free` method remains significant for primitive data types. This method is similar to the `decoder` method that traverses the `TypeCode` and deallocates dynamic memory. For instance, the `deep_free` method has the same type signature

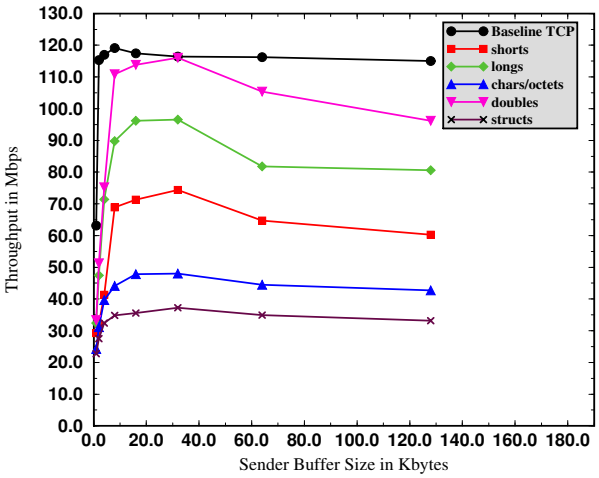


Figure 12: Throughput After Applying the Second Optimization (precomputation and eliminating waste)

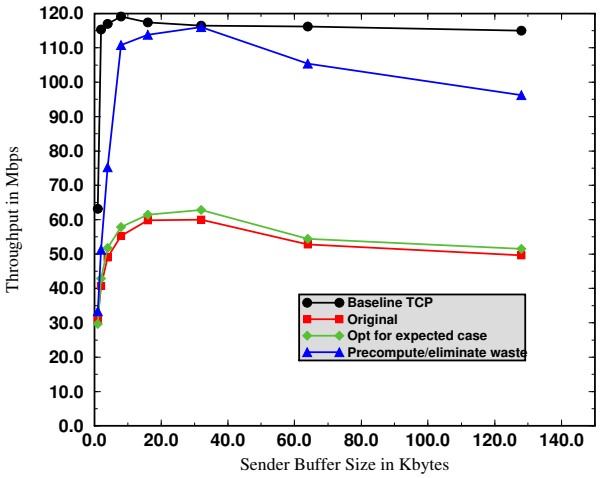


Figure 13: Throughput Comparison for Doubles After Applying the Second Optimization (precomputation and eliminating waste)

as the `decoder` method. Therefore, it can use the recursive `traverse` method to navigate the data structure and deallocate memory.

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

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

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

Optimization results: The throughput measurements recorded after incorporating these optimizations are shown in Figure 12. Figures 13 and 14 illustrate the benefits of the optimizations from step 2 by comparing the throughput obtained for `doubles` and `BinStructs`, respectively, with results from previous optimization steps.

Figure 15 illustrates the benefits of the optimizations from step 2 by comparing the throughput obtained for `doubles` and `BinStructs`, respectively, with those from the previous optimization steps. These results indicate that the second optimization step improves the performance of the original IIOP implementation substantially. The performance of

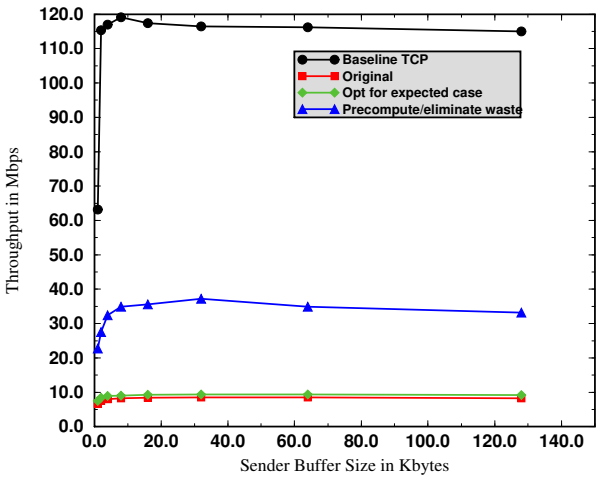


Figure 14: Throughput Comparison for Structs After Applying the Second Optimization (precomputation and eliminating waste)

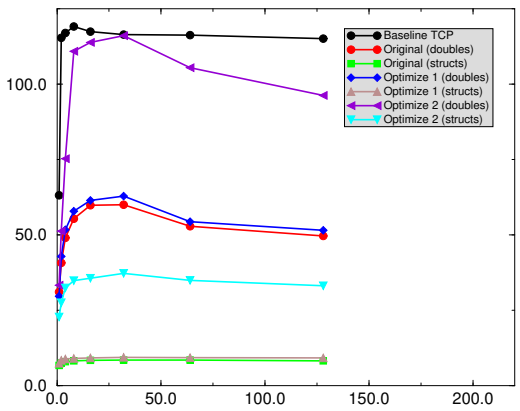


Figure 15: Throughput Comparison for Doubles and Structs After Applying the Second Optimization (precomputation and eliminating waste)

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

Table 3: Receiver-side Overhead After Applying the Second Optimization (getting rid of waste and precomputation)

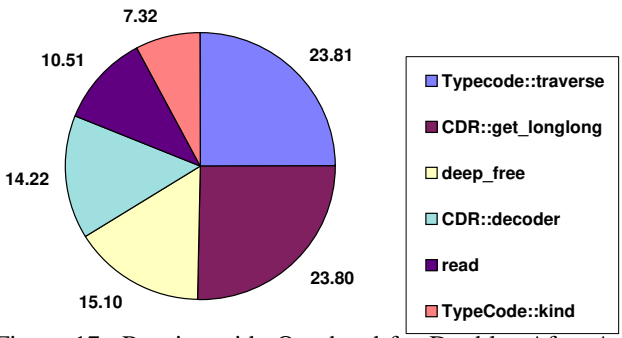


Figure 17: Receiver-side Overhead for Doubles After Applying the Second Optimizations

doubles increased by a factor of 1.9, longs by a factory of 3.3, shorts by a factor of 4, chars and octets by a factor of 5, and BinStructs by a factor of 6.7. The maximum throughput of 115 Mbps obtained for sending doubles is comparable to the baseline TCP/IP performance.

