# Techniques for Optimizing CORBA Middleware for Distributed Embedded Systems

Aniruddha Gokhale and Douglas C. Schmidt

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

This paper has been submitted to INFOCOM '99, March 21-25th, New York, New York.

## Abstract

*The distributed embedded systems industry is poised to leverage emerging real-time operating systems, such as Inferno, Windows CE 2.0, and Palm OS, to support mobile communication applications. Advances in off-the-shelf real-time operating systems provides an enabling framework for a wide range of mobile communication applications, such as such as electronic mail, Internet browsing, and network management. Ideally, these applications can be developed using standard middleware components like CORBA to improve their quality and reduce their cost and cycle time. However, stringent constraints on the available memory in embedded systems imposes a severe limit on the footprint of CORBA middleware.*

*This paper provides three contributions to the study and design of small footprint, real-time CORBA middleware. First, we describe the optimizations used to develop the protocol engine and CORBA IDL compiler provided by TAO, which is our real-time CORBA implementation. TAO's IDL compiler produces stubs that can use either compiled and/or interpretive marshaling. Second, we compare the performance and footprint of TAO IDL compiler-generated stubs and skeletons that use compiled and/or interpretive marshaling for a wide range of IDL data types. Third, we illustrate the benefits of the small footprint and efficiency of TAO IDL compiler-generated stubs and skeletons for a range of standard CORBA services implemented using TAO.*

*Our results comparing the performance of the compiled and interpretive stubs and skeletons indicate that the interpretive stubs and skeletons perform between 75-100% of the compiled stubs and skeletons for a wide range of data types. On the other hand, the code sizes for the interpreted stubs and skeletons were between 26-45% and 50-80% of the compiled stubs and skeletons, respectively. These results indicate a positive step towards implementing high performance, small footprint middleware for distributed embedded systems.*

**Keywords:** Communication Protocols and Software, Real-time CORBA, minimal ORB footprint, performance, hand-held devices.

## 1 Introduction

Existing markets for distributed embedded systems, in particular hand-held devices like Personal Digital Assistants (PDAs), are increasingly leveraging off-the-shelf real-time operating systems, such as Inferno, Windows CE 2.0, and Palm OS. Analysts estimate in excess of five million units of PDAs being sold by 1999. However, responses to recent surveys by users of PDAs indicate a dearth of software and applications. Many users require PDAs to possess high-speed, built-in data/cellular/fax modems that enable the PDA to be used as a cellular phone, a fax machine, Internet browser, as well as to send and receive email.

Adding efficient and predictable communication capability to distributed embedded systems yields many research challenges related to mobile computing [2]. These challenges include dealing with low bandwidth, heterogeneity in the network connections, frequent changes and disruptions in the established connections due to migrating targets, maintaining consistency of data, and dealing with heterogeneous architectures to which these devices can be docked. In addition to the mobility issues, the restrictions on the physical size and power consumptions of these devices constrains the amount of storage capabilities they can possess.

CORBA [18] is a distributed object computing middleware standard defined by the Object Management Group (OMG). CORBA is designed to allow clients to invoke operations on remote objects without concern for where the object resides or what language the object is written in [23]. In addition,

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

CORBA shields applications from non-portable details related to the OS/hardware platform they run on and the communication protocols and networks used to interconnect distributed objects. These benefits of CORBA make it ideally suited to provide core communication services for distributed embedded systems.

A key research challenge, however, is determining how to maintain a small footprint for the ORB middleware and the stubs and skeletons generated automatically by a CORBA IDL compiler. Likewise, the performance of the generated stubs should not be unduly compromised due to footprint constraints. To address these issues, the OMG has recently issued a Request for Proposals (RFPs) for a minimal-CORBA [16] implementation geared towards embedded systems and other special systems that have constraints on the available resources such as memory.

Our prior research on CORBA middleware has explored several dimensions of real-time ORB endsystem design including static [21] and dynamic [4] real-time scheduling, real-time request demultiplexing [8], real-time event processing [11], real-time I/O subsystem integration [20], and the real-time performance of various commercial and research ORBs [22] over ATM networks. This paper focuses on a previously unexamined point in the real-time ORB endsystem design space: *techniques for optimizing the footprint and performance of CORBA middleware in distributed embedded systems*.

To address these issues, this paper compares compiled and interpretive marshaling used by CORBA IDL stubs and skeletons in terms of their performance and footprint. The stubs and skeletons evaluated in this paper use a hybrid compiled/interpreted marshaling technique generated by the TAO [21] IDL compiler. TAO (The ACE ORB) is a real-time ORB that complies with OMG's recently adopted Portable Object Adapter standard [18]. TAO's IDL compiler-generated stubs use a highly optimized interpretive scheme [9] to marshal and demarshal data types.

