Optimizing a CORBA Inter-ORB Protocol (IIOP) Engine for Minimal Footprint Embedded Multimedia Systems

Aniruddha Gokhale
gokhale@research.bell-labs.com
Bell Laboratories
Lucent Technologies
600 Mountain Avenue
Murray Hill, NJ 07974

Douglas C. Schmidt
schmidt@cs.wustl.edu
Dept. of Computer Science
Washington University
One Brookings Drive
St. Louis, MO 63130

This paper was published in the IEEE Journal on Selected Areas in Communications special issue on Service Enabling Platforms for Networked Multimedia Systems, September 1999.

Abstract

To support the quality of service (QoS) requirements of embedded multimedia applications, such as real-time audio and video, electronic mail and fax, and Internet telephony, off-the-shelf middleware like CORBA must be flexible, efficient, and predictable. Moreover, stringent memory constraints imposed by embedded system hardware necessitates a minimal footprint for middleware that supports multimedia applications.

This paper provides three contributions towards developing efficient ORB middleware to support embedded multimedia applications. First, we describe the optimization principle patterns used to develop a time- and space-efficient CORBA Inter-ORB Protocol (IIOP) interpreter for TAO, which is our high-performance, real-time ORB. Second, we describe the optimizations applied to TAO's IDL compiler to generate efficient and small stubs/skeletons used in TAO's IIOP protocol engine. Third, we empirically compare the performance and memory footprint of interpretive (de)marshaling versus compiled (de)marshaling for a wide range of IDL data types.

Applying our optimization principle patterns to TAO's IIOP protocol engine improved its interpretive (de)marshaling performance to the point where it is now comparable to the performance of compiled (de)marshaling. Moreover, our IDL compiler optimizations generate interpreted stubs/skeletons whose footprint is substantially smaller than compiled stubs/skeletons. Our results illustrate that careful application of optimization principle patterns can yield both time- and space-efficient standards-based middleware.

Keywords: CORBA performance optimizations, minimal footprint ORBs, embedded multimedia applications.

I. Introduction

A. Emerging trends in embedded multimedia application development

Three trends are shaping the future development environments for embedded multimedia applications, such as MIME-enabled email, Web browsing, and Internet telephony. First, there is a movement away from *programming* applications from scratch using low-level protocols and operating system APIs to *integrating* applications using reusable components [1]. Second, there is great demand for middleware that provides remote method invocation to simplify distributed application component collaboration [2]. Third, there are increasing efforts to define *standard* middleware, such as the Object Management Group (OMG)'s Common Object Request Broker Architecture (CORBA) [3], that permits embedded multimedia applications to interwork seamlessly throughout *heterogeneous* networks and endsystems.

Standard CORBA middleware is now available that allows clients to invoke operations on distributed components without concern for component location, programming language, OS platform, communication protocols and interconnects, or hardware. These features of CORBA make it potentially suited to provide communication middleware for distributed embedded systems. However, conventional CORBA middleware generally lacks support for efficient and predictable performance *and* small footprints, which limit the rate at which performance-sensitive embedded multimedia applications have been developed to leverage advances in standard middleware.

B. Research challenges for communication middleware

Developing efficient and predictable communication middleware like CORBA for embedded multimedia applications yields many research challenges. For hand-held embedded devices, these challenges center on meeting mobile computing demands [4], [5], such as handling low bandwidth, heterogeneity in the network connections, frequent changes and disruptions in the established connections due to migration, and maintaining cache consistency.

In addition to the mobility challenges, there are restrictions on the physical size and power consumption of embedded multimedia system hardware. These restrictions constrain the amount of storage used by these systems. Likewise, storage constraints dictate the size, flexibility, and performance requirements of the middleware software that supports multimedia applications on these embedded systems.

The memory footprint of CORBA middleware is determined largely by the static and dynamic size of the ORB Core, Object Adapter, and stubs/skeletons generated by an OMG Interface Definition Language (IDL) compiler [6]. The OMG's *Minimum CORBA* [7] specification defines a standard subset of CORBA that minimizes the size of the ORB Core and Object Adapter for

Work done by the first author while at Washington University.

This work was supported in part by Boeing, DARPA contract 9701516, Motorola, NSF grant NCR-9628218, Nortel, Siemens, and Sprint.

embedded systems. However, the OMG does not define a standard specification for minimizing the footprint of IDL compiler-generated stubs/skeletons, which is considered a "quality of implementation" issue for ORB developers.

C. *Addressing research challenges with optimization principle patterns:*

Our previous research has examined many dimensions of high-performance and real-time ORB endsystem design, including static [8] and dynamic [9] scheduling, event processing [10], I/O subsystem integration [11], ORB Core connection and concurrency architectures [12], and Object Adapter demultiplexing optimizations [6]. This paper focuses on another dimension in the high-performance and real-time ORB endsystem design space: the *optimization principle patterns* used to develop a time- and space-efficient CORBA Inter-ORB Protocol (IIOP) engine for TAO [8], which is an open-source[1], standard-compliant implementation of CORBA optimized for high-performance and real-time applications.

The optimizations used in TAO's IIOP protocol engine are guided by a set of *principle patterns* [13] that have been applied to middleware [6] and lower-level networking protocols [14], such as TCP/IP. Optimization principle patterns document rules for avoiding common design and implementation mistakes that degrade the performance, scalability, and predictability of complex systems. The optimization principle patterns we applied to TAO's IIOP protocol engine 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; optimizing for the processor cache; and factoring common tasks to reduce footprint*.

The performance of the optimized version of TAO is as fast, or faster, than existing ORBs [15], [16] when using the static invocation interface (SII). Moreover, depending on the data type, it is ∼2 to 4.5 times faster than ORBs when using the dynamic skeleton interface (DSI) [17]. In addition, the memory footprint of TAO's stubs/skeletons is substantially (*i.e.*, ∼40%) smaller than stubs/skeletons generated by other optimizing IDL compilers [18].

D. *Paper organization*

This paper is organized as follows: Section II outlines the CORBA reference model, the GIOP/IIOP interoperability protocols, SunSoft IIOP, and TAO; Section III presents the results of TAO's performance optimizations on the SunSoft IIOP interpreter; Section IV presents the results of optimizing TAO's IDL compiler to produce efficient and small footprint stubs and skeletons for a range of IDL data types; Section V compares our research with related work; and Section VI provides concluding remarks.

[1]The source code and documentation for TAO is freely available at `www.cs.wustl.edu/∼schmidt/TAO.html`.

II. Background

TAO is a real-time ORB based on the SunSoft IIOP protocol engine. TAO is targeted for applications with deterministic and statistical QoS requirements, as well as best effort requirements. This section outlines the CORBA reference model, its GIOP/IIOP interoperability protocols, SunSoft IIOP, and TAO.

A. *Overview of CORBA*

CORBA Object Request Brokers (ORBs) [19] allow clients to invoke operations on distributed objects without concern for object location, programming language, OS platform, communication protocols and interconnects, and hardware. Figure 1 illustrates the key components in the CORBA reference model that collaborate to provide this degree of portability, interoperability, and transparency. For a complete synopsis of CORBA's

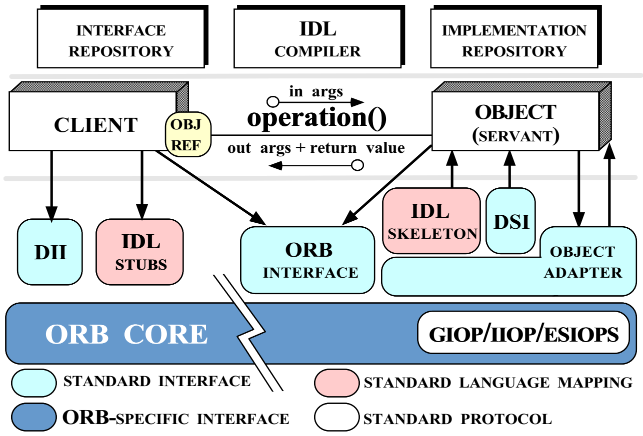


Fig. 1. Key Components in the CORBA 2.x Reference Model

components, see [3].

B. *Overview of CORBA GIOP and IIOP*

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

The GIOP specification consists of the following elements:

B.1 A Common Data Representation (CDR) definition

The GIOP specification defines the CDR transfer syntax, which maps OMG IDL types from the native host format into a low-level *bi-canonical* representation that supports both little-endian and big-endian formats. All OMG IDL data types are marshaled using the CDR syntax into an *encapsulation*, (which is an octet stream that holds marshaled data) and exchanged between clients and servers.

B.2 GIOP message formats

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

B.3 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 [3].

C. Overview of the SunSoft IIOP Protocol Engine

SunSoft IIOP is a freely available, open-source[2] implementation of IIOP version 1.0. The key features and architecture of SunSoft IIOP are outlined below.

C.1 CORBA Features Supported by SunSoft IIOP

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

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

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

SunSoft IIOP supports dynamic skeletons via the dynamic skeleton interface (DSI). The DSI is used by applications and ORB bridges [3] that have no compile-time knowledge of the interfaces they implement. Thus, the DSI parses incoming requests, unmarshals their parameters, and demultiplexes requests to the appropriate servants.

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

C.2 The Sunsoft IIOP Software Architecture

The components in SunSoft IIOP are shown in Figure 2. The

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

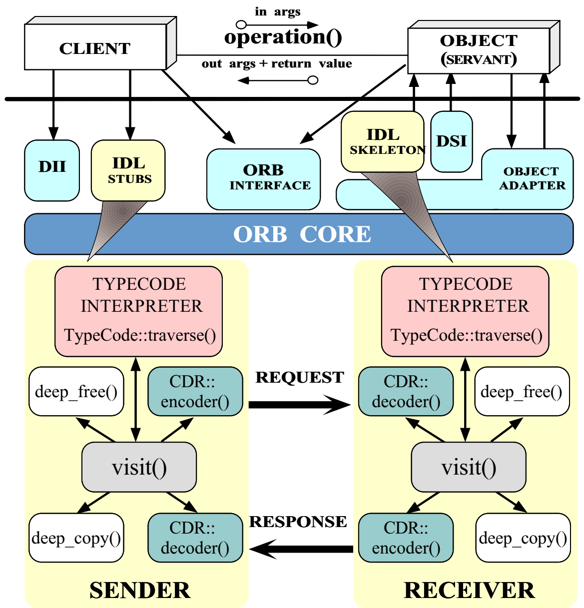


Fig. 2. Components in the SunSoft IIOP Implementation

`TypeCode` (de)marshaling protocol engine is the primary component of SunSoft IIOP. SunSoft IIOP's protocol engine is an interpreter that encodes or decodes parameter data by identifying their `TypeCodes` at run-time using the `_kind` field of each `TypeCode` object.

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

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

C.2.a The `TypeCode::traverse` method. The SunSoft IIOP interpreter is implemented within the `traverse` method of the `TypeCode` class. All parameter marshaling and demarshaling is performed interpretively by traversing the data structure according to the layout of the `TypeCode/Request` tuple passed to `traverse`. This method is passed a pointer to a `visit` method (described below), which interprets CORBA requests based on their `TypeCode` layout. The request part of the tuple contains the data that was passed by an application on the client-side or received from the OS protocol stack on the server-side.

C.2.b The `visit` method. The `TypeCode` interpreter invokes the `visit` method to marshal or demarshal the data associated with the `TypeCode` it is currently interpreting. The `visit` method is a pointer that contains the address of one of the four methods described below:

- *The* `CDR::encoder` *method:* The `encoder` method of the `CDR` class converts application data types from their native host representation into the CDR representation used to transmit CORBA requests over a network.
- *The* `CDR::decoder` *method:* The `decoder` method of

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

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

• *The* deep_free *method:* The deep_free method is used by the SunSoft DSI server to release dynamically allocated memory after incoming data has been demarshaled and passed to a server application.

C.2.c The utility methods. The following SunSoft IIOP methods perform various ORB utility tasks:

• *The* calc_nested_size_and_alignment *method:* This method calculates the size and alignment of composite IDL data types like structs or unions.

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

Section II-C.3 examines the run-time behavior of SunSoft IIOP by tracing the path taken by requests used to 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

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

The performance of SunSoft IIOP for these data types is examined in Section III.

C.3 Tracing the Data Path of a SunSoft IIOP Request

To illustrate the run-time behavior of SunSoft IIOP, we trace the path taken by requests that transmit a sequence of BinStructs. We show how the TypeCode interpreter consults the TypeCode information as it (de)marshals parameters. We use the same BinStruct in this example and in our optimization experiments described in Section III-B.1.

C.3.a Client-side Data Path. The client-side data path is shown in Figure 3. This figure depicts the path traced by outgoing client requests through the TypeCode interpreter. The CDR::encoder method marshals the parameters from native host format into a CDR representation suitable for transmission on the network.

The client uses the do_call method, which is the static invocation interface (SII) API provided by SunSoft IIOP. This method uses the TypeCode interpreter to marshal the parameters and send the client requests. The dynamic invocation interface (DII) mechanism uses the do_dynamic_call method to send client requests.

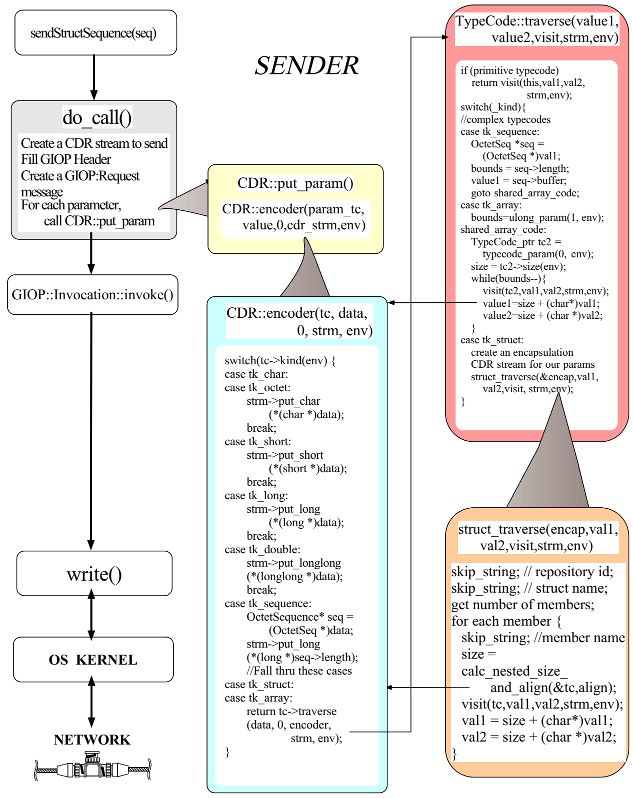


Fig. 3. Sender-side Datapath for the Original SunSoft IIOP Implementation

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

The do_call method creates a CDR stream into which CORBA operations and their 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 contained in a composite data type, such as a struct or union. For each member of a composite 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.