Figures 17 and 17 depict the profiling measurements for the receiver (the sender continues to spends most of its execution time performing network write calls). The receiver methods accounting for most execution time for doubles include traverse, decoder, and deep_free. For BinStructs, the run-time costs of the traverse method in the receiver increases significantly compared to the previous optimization steps. This is due primarily to the inclusion of the struct_traverse, encoder, and decoder logic. Although the run-time costs of the interpreter increased, the overall performance improved since the number of calls to functions other than itself decreased. As a result, this design improved processor cache affinity, which yielded better performance. In addition, due to precomputation, calc_nested_size_and_alignment method need not be called repeatedly.

Figures 16 and 3 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 source of run-time costs. This result represents a substantial improvement and illustrates that the marshaling overhead of IIOP need not be a limiting factor in ORB performance.

For BinStructs, the TypeCode interpreter still remains the dominant factor – accounting for over 90% of the receiver-side run-time costs. To further reduce the overhead of the interpreter, we are applying more sophisticated compiler-based optimizations, such as 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) [5] implementation of the interpreter. An ILP-based implementation can reduce the excessive data manipulation operations, which is essential to optimize RISC architectures.

Another technique to improve overall marshaling performance is to use a hybrid scheme [15] in which frequently transferred data types can be marshaled using fast, but large compiled stubs. In contrast, an interpretive scheme can be used to marshal data types that are seldom transferred. This hybrid scheme tries to achieve an optimal time and space tradeoff by using fast, but large compiled stubs as well as a slow, but compact interpreter.

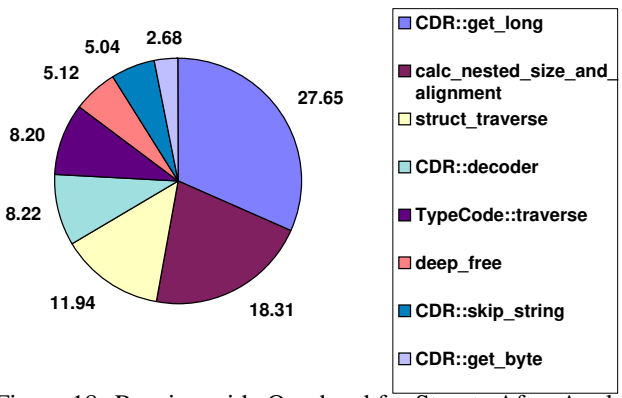
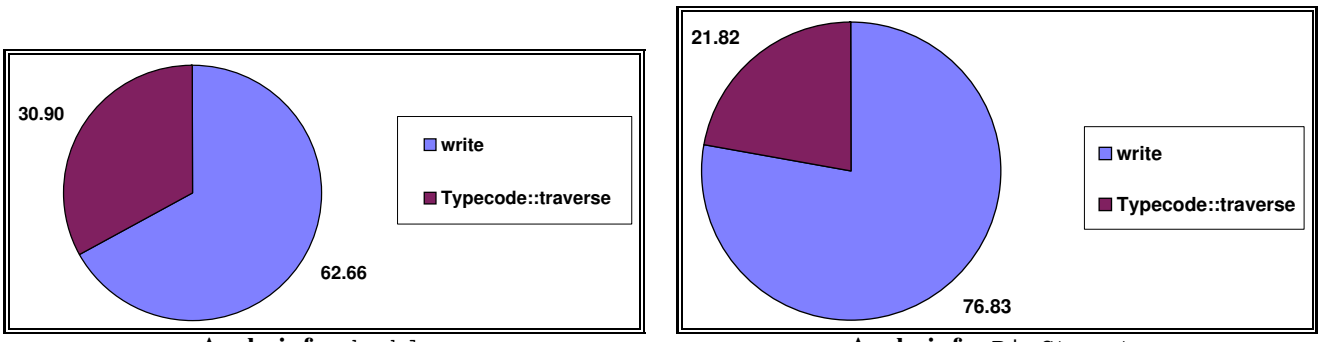


Figure 18: Receiver-side Overhead for Structs After Applying the Second Optimizations



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

Figure 16: Sender-side Overhead After Applying the Second Optimization (getting rid of waste and precomputation)

3.2.5 Optimization Steps 3 and 4: Optimizating for Processor Caches

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

Problem: Very large, monolithic interpreter: Section 3.2.4 describes optimizations based on precomputation, eliminating waste, and specializing generic methods. These optimizations yield an efficient, albeit excessively large, `TypeCode` interpreter. The efficiency stems from the fact that the monolithic structure results in low function call overhead. Recursive function calls are affordable since the processor cache is already loaded with the instructions for the same function. However, for machine architectures with smaller cache sizes, it may be desirable to have smaller functions.

Solution: Split large functions into smaller ones and outlining: This section describes optimizations we used to improve processor cache affinity for SunSoft IIOP. Our optimizations are based on two principles described below:

1. Splitting large, monolithic functions into small, modular functions: In our case, the `TypeCode` interpreter `traverse` method is the prime target for this optimization. As described earlier in Section 3.2.4, the logic for `encoder`, `decoder`, `struct_traverse`, `deep_free`, and `deep_copy` is merged into the interpreter, which increases its code size. The primary purpose of merging these methods is to reduce excessive function call overhead.

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

It is possible to split the `decode_sequence` method further to support highly specialized methods (*e.g.,* `decode_sequence_long` to decode a `sequence` of `longs`). We are currently working on providing optimizations at this level of granularity.

This optimization principle is similar to Principle 3 from Table 1, which replaces general-purpose methods with efficient special-purpose ones. In the present case, however, the large, monolithic interpreter is replaced by special-purpose methods for encoding and decoding.

2. Using "outlining" to optimize for the frequently executed case: Outlining [19] is used to remove gaps that are introduced in the processor cache as a result of branch instructions arising from error handling code. Processor cache gaps are undesirable because they waste memory bandwidth and introduce useless no-op instructions in the cache.