C.3.b Server-side Data Path. The server-side data path is shown in Figure 4. This figure depicts the path traced by in-

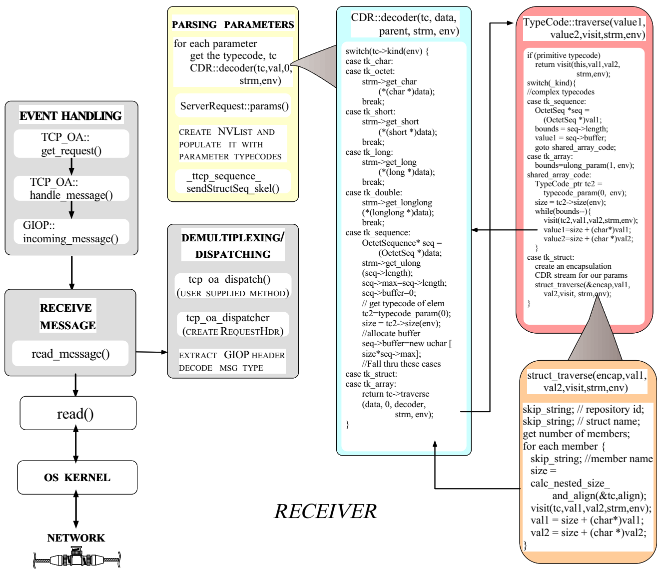


Fig. 4. Receiver-side Datapath for the Original SunSoft IIOP Implementation

coming client requests through the SunSoft IIOP `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 operation 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 by invoking an upcall on a user-supplied servant 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 servant method.

D. Overview of TAO

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

D.1 Lack of complete ORB features

Although SunSoft IIOP provides an ORB Core, an IIOP protocol engine, and a DII and DSI implementation, it lacks an IDL compiler, an Implementation Repository, and a Portable Object Adapter (POA). TAO implements these missing features and provides several new features, such as real-time scheduling and dispatching mechanisms [8].

D.2 Lack of real-time features

SunSoft IIOP provides no support for real-time features. For instance, it uses a FIFO strategy for scheduling and dispatching client IIOP requests. FIFO strategies can yield unbounded priority inversions when lower priority requests block the execution of higher priority requests [12]. TAO is designed carefully to prevent unbounded priority inversions. For instance, it provides a flexible scheduling service [8], [9] that utilizes QoS information associated with the I/O subsystem [11] to schedule and dispatch requests according to their end-to-end priorities. To enable this, TAO extends SunSoft IIOP to allow separate IIOP connections to run within real-time threads with suitable priorities [20].

D.3 Lack of IIOP optimizations

As described in Section III, SunSoft IIOP incurs relatively high performance overhead due to excessive marshaling/demarshaling overhead, data copying, and high-levels of function call overhead. Therefore, we applied the following optimization principle patterns [14], [6] that improved its performance considerably: *(1) optimizing for the common case, (2) eliminating gratuitous waste, (3) replacing general-purpose methods with efficient special-purpose ones, (4) precomputing values, if possible, (5) storing redundant state to speed up expensive operations, (6) passing information between layers, and (7) optimizing for processor cache affinity.* As shown in Section III, our optimizations yielded speedups of 2 to 6.7 for various types of OMG IDL data.

TAO alleviates the limitations with SunSoft IIOP described above to create a complete real-time ORB endsystem. TAO is a high-performance, real-time ORB endsystem targeted for applications with deterministic and statistical QoS requirements, as well as "best-effort" requirements. The TAO ORB endsystem contains the network interface, OS, communication protocol, and CORBA-compliant middleware components and features shown in Figure 5. TAO supports the standard OMG CORBA

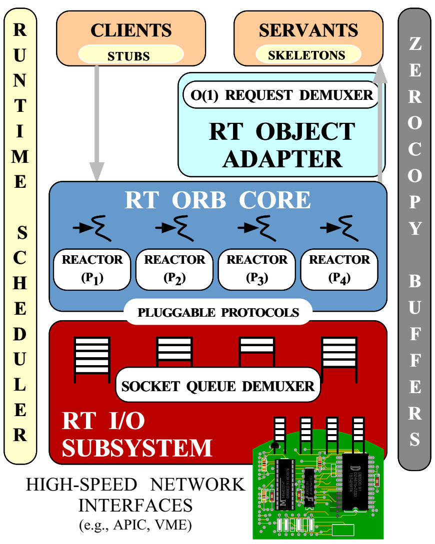


Fig. 5. Components in the TAO Real-time ORB Endsystem

reference model [3], with many enhancements [20], [6], [11] designed to overcome the shortcomings of conventional ORBs for high-performance and real-time applications.

TAO is developed atop lower-level middleware called ACE [21], which implements core concurrency and distribution patterns [22] for communication software. ACE provides reusable C++ wrapper facades and framework components that support the QoS requirements of high-performance, real-time applications. ACE runs on a wide range of OS platforms, including Win32, most versions of UNIX, and real-time operating systems like Sun/Chorus ClassiX, LynxOS, and VxWorks.

III. Optimizing TAO's IIOP Interpreter

As explained in Section II-C, SunSoft IIOP is a protocol engine that implements IIOP version 1.0 using a `TypeCode` interpreter to (de)marshal operation parameters. The interpretive design and lack of optimizations in SunSoft IIOP degrades its performance substantially and renders it unsuitable to support performance-sensitive embedded multimedia applications. This section describes how we used a measurement-driven methodology, guided by optimization principle patterns, to improve the performance of SunSoft IIOP for the TAO real-time ORB.

A. CORBA/ATM Testbed Environment

A.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 Mbps/port switch. Each UltraSparc-2 contains two 168 MHz Super SPARC CPUs with a 1 Megabyte cache per-CPU. The SunOS 5.5.1 TCP/IP protocol stack is 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 6.

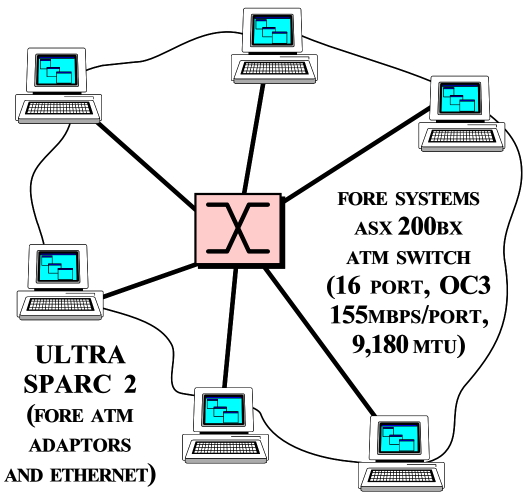


Fig. 6. Hardware for the CORBA/ATM Testbed

A.2 Traffic Generator for Throughput Measurements

Traffic for the experiments was generated and consumed by an extended version of the widely available `ttcp` [23] protocol benchmarking tool. We extended `ttcp` for use with SunSoft IIOP. We hand-crafted the stubs and skeletons for the different operations defined in the interface. Our hand-crafted client-side stubs use SunSoft IIOP's SII API, *i.e.*, the `do_call` method. The `do_call` method provides an interface to pass client operation arguments to the ORB's interpretive (de)marshaling engine. 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 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 sent repeatedly until a total of 64 Mbytes of data was transmitted.

A.3 Profiling Tools

The profile information for the empirical analysis was obtained using the `Quantify` [24] 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 system calls made by an application. `truss` was used to observe the return values of system calls, such as `read` and `write`, which indicates the number of times that buffers were written to and read from the network.

B. Performance Results and Benefits of Optimization Principle Patterns

B.1 Methodology

CORBA implementations like SunSoft IIOP are representative of complex communication software. Optimizing such software is hard since seemingly minor "mistakes," such as excessive data copying, dynamic allocation, or locking, can reduce performance significantly [15], [12]. Therefore, developing high-performance, predictable, and space-efficient ORBs requires an iterative, multi-step optimization process. First, we measured the performance of the ORB with blackbox and whitebox benchmarks to pinpoint key sources of overhead. Next, we analyzed these sources of overhead carefully and systematically applied optimization principle patterns to remove the bottlenecks. We repeated this optimization process until no major performance bottlenecks remained.

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

The optimizations described in this section are based on the core principle patterns shown in Table I for implementing protocols efficiently. [14], [6] describe a family of optimization principle patterns and illustrates how they have been applied in existing protocol implementations, such as TCP/IP. This section focuses on the optimization principle patterns we applied systematically to improve the performance of SunSoft IIOP. We focused on these principle patterns since our experiments revealed they were the most strategic to improving SunSoft IIOP's performance. When describing our optimizations, we refer to these principle patterns and empirically show how their use is justified.

#	Optimization Principle Pattern
1	Optimize for the common case
2	Eliminate gratuitous waste
3	Replace inefficient general-purpose methods with efficient special-purpose ones
4	Precompute values, if possible
5	Store redundant state to speedup expensive operations
6	Pass information between layers
7	Optimize for the processor cache

TABLE I
OPTIMIZATION PRINCIPLE PATTERNS FOR EFFICIENT PROTOCOL IMPLEMENTATIONS

The SunSoft IIOP optimizations were performed in the following three steps, corresponding to the principle patterns from Table I:

1. *Aggressive inlining to optimize for the common case* – which is discussed in Section III-B.3;
2. *Precomputing, adding redundant state, passing information through layers, eliminating gratuitous waste, and specializing generic methods* – which is discussed in Section III-B.4;

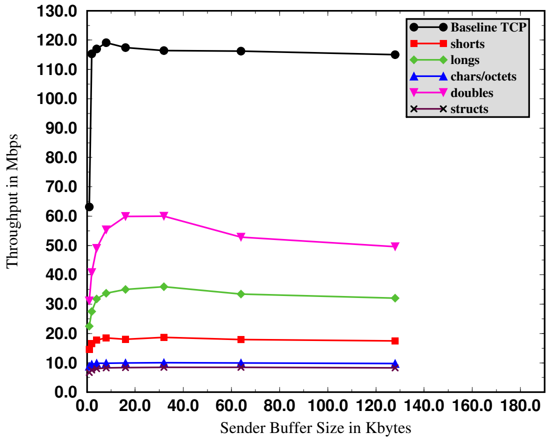


Fig. 7. Throughput for the Original SunSoft IIOP Implementation

3. *Optimizing for the processor cache* – which is discussed in Section III-B.5.

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

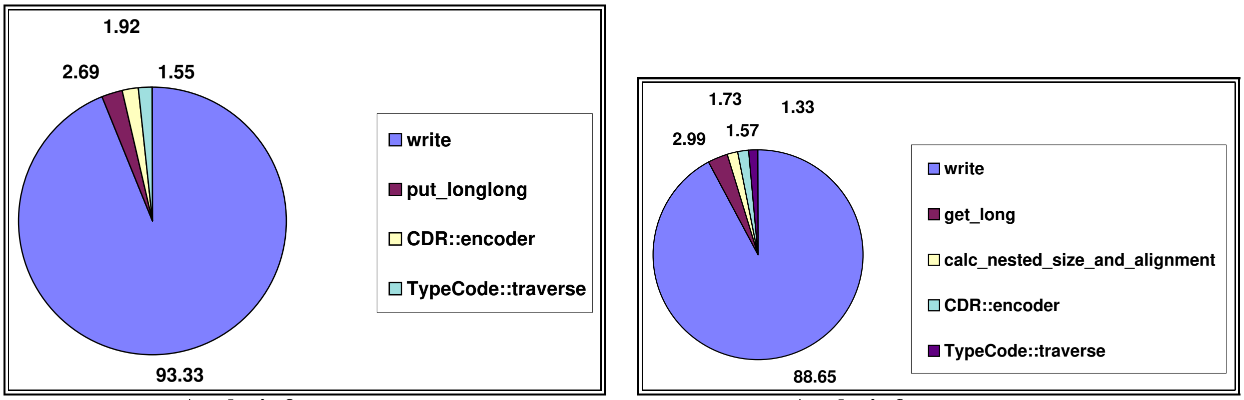
The comparisons focus on data types that exhibited the widest range of performance, *i.e.*, `double` and `BinStruct`. As shown below, the first optimization step did not improve performance significantly. However, this step was necessary since it revealed the actual sources of overhead, which were then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

B.2 Performance of the Original SunSoft IIOP Implementation

B.2.a Sender-side performance. Figure 7 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 using TCP/IP and sockets. These results indicate that different data types achieved substantially different levels of throughput.

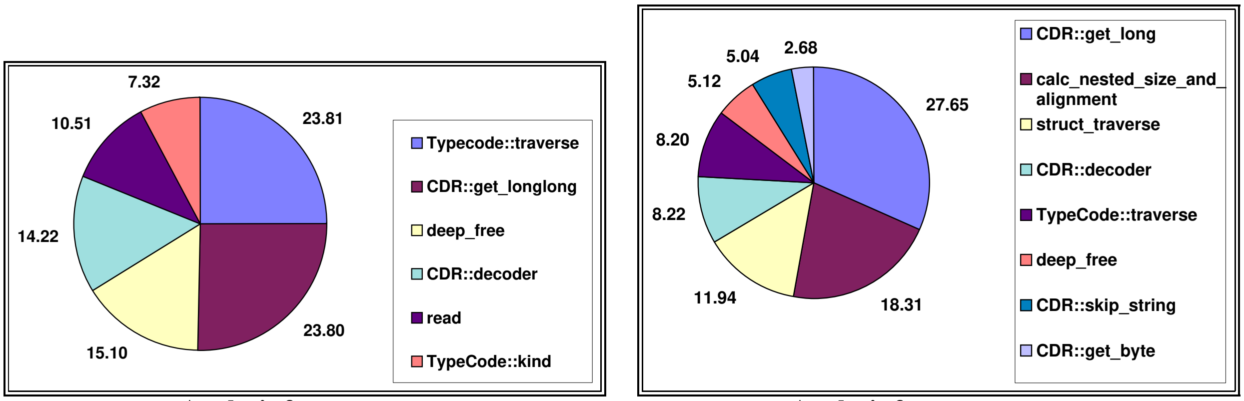
The highest ORB throughput results from sending `doubles`, whereas `BinStructs` displayed the worst behavior. This variation in behavior stems from the (de)marshaling overhead for different data types. In addition, the original implementation of the interpretive (de)marshaling engine in SunSoft IIOP incurred a large number of recursive method calls.