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

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

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

The optimizations described in this section were applied in two steps. Since the `Quantify` analysis in the previous steps revealed the receiver as the source of overhead, we

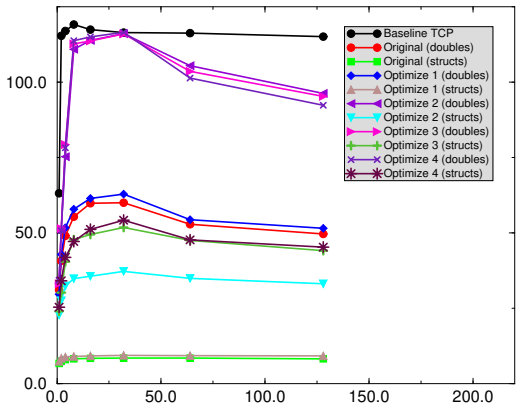


Figure 19: Throughput Comparison for Doubles and Structs After Applying Cache Optimization

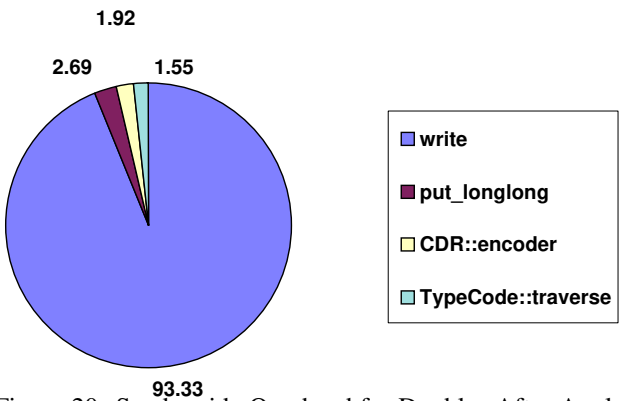


Figure 20: Sender-side Overhead for Doubles After Applying Optimizations for Cache

optimized the receiver side to gain greater processor cache effectiveness. However, the resulting `Quantify` analysis for `BinStructs` revealed that the sender-side which was `write` bound after the optimizations in step 2, spends a substantial amount of time (88%) in the interpreter. Hence we applied the similar optimizations for the cache for the sender side. Specifically, the sender-side processor cache optimizations involve splitting the interpreter into smaller, specialized functions that can encode different OMG IDL data types.

Figure 19 illustrates the benefits of the optimizations for the cache by comparing the throughput obtained for `doubles` and `BinStructs`, respectively, with those from the previous optimization steps.

Figures 22 and 23 illustrate the remaining high cost receiver-side methods. The `Quantify` analysis indicates 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. This represents a substantial improvement and illustrates that the marshaling overhead of IIOP need not be a limiting factor in ORB performance. In addition, for `BinStructs`, the sender-side which spent most of

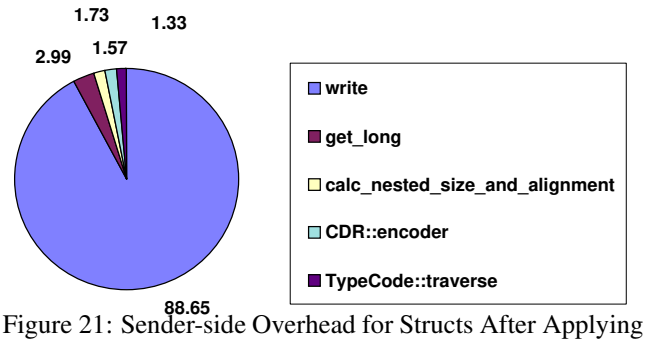


Figure 21: Sender-side Overhead for Structs After Applying Optimizations for Cache

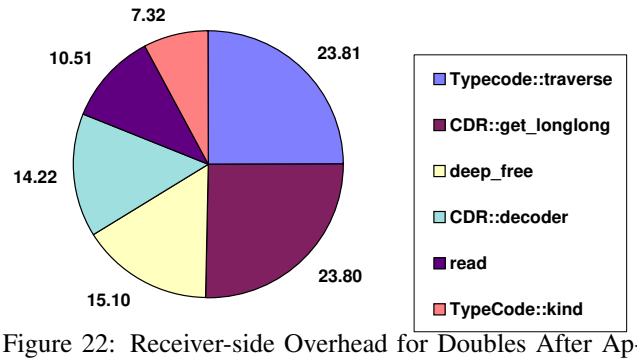


Figure 22: Receiver-side Overhead for Doubles After Applying Optimizations for Cache

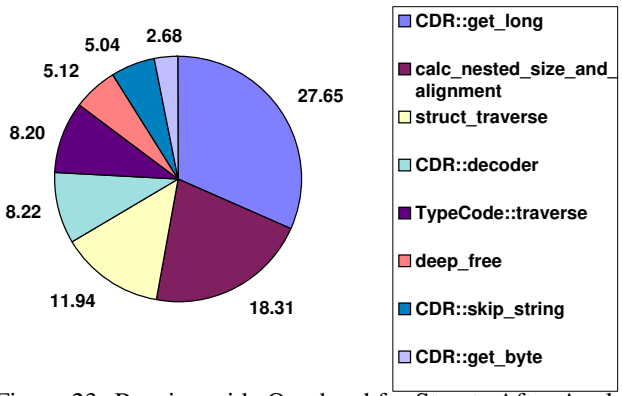


Figure 23: Receiver-side Overhead for Structs After Applying Optimizations for Cache

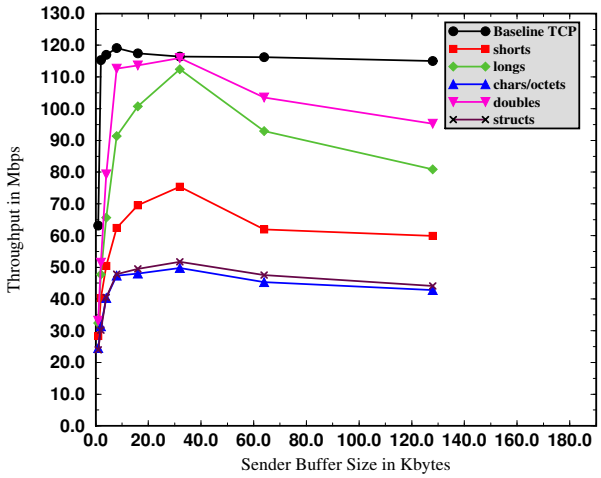


Figure 24: Throughput After Applying the Third Optimization (receiver-side processor cache optimization)