Figure 8 presents the results of using `Quantify` to send 64 Mbytes of `doubles` and `BinStructs` using a 128 Kbyte sender buffer. The results reveal that the sender spends ∼90% of its run-time performing `write` system calls to the network. This overhead stems from the transport protocol flow control enforced by the receiving side, which cannot keep pace with the sender due to excessive presentation layer overhead. Table II provides detailed `Quantify` measurements indicating the time



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

Fig. 8. Sender-side Overhead in the Original IIOP Implementation



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

Fig. 9. Receiver-side Overhead in the Original IIOP Implementation

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

TABLE II
SENDER-SIDE OVERHEAD IN THE ORIGINAL IIOP IMPLEMENTATION

taken by dominant operations and the number of times they were invoked.

B.2.b Receiver-side performance. The `Quantify` analysis for the receiver-side is shown in Figure 9 and Table III. The receiver-side results[3] for sending primitive data types indicate that most run-time overhead is 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.

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

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

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

TABLE III
RECEIVER-SIDE OVERHEAD IN THE ORIGINAL IIOP IMPLEMENTATION

Figures 8 and 9 illustrate that the SunSoft IIOP receiver is the primary performance bottleneck. Therefore, our initial set of optimizations are designed to improve receiver performance. Likewise, since the receiver is the bottleneck, we only show its `Quantify` profile measurements.

B.3 Optimization Step 1: Inlining to Optimize for the Common Case

B.3.a Problem: high invocation overhead for small, frequently called methods. This subsection describes an optimization to improve the performance of IIOP receivers. We applied principle pattern 1 from Table I, which *optimizes for the common case*. Figure 9 illustrates that the appropriate `get` method of the CDR class must be invoked to retrieve the data from the incoming stream into a local copy. For instance, depending on the data type, methods like `CDR::get_long` or `CDR::get_longlong` are called between 10-80 million times to decode 64 Mbytes of data, as indicated in Table III. Since these `get` methods are invoked quite frequently they are prime targets for our first optimization step.

B.3.b 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.

B.3.c Problem: lack of C++ compiler support for aggressive inlining. Our intermediate `Quantify` results after inlining, shown in Figure 10, reveal that supplying the `inline` keyword to the compiler does not always work since the compiler occasionally ignores this "hint." Likewise, inlining some methods may cause others to become "non-inlined." This occurs since the originally inlined operations (*e.g.,* `ptr_align_binary`) now invoke newly inlined operations thereby increasing their size. The C++ compiler then chooses not to inline operations that were inlined originally.

B.3.d 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 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 transferred data between UltraSPARC machines with the same byte order.

B.3.e Optimization results. The throughput measurements after aggressive inlining are shown in Figure 11. Figures 12 and 13 illustrate the effect of aggressive inlining on the throughput of `doubles` and `BinStructs`. Figures 12 and 13 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 14 and 15, and Tables IV and V 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.

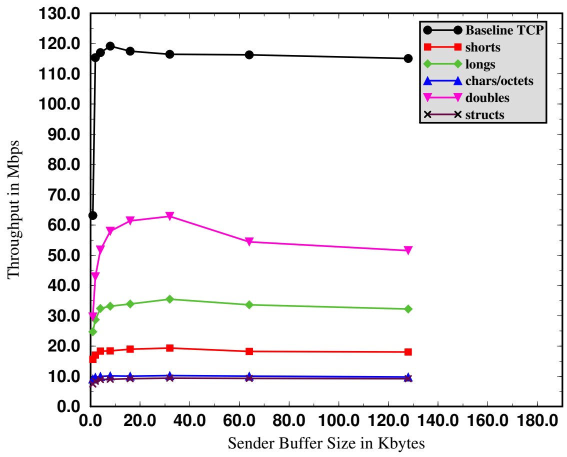
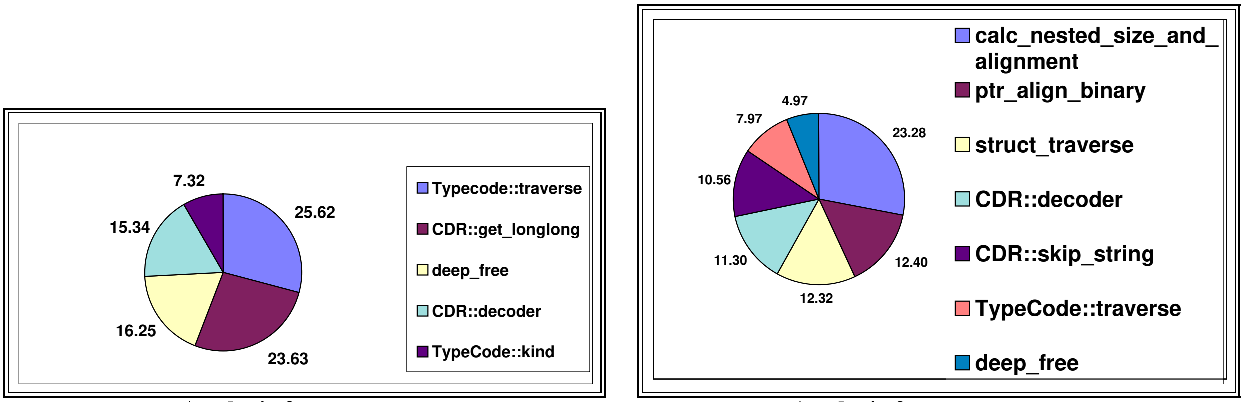


Fig. 11. Throughput After Applying the First Optimization (inlining)

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



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

Fig. 10. Receiver-side Overhead in the IIOP Implementation After Simple Inlining

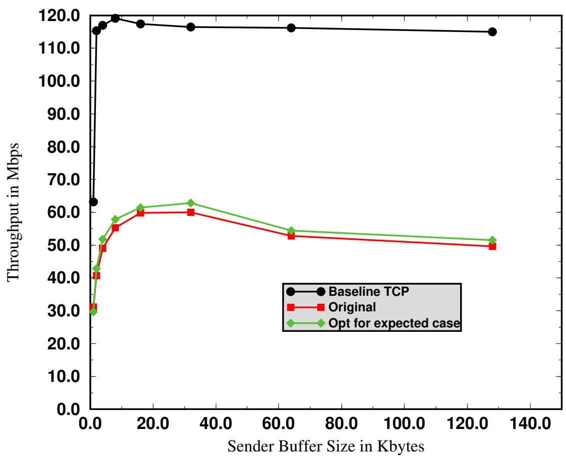


Fig. 12. Throughput Comparison for Doubles After Applying the First Optimization (inlining)

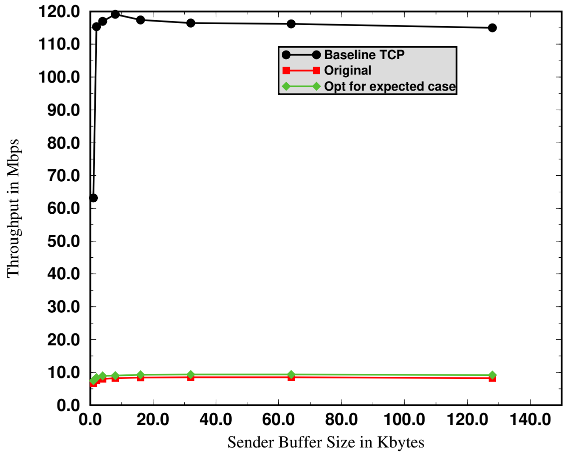


Fig. 13. Throughput Comparison for Structs After Applying the First Optimization (inlining)

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

TABLE IV

SENDER-SIDE OVERHEAD 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

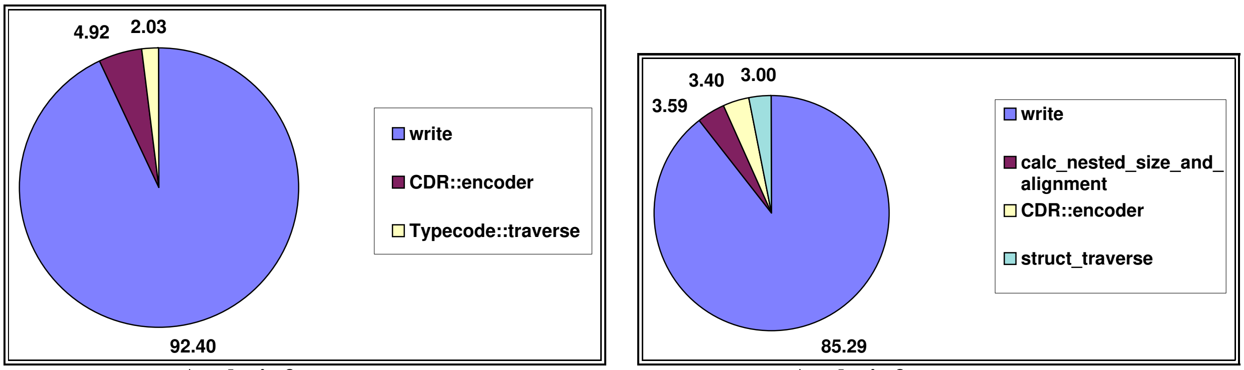
TABLE V

RECEIVER-SIDE OVERHEAD AFTER APPLYING THE FIRST OPTIMIZATION (AGGRESSIVE INLINING)

their run-time costs. As shown in Figures 3 and 4, these methods are called a large number of times, as indicated in Figure 15 and Table V.

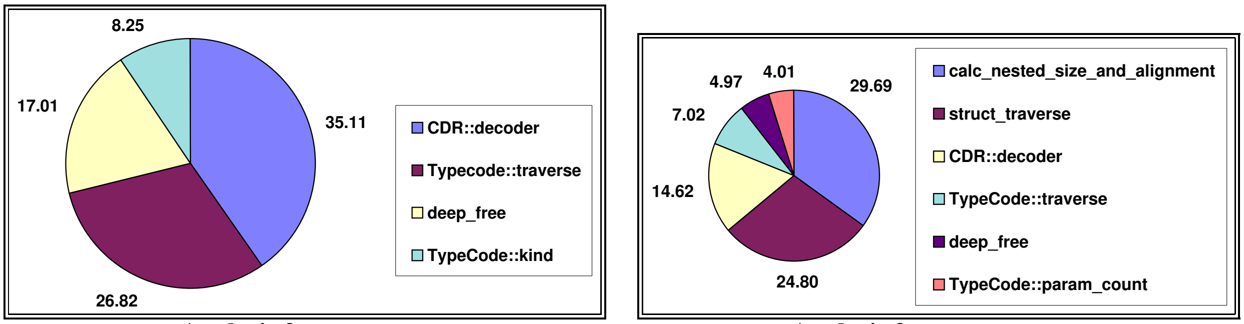
Certain SunSoft IIOP methods such as `CDR::decoder` and `TypeCode::traverse` are large and general-purpose. Inlining the small methods described above causes further "code bloat" for these methods. Thus, when they call each other recursively a large number of times, very high method call overhead results.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

In summary, although our first optimization step did not improve performance dramatically, it helped to reveal the actual sources of overhead in the code, as explained in Section III-B.4.



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

Fig. 14. Sender-side Overhead After Applying the First Optimization (aggressive inlining)



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

Fig. 15. Receiver-side Overhead After Applying the First Optimization (aggressive inlining)

B.4 Optimization Step 2: Precomputing, Adding Redundant State, Passing Information Through Layers, Eliminating Gratuitous Waste, and Specializing Generic Methods

B.4.a Problem: too many method calls. The aggressive inlining optimization in Section III-B.3 did not cause substantial improvement in performance due to the processor cache effects shown in this section and Section III-B.5.

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

To see why these methods were invoked so frequently, we analyzed the calls to `struct_traverse`. The `TypeCode` interpreter invoked `struct_traverse` 2,097,152 times for data transmissions of 64 Mbytes in `sequences` of 32-byte `Binstructs`. In addition, the SunSoft IIOP 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 detailed analysis of the SunSoft IIOP implementation (shown in Figure 4 and in the explanation in Section II-C), 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 in turn may invoke the `traverse` method.

Close scrutiny of the CORBA request datapath shown in Figure 4 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:

B.4.b Solution 1: reduce gratuitous waste by precomputing values and storing additional state. This solution is motivated by 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 pattern 2 from Table I). 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 gratuitous waste, we can *precompute* (principle pattern 4) the size and alignment requirements of each member and store them using *additional state* (principle pattern 5) to speed up expensive operations. We store this additional state as private data members of the SunSoft's `TypeCode` class. Thus, the `TypeCode` for `BinStruct` will calculate the size and alignment *once* and store these in the private data members. Every time the interpreter wants to traverse `BinStruct`, it uses the `TypeCode` for `BinStruct` that has already precomputed its size and alignment. Note that our additional state does not affect the IIOP protocol since this state is stored locally in the `TypeCode` interpreter and is not passed across the network.

In general, all `struct` elements in a `sequence` may not have the same size. For instance, a `sequence` of `Any`s or `struct`s with `string` fields may have elements with variable sizes. In such cases, this optimization will not apply. For the `BinStruct` case described in this paper, however, a highly optimizing IDL compiler, such as Flick [18], could determine that all `sequence` elements have identical sizes. It could then generate stub and skeleton code that can eliminate gratuitous waste.

B.4.c Solution 2: convert generic methods into special-purpose, efficient ones. To further reduce method call overhead, and to decrease the potential for processor cache misses, we moved the `struct_traverse` logic for handling `struct`s into the `traverse` method. In addition, we introduced the `encoder`, `decoder`, `deep_copy`, and `deep_free` logic into the `traverse` method. This optimization illustrates an application of principle pattern 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 (de)marshaling engine to remain in the cache. However, this scheme may not result in optimal cache hit performance for embedded system hardware with small caches since the `traverse` method is excessively large. Section III-B.5 describes optimizations we used to improve processor cache performance.