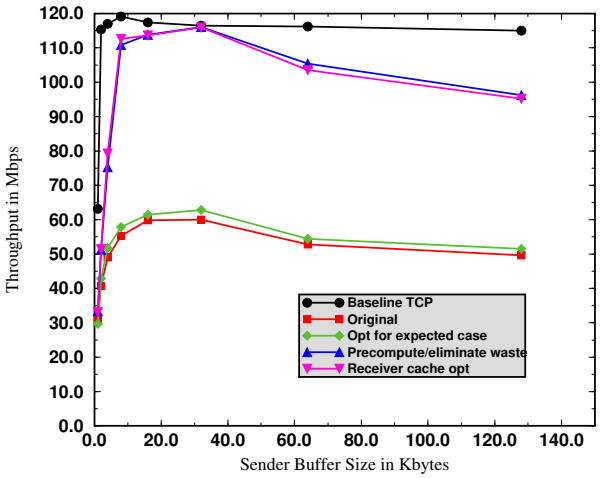


Figure 25: Throughput Comparison for Doubles After Applying the Third Optimization (receiver-side processor cache optimization)

its time in the interpreter after step 3, becomes `write` bound once again. The receiver-side analysis is comparable to that after step 3.

Optimization Step 3: Receiver-side optimizations: Figures 16 and 3 reveal that the sender is largely `write` bound. In contrast, the receiver spends most of its time in the interpreter. Therefore, it is appropriate to optimize the receiver-side code first to improve processor cache performance.

The throughput measurements recorded after incorporating these optimizations are shown in Figure 24. Figures 25 and 26 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.

Figures 27 and 28 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. These results represent a substantial improvement over the original results and illustrate that IIOP's marshalling overhead need not unduly limit ORB performance. For `BinStructs`, however, the sender-side (which was `write` bound after the optimizations in step 2) spends a substantial amount of time (88%) in the interpreter. The receiver spends most of its time in the specialized functions such as `decode_sequence`, `decode_array`. The receiver-side analysis also reveals that the function call overhead has decreased significantly compared to step 2.

Optimization Step 4: Sender-side optimizations: The sender-side processor cache optimizations involve splitting the interpreter into smaller, specialized functions that can encode different OMG IDL data types.

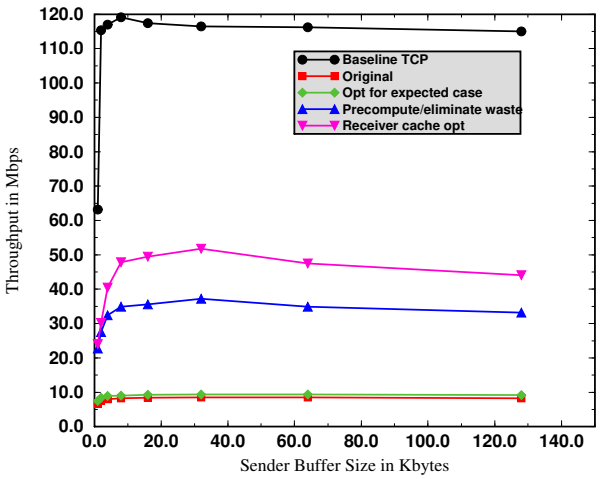
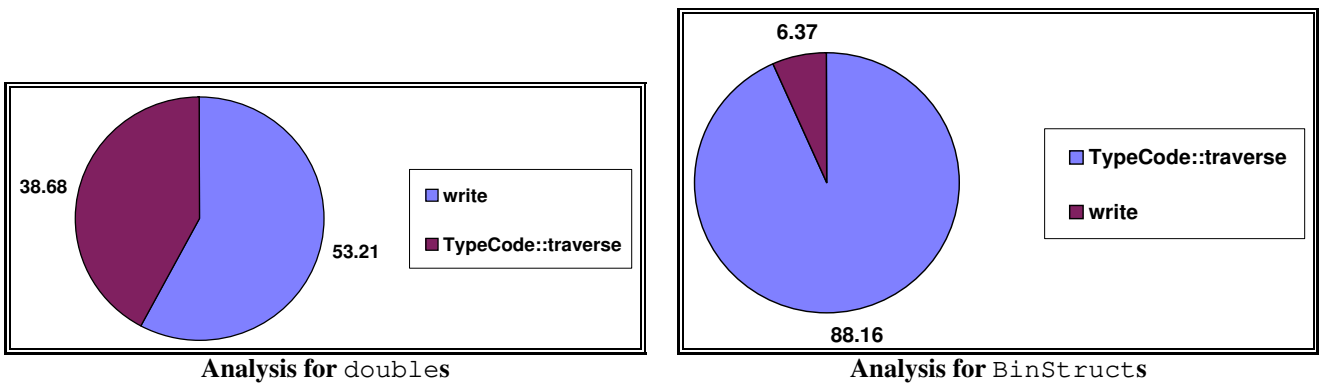
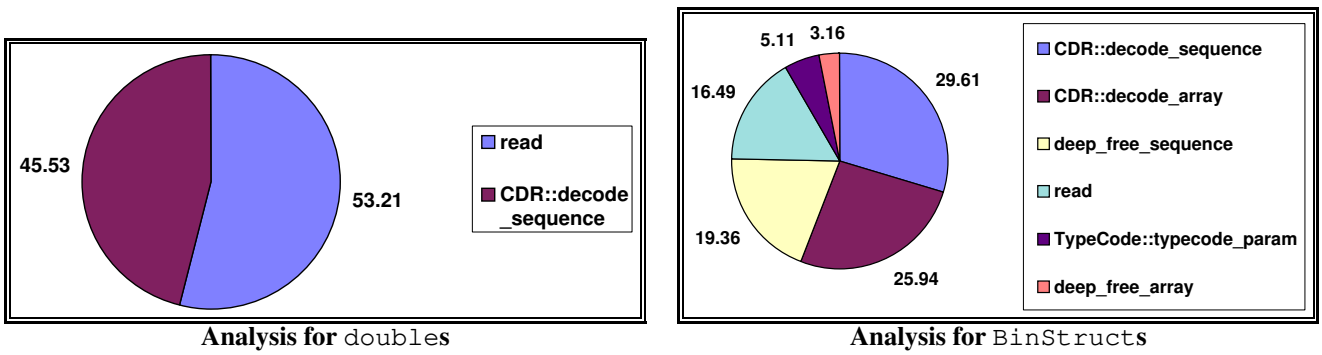