B.4.d Problem: expensive no-ops for memory deallocation. Figure 15 reveals that the overhead of the `deep_free` method remains significant for primitive data types. This method is similar to the `decoder` method that traverses the `TypeCode` and deallocates dynamic memory. For instance, the `deep_free` method has the same type signature as the `decoder` method. Therefore, it can use the recursive `traverse` method to navigate the data structure corresponding to the parameter and deallocate memory.

Careful analysis of the `deep_free` method indicates that memory must be freed for constructed data structures, such as IDL `sequence`s and `struct`s. In contrast, for `sequence`s 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.

B.4.e Solution: eliminate gratuitous waste. To optimize the no-op memory deallocations, we changed the deletion strategy

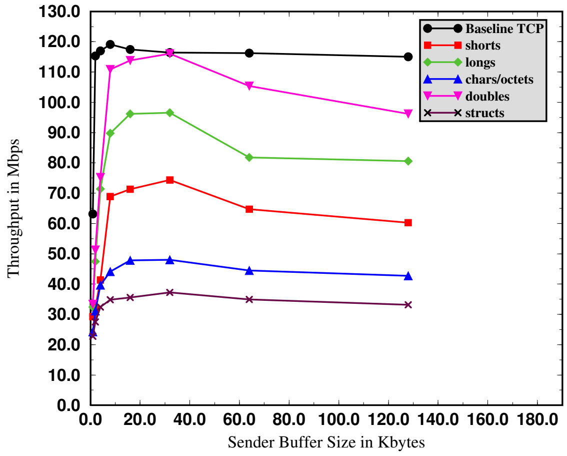


Fig. 16. Throughput After Applying the Second Optimization (precomputation and eliminating waste)

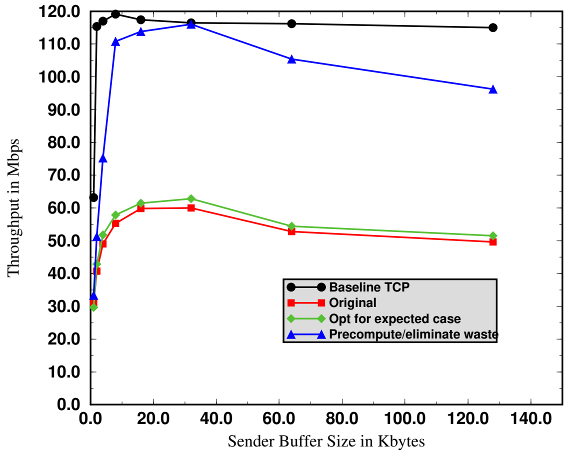


Fig. 17. Throughput Comparison for Doubles After Applying the Second Optimization (precomputation and eliminating waste)

for `sequence`s so that the element's `TypeCode` is checked *first*. If it is a primitive type, such as `double`, the traversal is not done and memory is deallocated directly.

B.4.f Optimization results. The throughput measurements recorded after incorporating these optimizations are shown in Figure 16. Figures 17 and 18 illustrate the benefits of the optimizations from step 2 by comparing the throughput obtained for `doubles` and `BinStructs`, respectively, with results from previous optimization steps.

Tables VI and VII, and Figures 20 and 20 depict the profiling measurements for the sender and receiver, respectively. The receiver methods accounting for the most execution time for `doubles` include `traverse`, `decoder`, and `deep_free`. For `BinStructs`, the run-time costs of the `traverse` method in the receiver increases significantly compared to the previous optimization steps. This is due primarily to the inclusion of the `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,

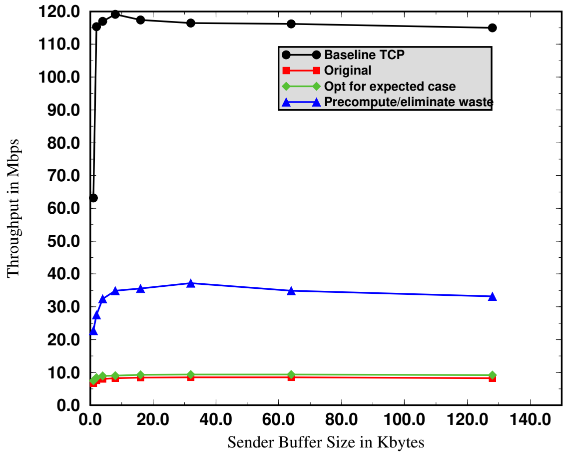


Fig. 18. Throughput Comparison for Structs After Applying the Second Optimization (precomputation and eliminating waste)

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

TABLE VI
SENDER-SIDE OVERHEAD AFTER APPLYING THE SECOND OPTIMIZATION (GETTING RID OF WASTE AND PRECOMPUTATION)

this design improved processor cache affinity, which yielded better performance. In addition, due to precomputation, the `calc_nested_size_and_alignment` method need not be called repeatedly.

Applying the optimization described above yields a substantial improvement. This result illustrates that the (de)marshaling overhead of IIOP need not be a limiting factor in ORB performance.

B.5 Optimization Steps 3 and 4: Optimizing for Processor Caches

Processor caches are small, very fast memory used to significantly speed up operations [25]. To leverage the advantages offered by the processor cache it is imperative that operation footprints be small.

[26] 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.

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 VII
RECEIVER-SIDE OVERHEAD AFTER APPLYING THE SECOND OPTIMIZATION (GETTING RID OF WASTE AND PRECOMPUTATION)

Hence, the optimizations described in this section are aimed at improving processor cache affinity, thereby improving performance.

B.5.a Problem: very large, monolithic interpreter. Section III-B.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 embedded system hardware with smaller cache sizes, it may be desirable to have smaller functions.

B.5.b 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 principle patterns 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 III-B.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 accomplished this by splitting the interpreter into smaller functions that are targeted for specific tasks, such as encoding or decoding individual data types. This strategy 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.

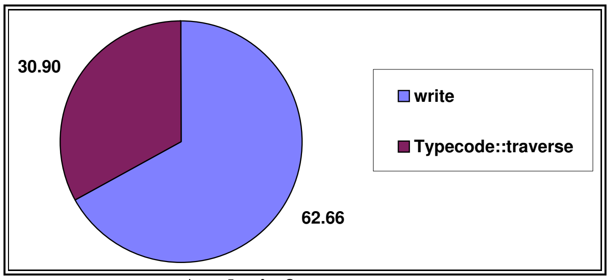
The `decode_sequence` method could support more specialized methods, *e.g.,* `decode_sequence_long` to decode a `sequence` of `long`s, by further decomposing it. A smaller piece of code that demonstrates high locality of reference is more likely to reside within processor caches.

The optimization principle pattern we employed here is similar to principle pattern 3 from Table I, 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.

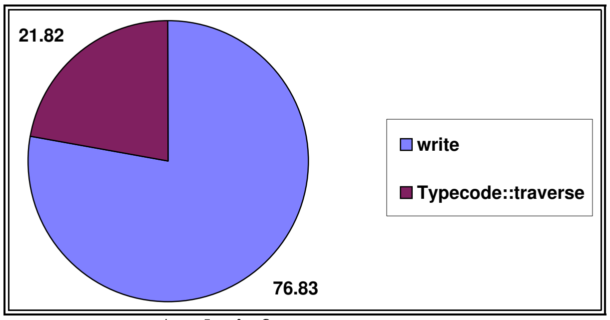
2. Using "outlining" to optimize for the frequently executed case: Outlining [26] is used to remove gaps that are introduced in the processor cache as a result of branch instructions arising from error handling code. Processor cache gaps are undesirable because they waste memory bandwidth and introduce 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.

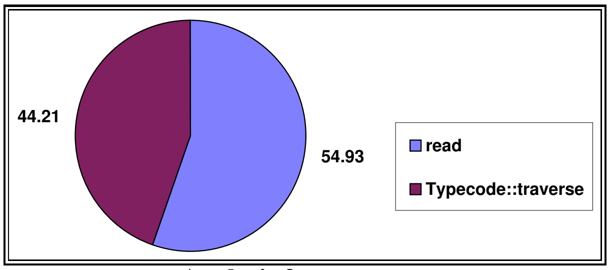


Analysis for `doubles`

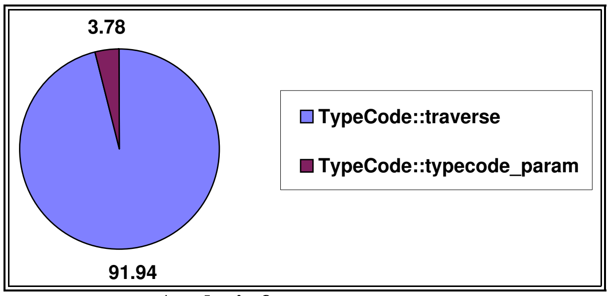


Analysis for `BinStructs`

Fig. 19. Sender-side Overhead After Applying the Second Optimization (getting rid of waste and precomputation)



Analysis for `doubles`



Analysis for `BinStructs`

Fig. 20. Receiver-side Overhead After Applying the Second Optimization (getting rid of waste and precomputation)

According to the 90-10 locality principle [25], 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 principle patterns 1 and 7 from Table I, which optimize for the expected case and optimize for the processor cache, respectively.

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

B.5.c Optimization step 3: receiver-side optimizations. Figures 19 and 20 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 21. Figures 22 and 23 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.

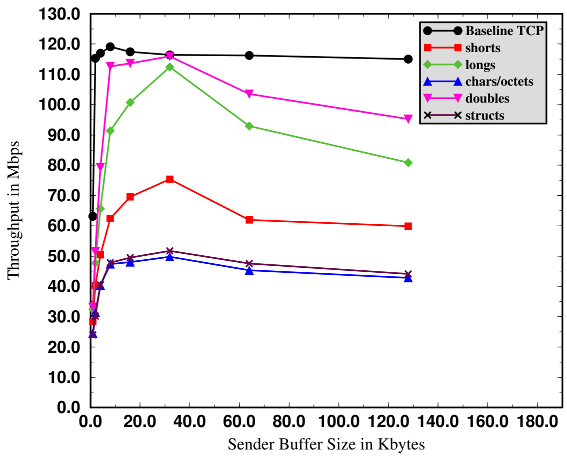


Fig. 21. Throughput After Applying the Third Optimization (receiver-side processor cache optimization)

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

TABLE VIII

SENDER-SIDE OVERHEAD AFTER APPLYING THE THIRD OPTIMIZATION (RECEIVER-SIDE PROCESSOR CACHE OPTIMIZATION)

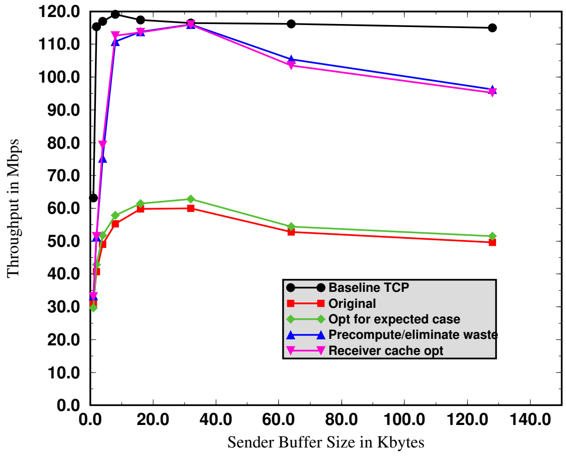


Fig. 22. Throughput Comparison for Doubles After Applying the Third Optimization (receiver-side processor cache optimization)

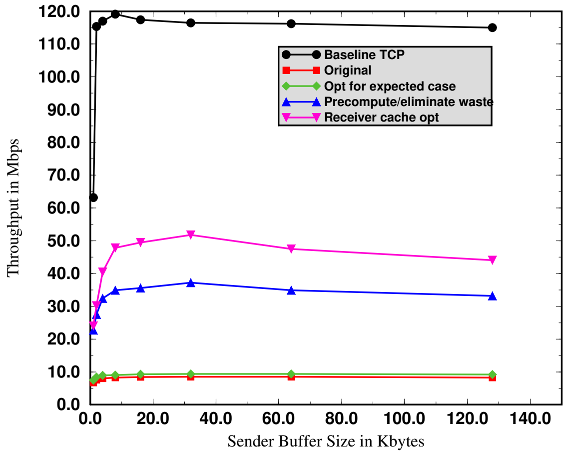


Fig. 23. Throughput Comparison for Structs After Applying the Third Optimization (receiver-side processor cache optimization)

Figures 24 and 25, and Tables VIII and IX illustrate the remaining high cost sender-side and receiver-side methods, respectively. These indicate that for primitive types, the cost of writing to the network and reading from the network becomes the primary contributor to the run-time costs. These results represent a substantial improvement over the original results presented in Section III-B.2 and illustrate that IIOP's marshaling overhead need not unduly limit ORB performance. For `BinStructs`, however, the sender-side, which was `write`-bound after the optimizations in step 2, spends a substantial amount of time (88%) in the interpreter. The receiver spends most of its time in the specialized functions such as `decode_sequence` (30%), and `decode_array` (26%). Analysis of the receiver-side revealed that the function call overhead decreased significantly compared to step 2.

B.5.d Optimization step 4: sender-side optimizations. The sender-side processor cache optimizations involve splitting the interpreter into smaller, specialized functions that can encode different OMG IDL data types.

The throughput measurements recorded after incorporating these optimizations are shown in Figure 26. Figures 27 and 28 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 29 and 30, and Tables X and XI illustrate the remaining high cost sender-side and receiver-side methods, re-

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

TABLE IX
RECEIVER-SIDE OVERHEAD AFTER APPLYING THE THIRD OPTIMIZATION (RECEIVER-SIDE PROCESSOR CACHE OPTIMIZATIONS)

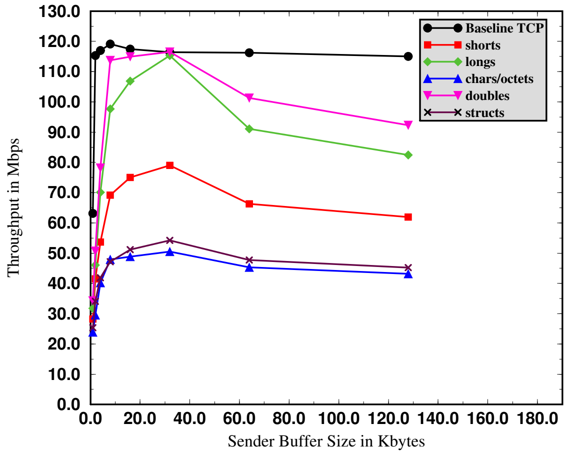


Fig. 26. Throughput After Applying the Fourth Optimization (sender-side processor cache optimization)

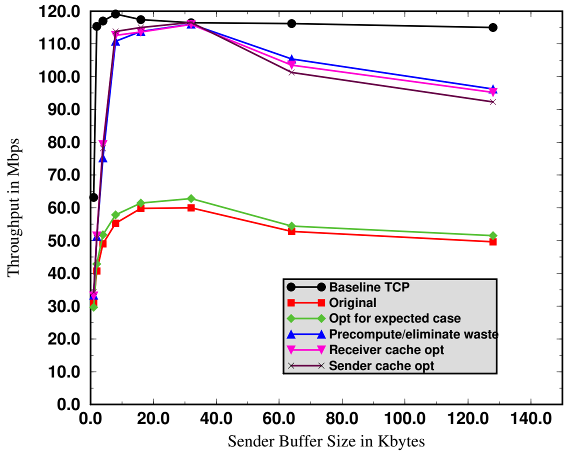
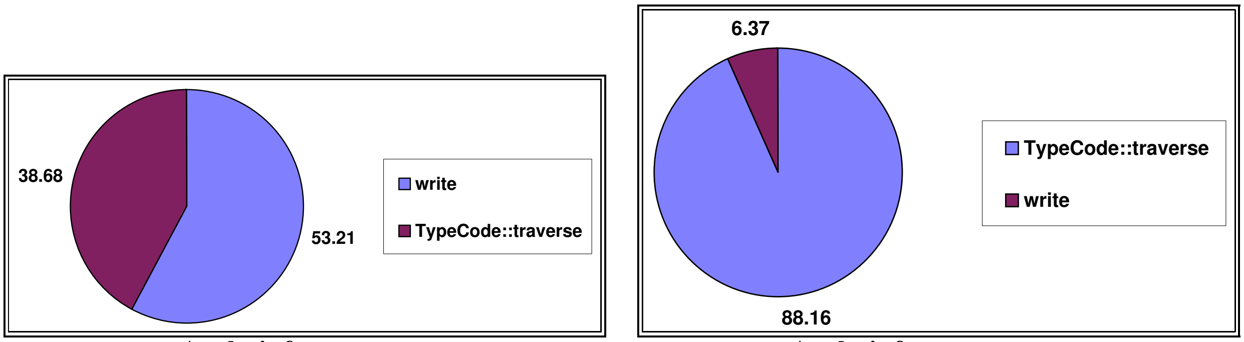
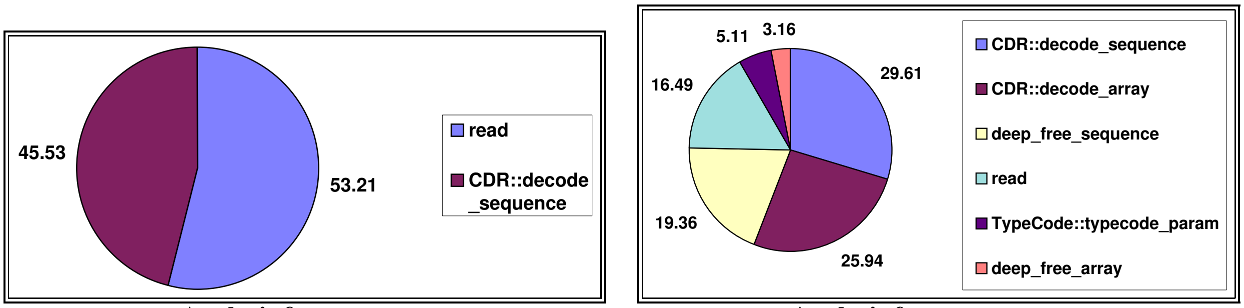


Fig. 27. Throughput Comparison for Doubles After Applying the Fourth Optimization (sender-side processor cache optimization)



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

Fig. 24. Sender-side Overhead After Applying the Third Optimization (receiver-side processor cache optimization)



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

Fig. 25. Receiver-side Overhead After Applying the Third Optimization (receiver-side processor cache optimizations)

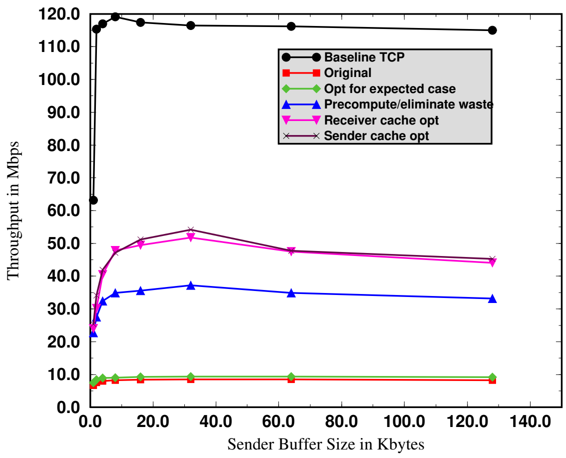


Fig. 28. Throughput Comparison for Structs After Applying the Fourth Optimization (sender-side processor cache optimization)

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

TABLE X

SENDER-SIDE OVERHEAD AFTER APPLYING THE FOURTH OPTIMIZATION (SENDER-SIDE PROCESSOR CACHE OPTIMIZATION)

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

TABLE XI

RECEIVER-SIDE OVERHEAD AFTER APPLYING THE FOURTH OPTIMIZATION (SENDER-SIDE PROCESSOR CACHE OPTIMIZATIONS)

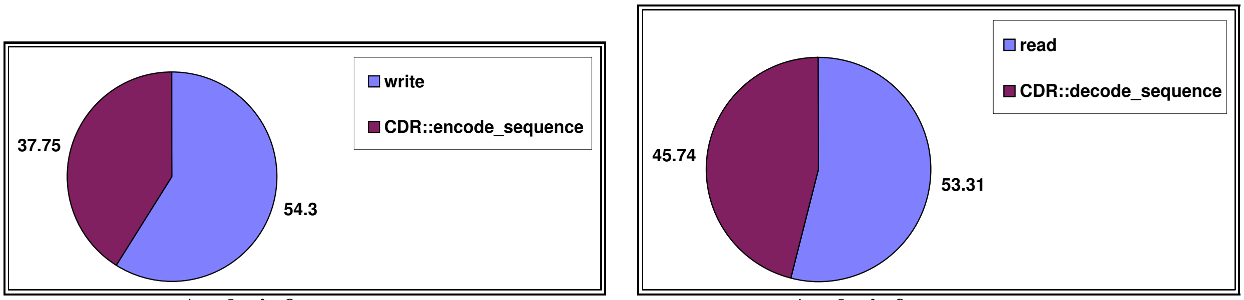
spectively.

IV. MINIMIZING FOOTPRINT OF IDL COMPILER GENERATED STUBS AND SKELETONS

A. *Motivation*

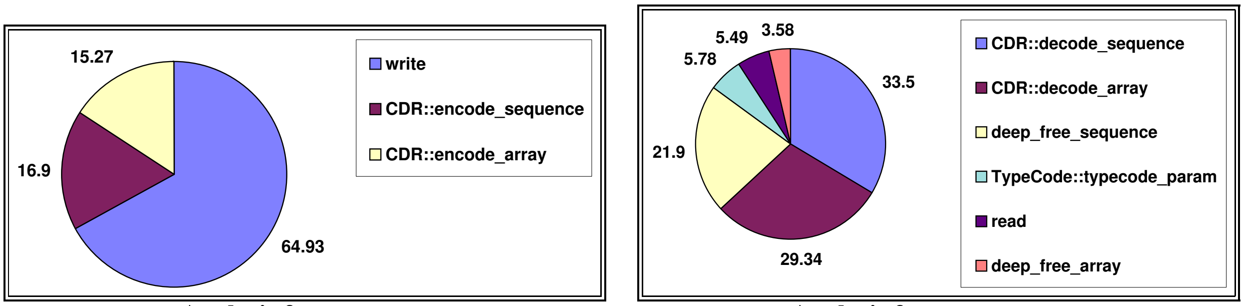
A OMG IDL compiler is responsible for generating *stubs* and *skeletons* that marshal and demarshal data types, respectively. In general, compiled (de)marshaling achieve higher run-time efficiency at the cost of increased memory footprint. Conversely, interpretive (de)marshaling can be used for applications that can afford to trade lower efficiency for a smaller memory footprint. Since neither approach is optimal for all distributed embedded multimedia applications, an OMG IDL compiler should produce stubs and skeletons that can use compiled *and/or* interpretive (de)marshaling.

This section describes the design of TAO's IDL compiler, which can selectively generate compiled and/or interpreted stubs and skeletons to enhance the optimization alternatives available to developers of embedded CORBA applications.



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

Fig. 29. Sender-side Overhead After Applying the Fourth Optimization (sender-side processor cache optimization)



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

Fig. 30. Receiver-side Overhead After Applying the Fourth Optimization (sender-side processor cache optimizations)

B. *Overview of TAO's IDL Compiler*

Figure 31 illustrates the interaction between the key components in the TAO's IDL Compiler. The TAO IDL compiler is

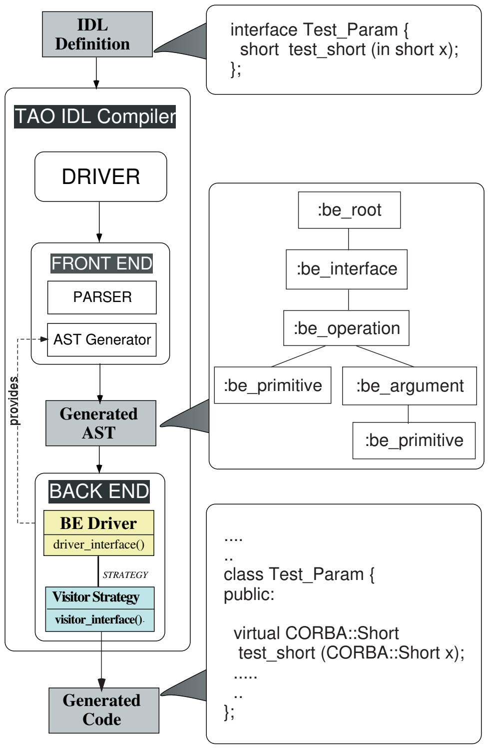


Fig. 31. Interactions Between Components in TAO's IDL Compiler

based on the freely available SunSoft IDL compiler[4] front-end, with many portability enhancements and with the defects removed. The front-end of the compiler parses OMG IDL input files and generates an abstract syntax tree (AST) that is stored entirely in memory.

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

The back-end of TAO's IDL compiler processes the AST and generates C++ source code that is optimized for TAO's IIOP protocol engine. In addition to generating interpreted stubs and skeletons, TAO's back-end can also produce compiled stubs and skeletons.

B.1 The Design of TAO's IDL Compiler Front-end

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

B.1.a OMG IDL parser:. The parser comprises a `yacc` specification of the OMG IDL grammar. The action for each grammar rule invokes methods of the AST node classes to build the AST.

B.1.b Abstract syntax tree generator:. Different nodes of the AST correspond to the different constructs of OMG IDL. The front-end defines a base class called `AST_Decl` that maintains information common to all AST node types. Specialized AST node classes like `AST_Interface` inherit from this base class, as shown in Figure 32.

In addition, the front-end defines the `UTL_Scope` class, which maintains scoping information, such as the nesting level and each component of the fully scoped name. All AST nodes representing OMG IDL constructs that can define scopes, such as `structs` and `interfaces`, also inherit from the `UTL_Scope` class.

B.1.c Driver program:. The driver program directs the parsing and AST generation process. It reads an input OMG IDL file and invokes the parser and the AST generator to create the in-memory AST and pass it to the back-end code generator.

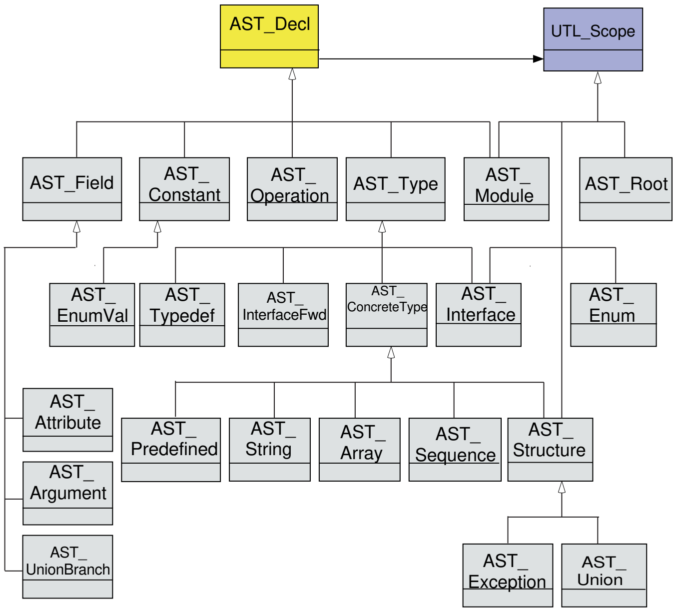


Fig. 32. TAO's IDL Compiler AST Class Hierarchy

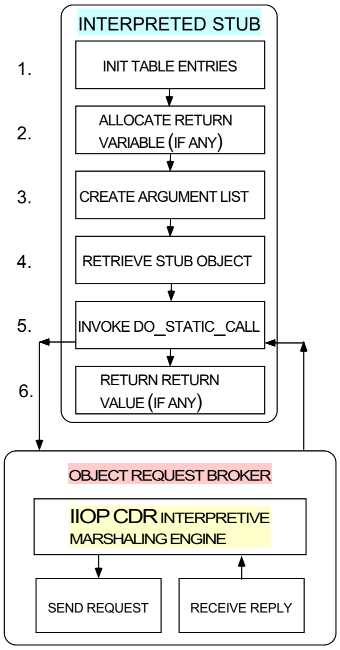


Fig. 33. Interpreted Stubs Generated by TAO's IDL Compiler

B.2 The Design of TAO's Back-end Code Generator

The original SunSoft IDL compiler front-end parses OMG IDL and generates a corresponding abstract syntax tree (AST). To create a complete OMG IDL compiler for TAO, we developed a back-end for the OMG IDL-to-C++ mapping. TAO's IDL compiler back-end uses several design patterns [22], such as *Abstract Factory*, *Strategy*, and *Visitor*. As a consequence of using patterns, TAO's IDL compiler back-end can be reconfigured readily to produce stubs and skeletons that use either compiled and/or interpretive (de)marshaling.