Figure 26: Throughput Comparison for Structs After Applying the Third Optimization (receiver-side processor cache optimization)



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

Figure 27: Sender-side Overhead After Applying the Third Optimization (receiver-side processor cache optimization)



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

Figure 28: Receiver-side Overhead After Applying the Third Optimization (receiver-side processor cache optimizations)

The throughput measurements recorded after incorporating these optimizations are shown in Figure 29. Figures 30 and 31 illustrate the benefits of the optimizations from step 4 by comparing the throughput obtained for `doubles` and `BinStructs`, respectively, with those from the previous optimization steps.

Figures 32 and 33 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. This represents a substantial improvement and illustrates that the marshaling overhead of IIOP need not be a limiting factor in ORB performance. In addition, for `BinStructs`, the sender-side which spent most of its time in the interpreter after step 3, becomes `write` bound once again. The receiver-side analysis is comparable to that after step 3.

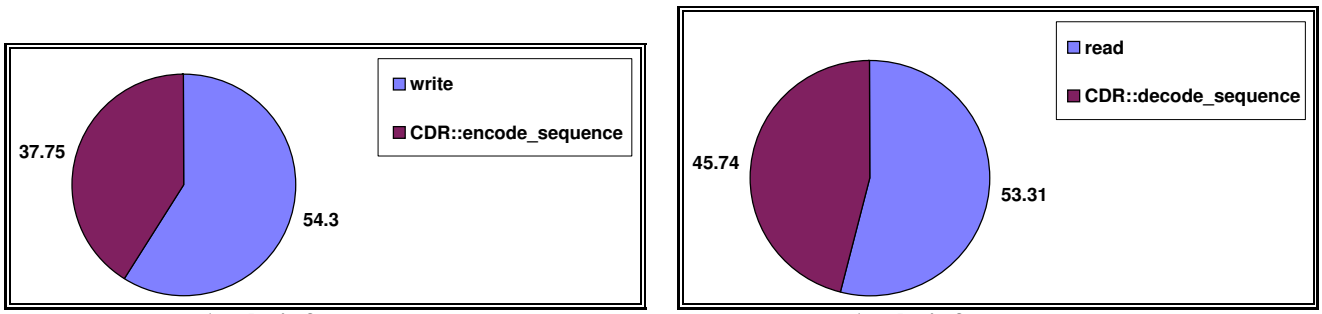
4 Related Work

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

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 common case* and Principle 3, which is *precompute, if possible*. We present the results of applying these principles to optimize IIOP in Sections 3.2.3, 3.2.4, and 3.2.5.

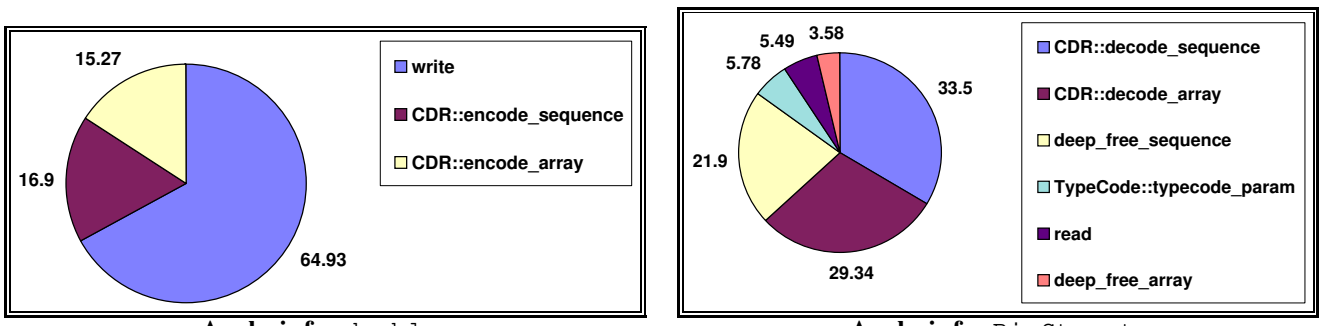
[5, 1, 3] describe the application of an optimization mechanism called *Integrated Layer Processing* (ILP). ILP is based on the observation that data manipulation loops that operate on the same protocol data are wasteful and expensive. The ILP mechanism integrates these loops into a smaller number of loops that perform all the protocol processing. The ILP optimization scheme is based on Principle 2, which *gets rid of gratuitous waste*. We demonstrate the application of this principle to IIOP in Section 3.2.4 where we eliminated unnecessary calls to the `deep_free` method, which frees primitive data types. [3] cautions against improper use of ILP since this may increase processor cache misses.

Packet filters [18, 2, 8] are a classic example of Principle 6, which recommends *passing information between layers*.



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

Figure 32: Sender-side Overhead After Applying the Fourth Optimization (sender-side processor cache optimization)



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

Figure 33: Receiver-side Overhead After Applying the Fourth Optimization (sender-side processor cache optimizations)

A packet filter demultiplexes incoming packets to the appropriate target application(s). Rather than having demultiplexing occur at every layer, each protocol layer passes certain information to the packet filter, which allows it to identify which packets are destined for which protocol layer. We applied Principle 6 for IIOP in Section 3.2.4 where we passed the `TypeCode` information and size of the element type of a `sequence` to the `TypeCode` interpreter. Therefore, the interpreter need not calculate the same quantities repeatedly.