The interpretive stubs and skeletons produced by the back-end of TAO's IDL compiler integrate with TAO's highly optimized IIOP interpretive protocol engine. The interpreted stubs and skeletons generated by TAO's IDL compiler are explained below.

B.2.a Interpreted stubs:. The interpreted stubs produced by TAO's IDL compiler use a table-driven technique to pass parameters to TAO's interpretive IIOP (de)marshaling engine. The basic structure of an interpretive stub is shown in Figure 33. Each step is described below:

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

The `do_static_call` method described above is the interface to TAO's interpretive IIOP protocol engine. It takes the table describing the parameter types and the argument list as parameters.

The table-driven technique and the `do_static_call` interface were provided in the original SunSoft IIOP implementation.[5] However, since the SunSoft IIOP implementation did not have an IDL compiler each stub was hand-crafted. In contrast, the TAO IDL compiler automatically generates stubs that use interpretive (de)marshaling.

B.2.b Interpreted skeletons:. SunSoft IIOP skeletons use a Dynamic Skeleton Interface (DSI) [3] strategy to demarshal parameters. The basic (non-optimized) algorithm for an interpreted skeleton is shown in Figure 34. Each step is described below:

1. Create an `NVList`, which is a list of "name/value" pairs, to hold the parameters.
2. Heap allocate all the `return`, `inout`, and `out` parameters since they are marshaled back into the outgoing stream. The `in` parameters can be allocated on the run-time call stack of the skeleton.
3. Add each parameter value to the `NVList` using the operations provided by the ORB's DSI mechanism.
4. Use the DSI operation `arguments` to demarshal incoming parameters.
5. Make an upcall on the target object, passing it all the demarshaled parameters.
6. Create a `CORBA::Any` to hold the return value, if any.
7. Return from the skeleton and let the ORB Core handle the task of marshaling the `return`, `inout`, and `out` parameters, which are returned back to the client.

Memory for the `inout`, `out`, and `return` values is allocated on the heap, which is necessary because these parameters are marshaled into the outgoing IIOP Reply message after the

[5] We renamed SunSoft IIOP's `do_call` to `do_static_call`.

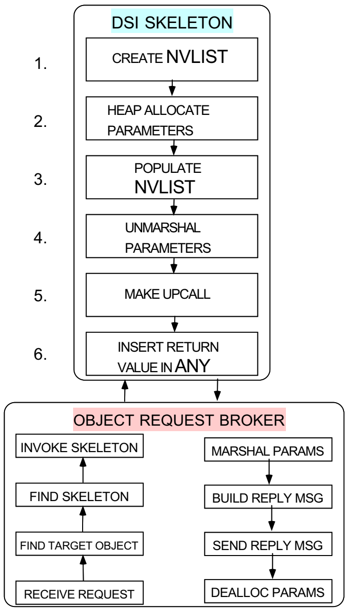


Fig. 34. Unoptimized Skeletons

call to the skeleton has returned. Therefore, it is not possible to allocate the parameters on the run-time stack of the skeleton. The heap allocated data structures are owned by the ORB and freed using an interpretive strategy similar to the interpretive (de)marshaling strategy [27].

The TAO IDL compiler, in contrast, produces an optimized version of these skeletons automatically. These optimizations include reducing the memory allocation overhead and reducing the size of the skeleton, as described in Section IV-C.

B.2.c Compiled stubs and skeletons:. The basic structure of a compiled stub is shown in Figure 35. The compiled stub's

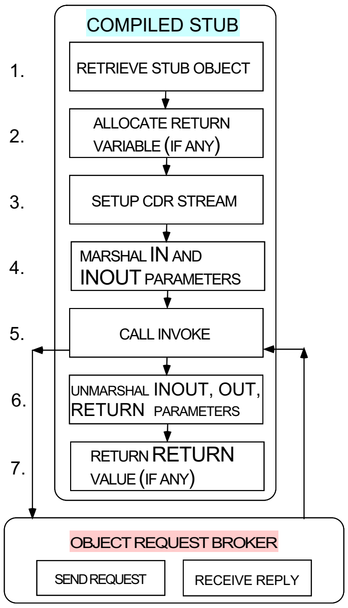


Fig. 35. Compiled Stubs/Skeletons

algorithm is very similar to the interpreted stub and works as follows:

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

The compiled stubs and skeletons use overloaded C++ iostream insertion and extraction operators, *i.e.*, `operator<<` and `operator>>`, respectively, to (de)marshal data types to/from the underlying CORBA Common Data Representation (CDR) stream. TAO's ORB Core provides these operators for primitive types. TAO's IDL compiler generates these operators for user-defined types.

A significant difference between the compiled skeleton and the interpreted skeleton is that no unnecessary heap allocation is required in the compiled skeleton. This is because a compiled skeleton has static knowledge of the types it (de)marshals. Moreover, all (de)marshaling of the parameters occur in the scope of the skeleton. In contrast, the interpretive skeletons in the DSI strategy require dynamic allocation since they marshal the `return`, `inout`, and `out` parameters in the ORB *after* the activation record of the skeleton has been destroyed.

C. Techniques for Optimizing Generated Stubs and Skeletons

As described in Section IV-A, it is imperative that an IDL compiler for embedded applications generate stubs and skeletons with exhibit small memory footprints. Therefore, we devised a technique to reduce the code size in TAO. Our code-size reduction techniques for interpreted stubs/skeletons are guided by the optimization principle patterns shown in Figure XII.

#	Principle Pattern
1	Factor out all common features
2	Avoid unnecessary heap allocation
3	Leverage compile time knowledge of data types

TABLE XII
OPTIMIZATION PRINCIPLE PATTERNS FOR SMALL FOOTPRINT STUBS/SKELETONS

Implementing these optimizations required the addition of several features to TAO's ORB Core. In particular, it was necessary to provide a pair of methods that marshal and demarshal parameters while the activation record of the stub and skeleton is active. This allows parameters to be allocated on the stack instead of from the heap, thereby eliminating dynamic memory allocation and locking.

Below, we describe other techniques we applied in TAO's IDL compiler to optimize the generated stubs and skeletons.

C.1 Optimizing DSI-style Interpretive Skeletons

As shown in Figure 34, each interpretive skeleton is required to create an `NVList` and populate it with parameters. In addition, memory is allocated from the heap rather than on the run-time stack for the `inout`, `out`, and `return` types. This is necessary since the marshaling of these parameters in the outgoing stream takes place after the call to the skeleton has returned.

Applications using the DSI must comply with the approach shown in Figure 34. However, the ORB Core can be modified to provide the necessary operations that is used by the IDL generated code. Applications cannot directly access these operations since they are protected.

Close scrutiny of early versions of TAO's IDL compiler-generated skeleton code revealed that each skeleton created an `NVList` and populated it with parameters. Similarly, the return value was stored in a `CORBA::Any` data structure. However, these common features can be factored out by TAO's ORB Core.

Based on our observations, we implemented a table-driven technique similar to the one used in the interpretive stubs. This table-driven approach defines two new interfaces in TAO's (de)marshaling engine that are similar to its `do_static_call` method. We could not reuse the `do_static_call` method since these two interfaces were required on the server-side "request" object. The space-efficient skeleton is shown in Figure 36.

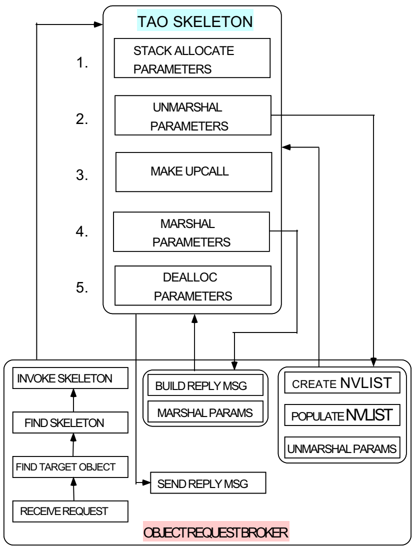


Fig. 36. Optimized DSI-interpretive Skeletons

The main benefit of the table-driven technique is that marshaling of outgoing parameters can occur while the activation record on the run-time stack frame of the skeleton is still valid. As a result, the `inout`, `out`, and `return` parameters need not be allocated from the heap. Instead, they can be allocated on the call stack using the same technique that TAO's compiled skeletons uses. In addition, if any outgoing types are mapped into C++ pointers, we can directly invoke the C++ `delete` operator rather than interpretively deallocating the memory.

D. Benchmarks Comparing the Performance and Footprint ofl Interpreted and Compiled (De)marshaling

This section describes our experiments comparing the size and performance of interpreted and compiled stubs and skeletons generated by TAO's IDL compiler.

D.1 Hardware and Software Platforms

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

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

D.2 Profiling Tools

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

The profile information for the empirical analysis was obtained using the `Quantify` 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.

D.3 Parameter Types for Stubs/Skeletons

We defined an *interface* in OMG IDL called `Param_Test` shown below.

```
interface Param_Test
{
  // Primitive types.
  short test_short (in short s1,
                    inout short s2,
                    out short s3);
  // Sequences and typedefs.
  typedef sequence<string> StrSeq;

  StrSeq test_strseq (in StrSeq s1,
                      inout StrSeq s2,
                      out StrSeq s3);
  // Other data types and operations
  // defined in a similar way.
};
```

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

D.4 Methodology

We measured the average throughput in terms of number of calls made per second by invoking each operation of the `Param_Test` interface 2,000 times. The servant object implements each operation by copying its `in` parameter into the `inout`, `out`, and `return` parameters. For complex data types, such as `struct_sequence`, this overhead becomes significant compared to the others.

We are primarily interested in measuring the performance of the stubs and skeletons. Therefore, all tests ran in loopback mode, which avoided network transfer overhead. However, we do measure OS effects like paging, context switching, and interrupts. In addition, delays incurred due to the run-time costs of the implementation of the operation by the servant object are also measured.

The code size of individual stubs and skeletons is measured using the GNU binary utility *objdump* and Windows NT's *dumpbin* as explained in Section IV-D.2.

D.5 Comparing Interpreted versus Compiled (De)marshaling

This section describes the results comparing the performance and code size of stubs and skeletons using interpretive and compiled form of (de)marshaling. As explained in Section IV-D.4, each operation of the `Param_Test` interface is invoked 2,000 times. The tests are performed in a loopback mode to avoid unnecessary network delays. The two-way average throughput of invoking the operations is reported. First, we report the performance results followed by comparison of the code sizes.

D.5.a Comparing twoway average throughput:. Figures 37, 38, and 39 depict the two-way average throughput in terms of calls made per second for invoking different methods of the `Param_Test` for 2,000 iterations for the UltraSPARC, a PC running NT, and PC running Linux, respectively. These figures

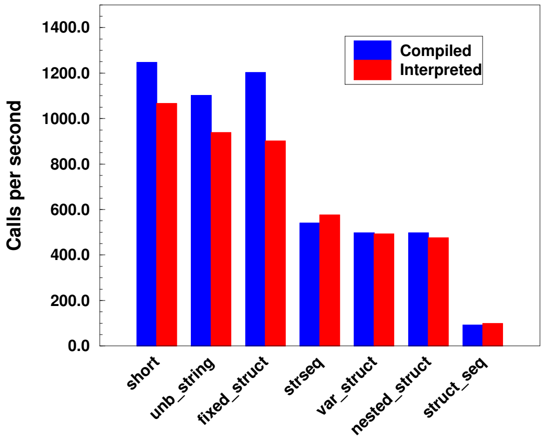


Fig. 37. UltraSPARC Performance

indicate that the two-way throughput of the interpreted stubs and skeletons is within 75 to 95% of the compiled stubs for primitive types such as `shorts`, and complex types, such as unbounded strings and fixed size structs. However, for other complex types such as `sequence` of `strings`, `sequence` of `structs`, variable-sized `structs`, and nested `structs`, the two-way throughput for interpreted stubs/skeletons is comparable or exceeds that of the compiled stubs. This is due to the optimizations we developed for the TAO ORB core and its interpretive IIOP (de)marshaling engine.

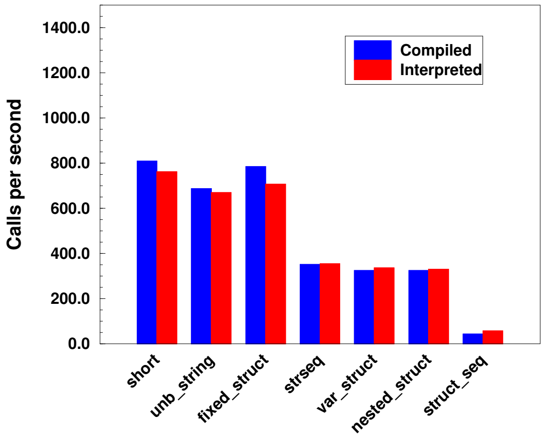


Fig. 38. NT Performance

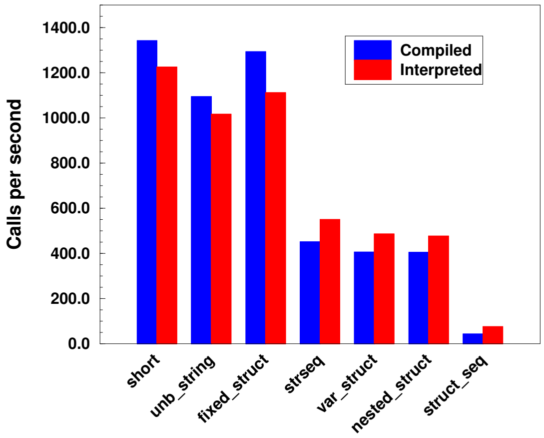


Fig. 39. Linux Performance

As mentioned in Section IV-D.4, these measurements include the effects of the OS, as well as the run-time costs of the operation implementations. These run-time operation implementation costs are more significant for the `test_struct_seq` case, where each `sequence` of `structs` has 9 variable-sized structs. Each variable-sized `struct` element in turn has two string members, each of length 128, and a `sequence` of `string` member. This member in turn has 9 `string` elements, each of length 128.