[6] describes a facility called fast buffers (FBUFS). FBUFS combines virtual page remapping with shared virtual memory to reduce unnecessary data copying and achieve high throughput. This optimization is based on Principle 2, which focuses on *eliminating gratuitous waste* and Principle 3, which *replaces generic schemes with efficient, special purpose ones*. We applied these principles for IIOP in Section 3.2.4 where we incorporated the `struct_traverse` logic and some of the `decoder` logic into the `TypeCode` interpreter.

[19] describes a scheme called "outlining" that when used improves processor cache effectiveness, thereby improving performance. We describe optimizatons for processor cache in Section 3.2.5.

Related work on CORBA performance measurements: [9, 10, 11] show that the performance of CORBA middleware implementations is relatively poor, compared to lower-level implementations using C/C++. The primary source of ORB-level overhead stems from marshaling and demarshaling. [9] measures the performance of the static invocation interface. [10] measures the performance of the dynamic invocation interface and the dynamic skeleton interface. [11] measures performance of CORBA implementations in terms of latency and support for very large number of objects.

However, the results of earlier CORBA benchmarking experiments 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.

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

[15] describes the tradeoffs of using compiled and inter-

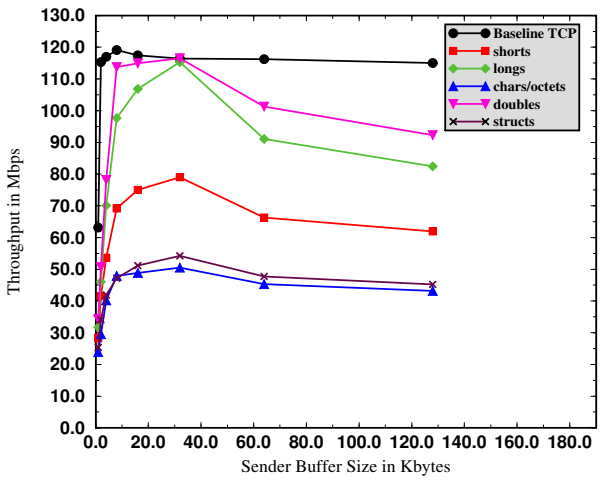


Figure 29: Throughput After Applying the Fourth Optimization (sender-side processor cache optimization)

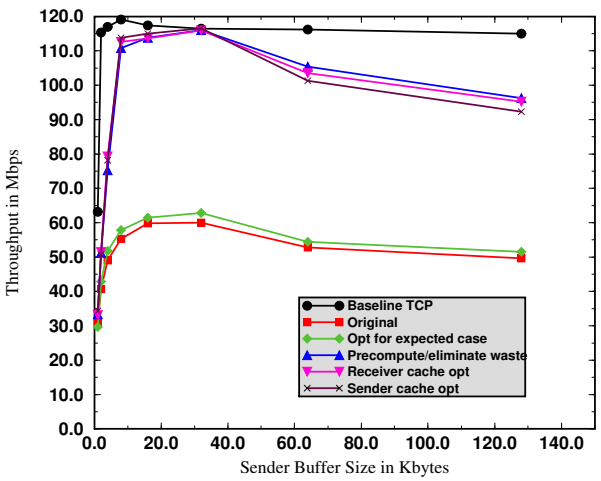


Figure 30: Throughput Comparison for Doubles After Applying the Fourth Optimization (sender-side processor cache optimization)

preted marshaling schemes. Although compiled stubs are faster, they are also larger. In contrast, interpretive marshaling is slower, but smaller in size. [15] 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 [17].

According to the SunSoft IIOP developers, interpretive marshaling is preferable since it decreases code size and increases the likelihood of remaining in the processor cache. As explained in Section 5, we are currently implementing a CORBA IDL compiler [13] that can generate compiled stubs and skeletons. Our goal is to generate efficient stubs and skeletons by extending optimizations provided in USC [23] and "Flick" [7], which is a flexible, optimizing IDL compiler. Flick uses an innovative scheme where intermediate representations guide the generation of optimized stubs. In addition, due to the intermediate stages, it is possible for Flick to map different IDLs (*e.g.,* CORBA IDL, ONC RPC IDL, MIG IDL) to a variety of target languages such as C, C++.

5 Concluding Remarks

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

Table 4 summarizes the problems encountered, the solu-

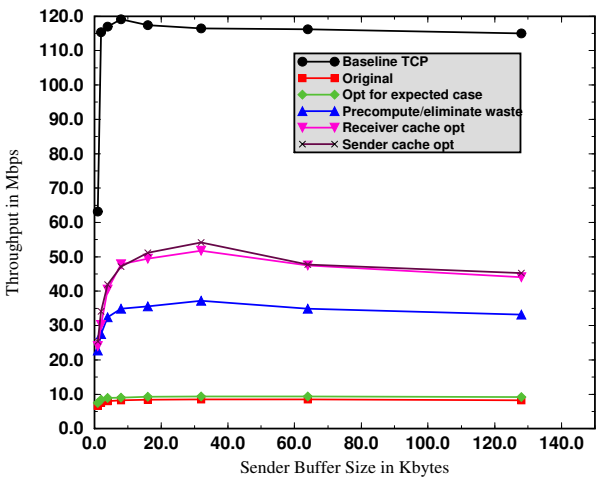


Figure 31: Throughput Comparison for Structs After Applying the Fourth Optimization (sender-side processor cache optimization)

tions proposed, and the optimization principle used to derive the solutions. The results of applying these optimizations principles to SunSoft IIOP improved its performance 1.9 times for `doubles`, 3.3 times for `longs`, 4 times for `shorts`, 5 times for `chars/octets`, and 6.7 times for richly-typed `structs` over ATM networks. Our optimized implementation is now competitive with existing commercial ORBs [9, 11] 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) [10]. The results of our optimizations provide sufficient proof that performance of complex distributed systems software can be improved by a systematic application of principle-driven optimizations.

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

Table 4: Optimization Principles Applied in TAO

We have integrated the optimized SunSoft IIOP implementation with our real-time ORB called TAO [24].[4] TAO extends the freely available SunSoft OMG IDL compiler. TAO's IDL compiler adds an optimizing code generation back-end phase to the existing front end, which generates optimized stubs and skeletons from IDL interfaces. These generated stubs and skeletons transform C++ methods into/from CORBA requests via our optimized IIOP implementation. TAO is being 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 [15].

In addition, we have incorporated an Real-time Object Adapter [14] that supports de-layered active request demultiplexing and rate monotonic scheduling and dispatching of client requests into TAO. SunSoft IIOP does not provide a standard CORBA Object Adapter. Therefore, TAO defines a set of Object Adapter strategies that dispatch requests to the correct skeleton of the target object. TAO's request demultiplexing and dispatching process provides different strategies to demultiplex requests to skeletons, including linear search, dynamic hashing or perfect hashing of operation names, or direct demultiplexing [12].

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

References

[1] M. Abbott and L. Peterson. Increasing Network Throughput by Integrating Protocol Layers. *ACM Transactions on Networking*, 1(5), October 1993.

[2] Mary L. Bailey, Burra Gopal, Prasenjit Sarkar, Michael A. Pagels, and Larry L. Peterson. Pathfinder: A pattern-based packet classifier. In *Proceedings of the* 1^{st} *Symposium on Operating System Design and Implementation*. USENIX Association, November 1994.

[3] 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.

[4] David D. Clark, Van Jacobson, John Romkey, and Howard Salwen. An Analysis of TCP Processing Overhead. *IEEE Communications Magazine*, 27(6):23–29, June 1989.

[5] David D. Clark and David L. Tennenhouse. Architectural Considerations for a New Generation of Protocols. In *Proceedings of the Symposium on Communications Architectures and Protocols (SIGCOMM)*, pages 200–208, Philadelphia, PA, September 1990. ACM.

[6] Peter Druschel and Larry L. Peterson. Fbufs: A High-Bandwidth Cross-Domain Transfer Facility. In *Proceedings of the* 14^{th} *Symposium on Operating System Principles (SOSP)*, December 1993.

[7] 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.

[8] Dawson R. Engler and M. Frans Kaashoek. DPF: Fast, Flexible Message Demultiplexing using Dynamic Code Generation. In *Proceedings of ACM SIGCOMM '96 Conference in Computer Communication Review*, pages 53–59, Stanford University, California, USA, August 1996. ACM Press.

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

[10] Aniruddha Gokhale and Douglas C. Schmidt. The Performance of the CORBA Dynamic Invocation Interface and Dynamic Skeleton Interface over High-Speed ATM Networks. In *Proceedings of GLOBECOM '96*, pages 50–56, London, England, November 1996. IEEE.

[11] 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.

[12] 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.

[13] Aniruddha Gokhale, Douglas C. Schmidt, and Stan Moyer. Tools for Automating the Migration from DCE to CORBA. In *Proceedings of ISS 97: World Telecommunications Congress*, Toronto, Canada, September 1997. IEEE Communications Society.

[14] 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.

[15] 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.

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

[17] International Organization for Standardization. *Information processing systems - Open Systems Interconnection - Specification of Basic Encoding Rules for Abstract Syntax No tation One (ASN.1)*, May 1987.

[18] Steven McCanne and Van Jacobson. The BSD Packet Filter: A New Architecture for User-level Packet Capture. In *Proceedings of the Winter USENIX Conference*, pages 259–270, San Diego, CA, January 1993.

[19] David Mosberger, Larry L. Peterson, Patrick G. Bridges, and Sean O'Malley. Analysis of Techniques to Improve Protocol Processing Latency. In *Proceedings of SIGCOMM '96*, pages 73–84, Stanford, CA, August 1996. ACM.

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

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

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

[23] Sean W. O'Malley, Todd A. Proebsting, and Allen B. Montz. USC: A Universal Stub Compiler. In *Proceedings of the Symposium on Communications Architectures and Protocols (SIGCOMM)*, London, UK, August 1994.

[24] Douglas C. Schmidt, Aniruddha Gokhale, Tim Harrison, and Guru Parulkar. A High-Performance Endsystem Architecture for Real-time CORBA. *IEEE Communications Magazine*, 14(2), February 1997.

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

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

A Overview of the CORBA GIOP and IIOP Protocols

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

A.1 Common Data Representation (CDR)

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

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

Aligned Types: Primitive OMG IDL data types are aligned on their "natural" boundaries within GIOP messages, as shown in the following table:

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

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

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:** which are shown in Table 5.

- **IIOP protocol profiles inside interoperable object references (IORs):** A protocol profile provides information about the transport protocol that enables client applications to talk to the servers. In the IIOP profile (Figure 34), this informations consists of the host name and port number on which the server is listening, and the object_key of the target object implemented by that server.

An IOR (see Figure 35) 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.

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

Table 5: TypeCode Enum Values, Parameter List Types, and Parameters

• **Service-specific contexts:** The CORBA Object Services (COS) specification [20] defines a wide range of services, such as transactions, events, naming, concurrency, and audio/video streaming. For interoperability, it may be required to pass service-specific information via opaque parameters. This is achieved using the `service-specific context`.

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 standard message types. Each message is assigned a unique value. The originator of a `giop` message can be a client and/or a server. The following table depicts the seven types of messages and the permissible originators of these messages:

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

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

Figure 34: Definition of IIOP Profile

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

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

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

Figure 35: Definition of an Interoperable Object Reference

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

A GIOP message begins with a GIOP header (Figure 36), followed by one of the message types (Figure 37), 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).

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

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

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

Figure 36: GIOP header

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 34. Figure 35 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 34.

B TTCP IDL Description and TypeCode Layout

The following CORBA IDL interface was used in our experiments to measure the throughput of SunSoft IIOP described in Section 3.2. An example of a `TypeCode` description for `BinStruct` is presented in Section 2.3.2.