Figures 37, 38, and 39 indicate that the two-way throughput for complex user-defined data types such as variable-sized `structs` and sequences is significantly poorer compared to the primitive types irrespective of the type of (de)marshaling used. This is due to the costs of copying the `in` parameter into the `inout`, `out`, and `return` in the server-side implementation of the operation. Irrespective of the type of (de)marshaling used by the stubs and skeletons, however, the implementation of the operations is same in both cases. Thus, our comparisons of two-way throughput are valid.

The blackbox results presented in Figures 37, 38, and 39 do *not* convey the effects of the OS or the run-time costs of the operation implementations. To pinpoint precisely the run-time costs of the stubs and skeletons in (de)marshaling, we configured our profiling tool `Quantify` to measure only these costs. Table XIII illustrates the Quantify analysis for the `test_fixed_struct` and the `test_strseq` tests on the UltraSPARC platform.[6]

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

TABLE XIII
WHITEBOX ANALYSIS OF PERFORMANCE OF STUBS/SKELETONS ON ULTRASPARC

Table XIII indicates that for `fixed_struct` the compiled stubs and skeletons accounted for 47.76 msec compared to 283.56 msec required for the interpretive stubs and skeletons. This result explains why the compiled (de)marshaling is significantly better than the interpretive (de)marshaling for fixed size structs. Conversely, for `sequences` of `strings`, the compiled stubs and skeletons required 1,232.47 msec compared to only 785.65 msec by the interpretive stubs and skeletons. This result explains why the interpretive stubs perform better than the compiled stubs for all the data types that are `sequences` or have `sequences` as their members.

TAO's interpretive IIOP (de)marshaling engine is highly optimized for (de)marshaling `sequences`. It defines a generic base `sequence` class with virtual methods. For every user-defined `sequence`, the TAO IDL compiler generates a C++ class that inherits from this base `sequence` class. In accordance with the IDL-to-C++ mapping, the C++ class generated for the `sequences` overrides all the methods of the base class. The derived class does not define any data members since they are already defined in the base class.

TAO's IIOP interpreter (de)marshals `sequences` by invoking methods on the base class. At run-time, these virtual method calls are invoked on the appropriate derived class. This design significantly enhances `sequence` (de)marshaling performance by allowing TAO to use compile-time knowledge of the `sequence` and its element type for decoding. Thus, there is no need to interpretively decode the `sequence` using expensive typecode traversals.

D.5.b Comparing code size for stubs and skeletons:. Below, we describe the code size measurements we conducted for stubs and skeletons. As mentioned in Section IV-D.2, we used the GNU binary utility called `objdump` and NT's `dumpbin` to measure the individual code sizes. Table XIV depicts the code sizes for the overloaded operators used for (de)marshaling user-defined IDL data types. The code size of the `nested_struct`

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

TABLE XIV
SIZES OF OVERLOADED OPERATORS FOR COMPILED STUBS/SKELETONS ON ULTRASPARC

is only 88 bytes since internally it calls the overloaded operator for `var_struct`.

Tables XV and XVI illustrate the code sizes for the stubs and skeletons, respectively.

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

TABLE XV
STUB SIZES ON ULTRASPARC

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

TABLE XVI
SKELETON SIZES ON ULTRASPARC

We account for the size of the tables in the size of the stubs and skeletons using interpreted (de)marshaling. Therefore, the total size of the stub/skeleton is the size of the stub/skeleton and the size of the statically allocated tables.

For the compiled (de)marshaling, we account for the size of helper overloaded operator methods used to (de)marshal user-defined data types. Since these helper methods are not inlined by the compiler, we account for them only once. Thus, although the `nested_struct`'s helper calls the helper for `var_struct`, we do not add the latter's size to the size computation of the stub/skeleton of `nested_struct`.

Figures 40 and 41 illustrate this information graphically for the UltraSPARC platform. The helper operators for primitive

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

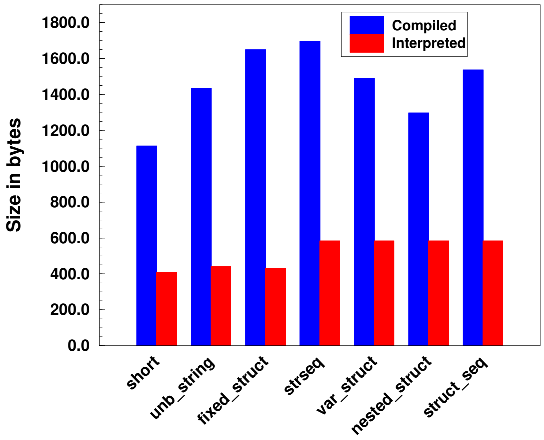


Fig. 40. UltraSPARC Stub Sizes

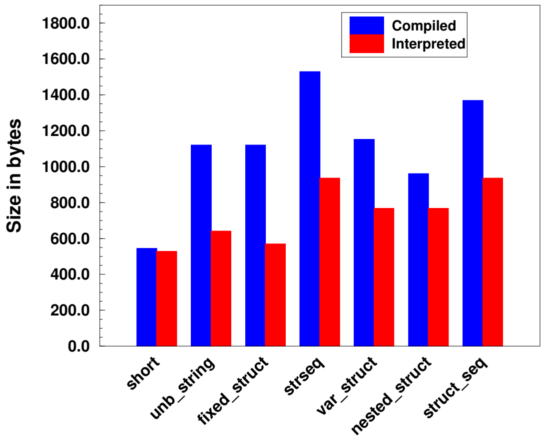


Fig. 41. UltraSPARC Skeleton Sizes

types are not generated by the IDL compiler since they are provided by the ORB core. Therefore, they are not shown.

Tables XV and XVI indicate that the stubs for interpretive (de)marshaling are much smaller than the ones for compiled (de)marshaling. As shown in Section IV-D.6, the interpretive stub sizes are ∼26-45% of the size of the compiled stubs. As shown in Section IV, in addition to explicitly (de)marshaling parameters, the compiled stub must initialize a GIOP/IIOP request message and invoke it.

For interpretive stubs, GIOP/IIOP request processing is performed by the `do_static_call` method in TAO's ORB Core. This method is an interpreter for stubs generated statically by TAO's IDL compiler. The size of skeletons for compiled (de)marshaling is relatively smaller than the compiled stubs since the `ServerRequest` object is already available.

The overloaded operators for primitives are provided by the ORB Core and no extra code is generated. Therefore, the skeleton code size for primitives like `shorts` and `longs` are comparable for interpretive and compiled (de)marshaling. However, for non-primitive data types, the skeleton code size for the interpreted (de)marshaling is between 50-80% of the compiled form.[7]

[7]The results of code size measurements for NT and Linux are not shown for lack of space. However, the results are similar to those for the UltraSparc.

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

TABLE XVII
COMPARISON OF INTERPRETIVE WITH COMPILED CODE ON ULTRASPARC IN PERCENTAGES

D.6 Summary of Comparisons

This section summarizes the results of Sections IV-D.5.a and IV-D.5.b. Table XVII illustrates how interpreted stubs and skeletons compared with the compiled versions for the UltraSPARC platform (all values are in percentages). Similar results are observed for the other two platforms.

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

D.7 Benefits of TAO's Interpretive Stubs and Skeletons

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

Table XVIII depicts the CORBA Object Services provided in the TAO release. We provide details on the number of user-defined structures, sequences, and total number of operations and/or attributes defined by their IDL definitions. We have not reported other data types, such as unions, enums, and exceptions defined by these IDLs.

As shown in Sections IV and IV-D, the total size of all the stubs/skeletons using compiled form of (de)marshaling will exceed that of interpretive (de)marshaling as the number of operations and user-defined types increase.

Table XVIII shows examples of standard CORBA Object Services, such as the Trading service and Naming service, as well as several services supported by TAO, such as a real-time Scheduling service [8]. As shown in the table, the IDL definitions for these services define a very large number of operations and/or attributes. The total size of stubs and skeletons using compiled (de)marshaling is significantly greater than that of interpretive (de)marshaling.

In addition, for every user-defined type, the compiled form will produce overloaded operators to (de)marshal these types. As shown in Section IV-D, the performance of the interpreted stub/skeleton strategy is comparable to, or exceeds, the compiled strategy. However, the code size for inter-

Service	structures	sequences	op/attributes
`Trading`	11	7	63
`A/V Streams`	2	3	50
`Property`	3	5	33
`Events`	0	0	18
`Naming`	2	2	13
`Scheduling`	4	4	10
`Logging`	1	0	6
`LifeCycle`	0	1	6

TABLE XVIII
NUMBER OF OPERATIONS AND USER-DEFINED TYPES IN STANDARD OMG SERVICES

preted stubs/skeletons is much smaller than the compiled stubs/skeletons.

V. RELATED WORK

Techniques for optimizing communication middleware is an emerging field of study. Our research on CORBA focuses on optimizing communication middleware at multiple protocol layers and multiple levels of abstraction including the I/O subsystem, communication protocols, and higher-level CORBA implementation itself. This section compares our research on TAO with related work on optimizing protocol implementations, generating efficient stubs for (de)marshaling, and evaluating performance of OO middleware.

A. Related Work on Optimization Principle Patterns

This section describes results from existing work on protocol optimization based on one or more of the principle patterns in Table I.

A.1 Optimizing for the expected case

[28] describes a technique called *header prediction* that predicts the message header of incoming TCP packets. This technique is based on the observation that many members in the header remain constant between consecutive packets. This observation led to the creation of a template for the expected packet header. The optimizations reported in [28] are based on Principle Pattern 1, which *optimizes for the common case* and Principle Pattern 3, which is *precompute, if possible*. We present the results of applying these principle patterns to optimize IIOP in Sections III-B.3, III-B.4, and III-B.5.

A.2 Eliminating gratuitous waste

[29], [30], [31] 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 Pattern 2, which *gets rid of gratuitous waste*. We demonstrate the application of this principle pattern to IIOP in Section III-B.4 where we eliminated unnecessary calls to the `deep_free` method, which frees primitive data types. [31] cautions against improper use of ILP since this may increase processor cache misses.

A.3 Passing information between layers

Packet filters [32], [33], [34] are a classic example of Principle Pattern 6, which recommends *passing information between layers*. A packet filter demultiplexes incoming packets to the appropriate target application(s). Rather than having demultiplexing occur at every layer, each protocol layer passes certain information to the packet filter, which allows it to identify which packets are destined for which protocol layer. We applied Principle Pattern 6 for IIOP in Section III-B.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.

A.4 Moving from generic to specialized functionality

[35] 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 Pattern 2, which focuses on *eliminating gratuitous waste* and Principle Pattern 3, which *replaces generic schemes with efficient, special purpose ones*. We applied these principle patterns for IIOP in Section III-B.4 where we incorporated the `struct_traverse` logic and some of the `decoder` logic into the `TypeCode` interpreter.

A.5 Improving cache-affinity

[26] describes a scheme called "outlining" that when used improves processor cache effectiveness, thereby improving performance. We describe optimizations for processor cache in Section III-B.5.

A.6 Efficient demultiplexing

Demultiplexing routes messages between different levels of functionality in layered communication protocol stacks. Most conventional communication models, such as the Internet model or the ISO/OSI reference model, require some form of multiplexing to support interoperability with existing operating systems and protocol stacks. In addition, conventional CORBA ORBs utilize several extra levels of demultiplexing at the application layer to associate incoming client requests with the appropriate servant and operation. Layered multiplexing and demultiplexing is generally disparaged for high-performance communication systems [36] due to the additional overhead incurred at each layer. [34] describes a fast and flexible message demultiplexing strategy based on dynamic code generation. [37] evaluates the performance of alternative demultiplexing strategies for real-time CORBA.

Our results for latency measurements have shown that with increasing number of servants, the latency increases. This is partly due to the additional overhead of demultiplexing the request to the appropriate operation of the appropriate servant. TAO uses a de-layered demultiplexing architecture [37] that can select optimal demultiplexing strategies based on compile-time and run-time analysis of CORBA IDL interfaces.

B. Related Work on Presentation Layer Conversions

B.1 Interpretive versus compiled forms of (de)marshaling

SunSoft IIOP uses an interpretive (de)marshaling engine. An alternative approach is to use *compiled* (de)marshaling. A compiled (de)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.

[38] describes the tradeoffs of using compiled and interpreted (de)marshaling schemes. Although compiled stubs are faster, they are also larger. In contrast, interpretive (de)marshaling is slower, but smaller in size. [38] describes a hybrid scheme that combines compiled and interpretive (de)marshaling to achieve better performance. This work was done in the context of the ASN.1/BER encoding [39].

According to the SunSoft IIOP developers, interpretive (de)marshaling can decrease code size and increase the likelihood of the protocol interpreter remaining in the processor cache. As explained in Section VI, we have implemented a CORBA IDL compiler [40] for TAO that optionally generates compiled stubs and skeletons. This IDL compiler generates efficient stubs and skeletons by extending optimizations provided in USC [41] and "Flick" [18], 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++.

B.2 Presentation layer and data copying

The presentation layer is a major bottleneck in high-performance communication subsystems [29]. This layer transforms typed data objects from higher-level representations to lower-level representations (marshaling) and vice versa (demarshaling). In both RPC toolkits and CORBA, this transformation process is performed by client-side stubs and server-side skeletons that are generated by interface definition language (IDL) compilers. IDL compilers translate interfaces written in an IDL (such as Sun RPC XDR [42], DCE NDR, or CORBA CDR [3]) to other forms such as a network wire format. A significant amount of research has been devoted to developing efficient stub generators. We cite a few of these and classify them as below.