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

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

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

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

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

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

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

    enum LocateStatusType {
        UNKNOWN_OBJECT,
        OBJECT_HERE,
        OBJECT_FORWARD
    };

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

Figure 37: GIOP Messages

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

Figure 38 shows the representation of the `TypeCode` layout that defines the `sequence` of `BinStructs` from Section 2.3.2. An IDL compiler is responsible for generating `TypeCode` information for all the data types described in an IDL definition. The `TypeCode` information generated by an IDL compiler is available at both the sender and the receiver end, which obviates the need to transmit typecode information along with the data over the network. Since SunSoft IIOP does not provide an IDL compiler the `TypeCode` information for all the `BinStruct` types was hand-crafted.

The layout of the `sequence` of `BinStructs` and its parameters are shown in Table 5 and described below:

TypeCode value: A CORBA `TypeCode` data structure contains a `_kind` field that indicates the `TCKind` value, which is an enumerated type. For instance, the

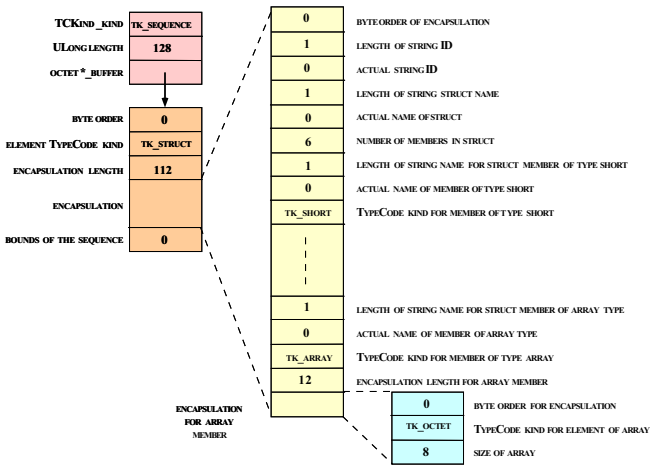


Figure 38: TypeCode for Sequence of BinStruct

`TCKind` value is `tk_sequence` in the "`sequence` of `BinStruct`" example.

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

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

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

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

C 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. We use the same `BinStruct` in this example and in our optimization experiments described in Section 3.2.

Client-side Data Path: The client-side data path is shown in Figure 39. This figure depicts the path traced by outgoing client requests through the `TypeCode` interpreter. The `CDR::encoder` method marshals the parameters from native host format into a CDR representation suitable for transmission on the network.

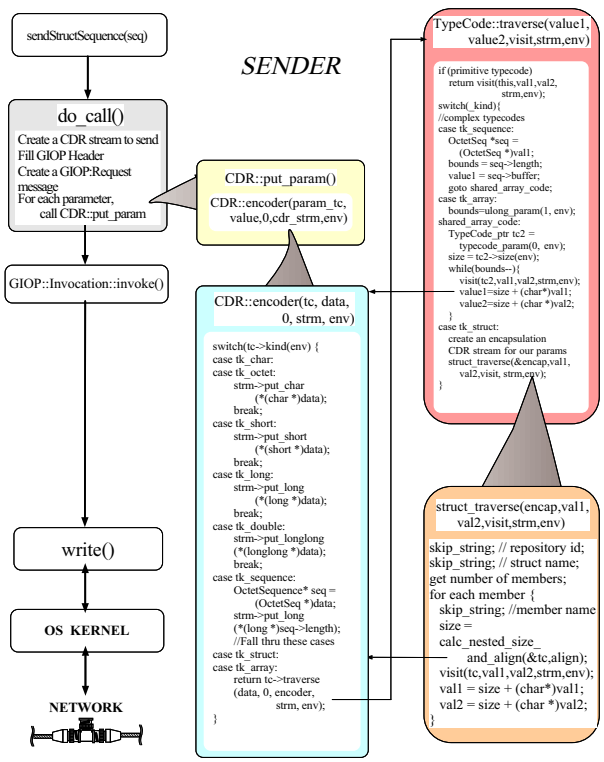


Figure 39: Sender-side Datapath for the Original SunSoft IIOP Implementation

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

Although the `do_call` and `do_dynamic_call` methods play similar roles, their type signatures are different. The `do_call` is used by the IDL compiler generated stubs to send client requests. The `do_dynamic_call` is used by the ORB's API (*e.g.* `send_oneway` and `invoke`) for DII to send client requests. The `do_dynamic_call` is

passed an NVList representing all the parameters to the operation being invoked. In addition, it is passed a flag indicating whether the operation is oneway or twoway, a string argument that represents the operation name, and a NamedValue pseudo-object that holds the results.

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

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

Server-side Data Path: The server-side data path is shown in Figure 40. This figure depicts the path traced by incoming

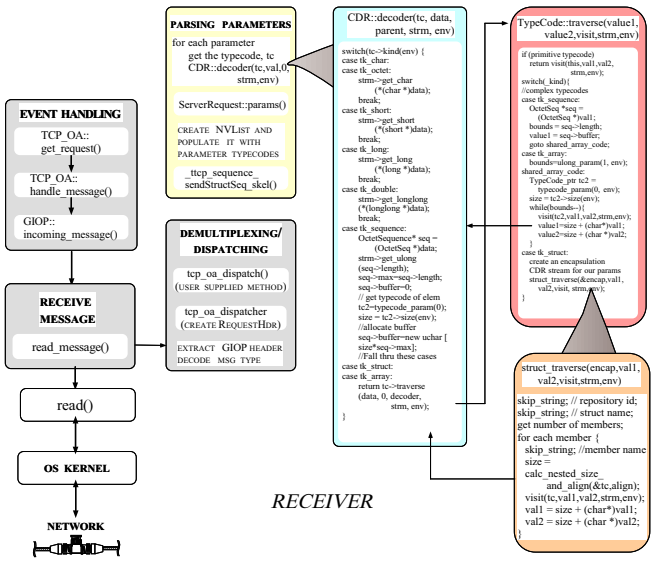


Figure 40: Receiver-side Datapath for the Original SunSoft IIOP Implementation

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

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