- *Annotating high level programming languages:* The Universal Stub Compiler (USC) [41] annotates the C programming language with layouts of various data types. The USC stub compiler supports the automatic generation of device and protocol header marshaling code. The USC tool generates optimized C code that automatically aligns data structures and performs network/host byte order conversions.
- *Generating code based on control flow analysis of interface specification:* [38] describes a technique of exploiting application-specific knowledge contained in the type specifications of an application to generate optimized marshaling code. This work tries to achieve an optimal tradeoff between interpreted code (which is slow but compact in size) and compiled code (which is fast but larger in size). A frequency-based ranking of application data types is used to decide between interpreted and compiled code for each data type. Our implementations of the stub compiler will be designed to adapt according to the runtime access characteristics of various data types and methods. The runtime usage of a given data type or method can be used to dynamically link in either the compiled or the interpreted version. Dynamic linking has been shown to be useful for mid-stream adaptation of protocol implementations [43].
- *Using high level programming languages for distributed applications:* [44] describes a stub compiler for the C++ language. This stub compiler does not need an auxiliary interface definition language. Instead, it uses the operator overloading feature of C++ to enable parameter marshaling. This approach enables distributed applications to be constructed in a straightforward manner. A drawback of using a programming language like C++ is that it allows programmers to use constructs (such as references or pointers) that do not have any meaning on the remote side. Instead, IDLs are more restrictive and disallow such constructs. CORBA IDL has the added advantage that it resembles C++ in many respects and a well-defined mapping from the IDL to C++ has been standardized.

B.3 Application level framing and integrated layer processing for communication subsystems

Conventional layered protocol stacks and distributed object middleware lack the flexibility and efficiency required to meet the quality of service requirements of diverse applications running over high-speed networks. One proposed remedy for this problem is to use *Application Level Framing* (ALF) [29], [45], [46] and *Integrated Layer Processing* (ILP) [29], [30], [43].

ILP ensures that lower layer protocols deal with data in units specified by the application. ILP provides the implementor with the option of performing all data manipulations in one or two integrated processing loops, rather than manipulating the data sequentially. [31] have shown that although ILP reduces the number of memory accesses, it does not reduce the number of cache misses compared to a carefully designed non-ILP implementation. Another limitation of ILP [31] is it is limited applicability to only non-ordering constrained protocol functions and its use of macros that restrict the protocol implementation from being adapted dynamically to changing application requirements.

As shown by our earlier benchmarks [15], [17], [16], conventional CORBA ORBs incur overhead stemming from many layers of software and large chains of function calls. We plan to use integrated layer processing to minimize the overhead of the various software layers. We are developing a factory of ILP based `inline` functions that are targeted to perform different functions. This design allows us to dynamically link required functionality as the requirements change and yet have an ILP-based implementation.

VI. Concluding Remarks

The embedded multimedia industry is growing rapidly and hand-held devices, such as PIMs, Web-phones, Web-TVs, and Palm computers, running multimedia applications, such as MIME-enabled email and Web browsing, are becoming ubiquitous. Ideally, these embedded multimedia applications can be developed using standard middleware components like CORBA,

Problem	Solution	Principle Pattern
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
Repetitive Code	Factor out common tasks and provide a single method to perform them	Factoring out common tasks
Excess Allocation and Deallocation	Try to allocate on function call stack	Avoid heap allocation

TABLE XIX

OPTIMIZATION PRINCIPLE PATTERNS APPLIED IN TAO

rather than building them from scratch. However, stringent constraints on the available memory in embedded systems impose a stringent limit on the footprint of the middleware, particularly the stubs and skeletons generated by CORBA IDL compilers.

This paper illustrates the benefits of applying optimization principle patterns to improve the performance of CORBA Inter-ORB Protocol (IIOP) middleware substantially. The principle patterns 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, (7) optimizing for processor cache affinity, (8) factoring out common tasks to reduce footprint, and (9) avoiding heap allocation as much as possible.*

Table XIX summarizes the problems encountered, the solutions we applied, and the optimization principle patterns that guided our solutions. The results of applying these optimization principle patterns 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.

We used the resulting optimized IIOP protocol engine as the basis for TAO. TAO's performance is competitive with existing ORBs [15], [16] using the static invocation interface (SII) and 2 to 4.5 times (depending on the data type) faster than existing ORBs using the dynamic skeleton interface (DSI) [17]. Our optimization results demonstrate empirically that performance of complex, performance-sensitive embedded multimedia software can be improved by a systematic application of optimization principle patterns.

This paper also compares the performance and code size of stubs and skeletons using interpretive and compiled (de)marshaling. Our empirical results indicate that compared with compiled stubs, interpretive stubs perform 86% for primitive types, 75% for fixed-size structures, and over 100% for data types with sequences. The memory footprint for user-defined types for interpreted stubs was 26-45% and for interpreted skeletons was 50-80% of the size of the compiled stubs and skeletons, respectively. For primitive types, the skeleton sizes were comparable. However, the interpreted stubs were $\sim$40% the size of the compiled stubs.

Our optimizations to the SunSoft IIOP protocol interpreter improved its performance substantially. It is now comparable to the performance of compiled stubs and skeletons. Likewise, TAO's IDL compiler optimizations result in stubs and skeletons whose footprint is substantially smaller than those using conventional compiled (de)marshaling [18].

We have integrated the optimized SunSoft IIOP implementation into the TAO real-time ORB [47]. TAO's IDL compiler generates optimized stubs and skeletons from IDL interfaces via an optimizing code generation back-end added to the SunSoft IDL compiler front end. These generated stubs and skeletons transform C++ methods into/from CORBA requests via our optimized IIOP implementation.

REFERENCES

[1] R. Johnson, "Frameworks = Patterns + Components," *Communications of the ACM*, vol. 40, Oct. 1997.

[2] M. Henning and S. Vinoski, *Advanced CORBA Programming With C++*. Addison-Wesley Longman, 1999.

[3] Object Management Group, *The Common Object Request Broker: Architecture and Specification*, 2.2 ed., Feb. 1998.

[4] G. Forman and J. Zahorhan, "The Challenges of Mobile Computing," *IEEE Computer*, vol. 27, pp. 38–47, April 1994.

[5] L. Chen and T. Suda, "Designing Mobile Computing Systems using Distributed Objects," *IEEE Communications Magazine*, vol. 14, February 1997.

[6] I. Pyarali, C. O'Ryan, D. C. Schmidt, N. Wang, V. Kachroo, and A. Gokhale, "Applying Optimization Patterns to the Design of Real-time ORBs," in *Proceedings of the* 5^{th} *Conference on Object-Oriented Technologies and Systems*, (San Diego, CA), USENIX, May 1999.

[7] Object Management Group, *Minimum CORBA - Joint Revised Submission*, OMG Document orbos/98-08-04 ed., August 1998.

[8] D. C. Schmidt, D. L. Levine, and S. Mungee, "The Design and Performance of Real-Time Object Request Brokers," *Computer Communications*, vol. 21, pp. 294–324, Apr. 1998.

[9] C. D. Gill, D. L. Levine, and D. C. Schmidt, "The Design and Performance of a Real-Time CORBA Scheduling Service," *The International Journal of Time-Critical Computing Systems, special issue on Real-Time Middleware*, 2000.

[10] T. H. Harrison, D. L. Levine, and D. C. Schmidt, "The Design and Performance of a Real-time CORBA Event Service," in *Proceedings of OOPSLA '97*, (Atlanta, GA), ACM, October 1997.

[11] F. Kuhns, D. C. Schmidt, and D. L. Levine, "The Design and Performance of a Real-time I/O Subsystem," in *Proceedings of the* 5^{th} *IEEE Real-Time Technology and Applications Symposium*, (Vancouver, British Columbia, Canada), pp. 154–163, IEEE, June 1999.

[12] D. C. Schmidt, S. Mungee, S. Flores-Gaitan, and A. Gokhale, "Alleviating Priority Inversion and Non-determinism in Real-time CORBA ORB Core Architectures," in *Proceedings of the* 4^{th} *IEEE Real-Time Technology and Applications Symposium*, (Denver, CO), IEEE, June 1998.

[13] Alistair Cockburn, "Prioritizing Forces in Software Design," in *Pattern Languages of Program Design* (J. O. Coplien, J. Vlissides, and N. Kerth, eds.), pp. 319–333, Reading, MA: Addison-Wesley, 1996.

[14] G. Varghese, "Algorithmic Techniques for Efficient Protocol Implementations ," in *SIGCOMM '96 Tutorial*, (Stanford, CA), ACM, August 1996.

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

[16] A. Gokhale and D. C. Schmidt, "Measuring and Optimizing CORBA Latency and Scalability Over High-speed Networks," *Transactions on Computing*, vol. 47, no. 4, 1998.

[17] A. Gokhale and D. C. Schmidt, "The Performance of the CORBA Dynamic Invocation Interface and Dynamic Skeleton Interface over High-Speed ATM Networks," in *Proceedings of GLOBECOM '96*, (London, England), pp. 50–56, IEEE, November 1996.

[18] E. Eide, K. Frei, B. Ford, J. Lepreau, and G. Lindstrom, "Flick: A Flexible, Optimizing IDL Compiler," in *Proceedings of ACM SIGPLAN '97 Conference on Programming Language Design and Implementation (PLDI)*, (Las Vegas, NV), ACM, June 1997.

[19] S. Vinoski, "CORBA: Integrating Diverse Applications Within Distributed Heterogeneous Environments," *IEEE Communications Magazine*, vol. 14, February 1997.

[20] D. C. Schmidt, S. Mungee, S. Flores-Gaitan, and A. Gokhale, "Software Architectures for Reducing Priority Inversion and Non-determinism in Real-time Object Request Brokers," *Journal of Real-time Systems*, To appear 2000.

[21] D. C. Schmidt and T. Suda, "An Object-Oriented Framework for Dynamically Configuring Extensible Distributed Communication Systems," *IEE/BCS Distributed Systems Engineering Journal (Special Issue on Configurable Distributed Systems)*, vol. 2, pp. 280–293, December 1994.

[22] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, MA: Addison-Wesley, 1995.

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

[24] P. S. Inc., *Quantify User's Guide*. PureAtria Software Inc., 1996.

[25] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*. Morgan Kauffman Publishers, San Francisco, California, 1990.

[26] D. Mosberger, L. L. Peterson, P. G. Bridges, and S. O'Malley, "Analysis of Techniques to Improve Protocol Processing Latency," in *Proceedings of SIGCOMM '96*, (Stanford, CA), pp. 73–84, ACM, August 1996.

[27] A. Gokhale and D. C. Schmidt, "Optimizing a CORBA IIOP Protocol Engine for Minimal Footprint Multimedia Systems," *Journal on Selected Areas in Communications special issue on Service Enabling Platforms for Networked Multimedia Systems*, vol. 17, Sept. 1999.

[28] D. D. Clark, V. Jacobson, J. Romkey, and H. Salwen, "An Analysis of TCP Processing Overhead," *IEEE Communications Magazine*, vol. 27, pp. 23–29, June 1989.

[29] D. D. Clark and D. L. Tennenhouse, "Architectural Considerations for a New Generation of Protocols," in *Proceedings of the Symposium on Communications Architectures and Protocols (SIGCOMM)*, (Philadelphia, PA), pp. 200–208, ACM, Sept. 1990.

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

[31] T. Braun and C. Diot, "Protocol Implementation Using Integrated Layer Processnig," in *Proceedings of the Symposium on Communications Architectures and Protocols (SIGCOMM)*, ACM, September 1995.

[32] S. McCanne and V. Jacobson, "The BSD Packet Filter: A New Architecture for User-level Packet Capture," in *Proceedings of the Winter USENIX Conference*, (San Diego, CA), pp. 259–270, Jan. 1993.

[33] M. L. Bailey, B. Gopal, P. Sarkar, M. A. Pagels, and L. 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.

[34] D. R. Engler and M. F. Kaashoek, "DPF: Fast, Flexible Message Demultiplexing using Dynamic Code Generation," in *Proceedings of ACM SIGCOMM '96 Conference in Computer Communication Review*, (Stanford University, California, USA), pp. 53–59, ACM Press, August 1996.

[35] P. Druschel and L. L. Peterson, "Fbufs: A High-Bandwidth Cross-Domain Transfer Facility," in *Proceedings of the* 14^{th} *Symposium on Operating System Principles (SOSP)*, Dec. 1993.

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

[37] A. Gokhale and D. C. Schmidt, "Evaluating the Performance of Demultiplexing Strategies for Real-time CORBA," in *Proceedings of GLOBECOM '97*, (Phoenix, AZ), IEEE, November 1997.

[38] P. Hoschka and C. Huitema, "Automatic Generation of Optimized Code for Marshalling Routines," in *IFIP Conference of Upper Layer Protocols, Architectures and Applications ULPAA'94*, (Barcelona, Spain), IFIP, 1994.

[39] International Organization for Standardization, *Information processing systems - Open Systems Interconnection - Specification of Basic Encoding Rules for Abstract Syntax Notation One (ASN.1)*, May 1987.

[40] A. Gokhale, D. C. Schmidt, and S. Moyer, "Tools for Automating the Migration from DCE to CORBA," in *Proceedings of ISS 97: World Telecommunications Congress*, (Toronto, Canada), IEEE Communications Society, September 1997.

[41] S. W. O'Malley, T. A. Proebsting, and A. B. Montz, "USC: A Universal Stub Compiler," in *Proceedings of the Symposium on Communications Architectures and Protocols (SIGCOMM)*, (London, UK), Aug. 1994.

[42] Sun Microsystems, "XDR: External Data Representation Standard," *Network Information Center RFC 1014*, June 1987.

[43] A. Richards, R. D. Silva, A. Fladenmuller, A. Seneviratne, and M. Fry, "The Application of ILP/ALF to Configurable Protocols," in *First International Workshop on High Performance Protocol Architectures, HIPPARCH '94*, (Sophia Antipolis, France), INRIA France, December 1994.

[44] G. Parrington, "A Stub Generation System for C++," *Computing Systems*, vol. 8, pp. 135–170, Spring 1995.

[45] I. Chrisment, "Impact of ALF on Communication Subsystems Design and Performance," in *First International Workshop on High Performance Protocol Architectures, HIPPARCH '94*, (Sophia Antipolis, France), INRIA France, December 1994.

[46] A. Ghosh, J. Crowcroft, M. Fry, and M. Handley, "Integrated Layer Video Decoding and Application Layer Framed Secure Login: General Lessons from Two or Three Very Different Applications," in *First International Workshop on High Performance Protocol Architectures, HIPPARCH '94*, (Sophia Antipolis, France), INRIA France, December 1994.

[47] D. C. Schmidt, A. Gokhale, T. Harrison, and G. Parulkar, "A High-Performance Endsystem Architecture for Real-time CORBA," *IEEE Communications Magazine*, vol. 14, February 1997.