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

This paper is organized as follows. For completeness, Section 2 presents a brief overview of the CORBA reference model. Section 3 summarizes the optimizations we developed for TAO's interpretive marshaling engine and describes the techniques used by TAO's IDL compiler so it generates stubs and skeletons that have a small footprint; Section 4 describes the experimental setup and the results of our benchmarks that compare the performance and code size of compiled and interpreted stubs and skeletons for a representative range of IDL data types; Section 5 describes related work; and Section 6 summarizes the results.

## 2 Overview of the CORBA ORB Reference Model

CORBA Object Request Brokers (ORBs) [23] allow clients to invoke operations on distributed objects without concern for:

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

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

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

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

**Hardware:** CORBA shields applications from side-effects stemming from differences in hardware such as storage layout and data type sizes/ranges.

Figure 1 illustrates the components in the CORBA reference model, all of which collaborate to provide the portability, interoperability, and transparency outlined above. Each component in the CORBA reference model is outlined below:

**Client:** This program entity performs application tasks by obtaining object references to objects and invoking operations on them. Objects can be remote or collocated relative to the client. Ideally, accessing a remote object should be as simple as calling an operation on a local object, *i.e.*, `object`$\rightarrow$`operation(args)`. Figure 1 shows the underlying components that ORBs use to transmit remote operation requests transparently from client to object.

**Object:** In CORBA, an object is an instance of an Interface Definition Language (IDL) interface. The object is identified by an *object reference*, which uniquely names that instance across servers. An *ObjectId* associates an object with its servant implementation, and is unique within the scope of an Object Adapter. An object has one or more servants associated with it that implement the interface.

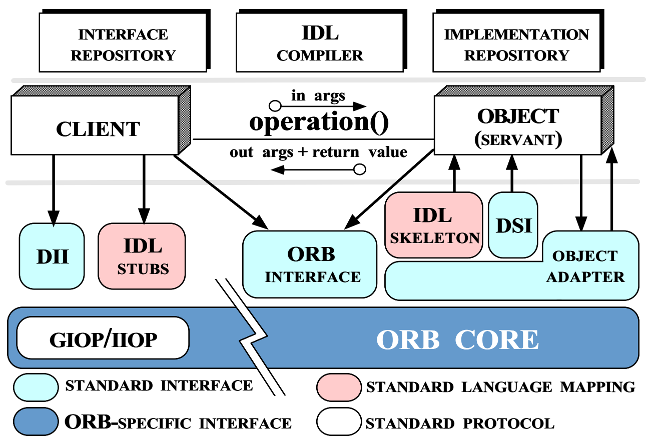

Figure 1: Components in the CORBA Reference Model

**Servant:** This component implements the operations defined by an OMG Interface Definition Language (IDL) interface. In languages like C++ and Java that support object-oriented (OO) programming, servants are implemented using one or more objects. In non-OO languages like C, servants are typically implemented using functions and `struct`s. A client never interacts with a servant directly, but always through an object.

**ORB Core:** When a client invokes an operation on an object, the ORB Core is responsible for delivering the request to the object and returning a response, if any, to the client. For objects executing remotely, a CORBA-compliant [18] ORB Core communicates via some version of the General Inter-ORB Protocol (GIOP), most commonly the Internet Inter-ORB Protocol (IIOP), which runs atop the TCP transport protocol. An ORB Core is typically implemented as a run-time library linked into both client and server applications.

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

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

**IDL Compiler:** An IDL compiler automatically transforms OMG IDL definitions into an application programming language like C++ or Java. In addition to providing programming language transparency, IDL compilers eliminate common sources of network programming errors and provide opportunities for automated compiler optimizations [1].

**Dynamic Invocation Interface (DII):** The DII allows clients to generate requests at run-time. This flexibility is useful when an application has no compile-time knowledge of the interface it is accessing. The DII also allows clients to make *deferred synchronous* calls, which decouple the request and response portions of twoway operations to avoid blocking the client until the servant responds. In contrast, SII stubs currently only support *twoway*, *i.e.*, request/response, and *oneway*, *i.e.*, request only operations, though the OMG has standardized an asynchronous method invocation interface in the recent Messaging Service specification [17].

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

**Object Adapter:** An Object Adapter associates a servant with objects, demultiplexes incoming requests to the servant, and dispatches the appropriate operation upcall on that servant. Recent CORBA portability enhancements [18] define the Portable Object Adapter (POA), which supports multiple nested POAs per ORB. Object Adapters enable ORBs to support various types of servants that possess similar requirements. This design results in a small and simple ORB that can still support a wide range of object granularities, lifetimes, policies, implementation styles, and other properties.

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

**Implementation Repository:** The Implementation Repository [12] contains information that allows an ORB to activate servers to process servants. Most of the information in the Implementation Repository is specific to an ORB or OS environment. In addition, the Implementation Repository provides a common location to store information associated with servers, such as administrative control, resource allocation, security, and activation modes.

## 3 Optimizing TAO's IDL Compiler

This section describes the design of TAO's IDL compiler and presents results from experiments that compare the optimization techniques we used to reduce the size of its generated stubs and skeletons without unduly reducing run-time ORB performance.

### 3.1 Overview of TAO's IDL Compiler

Figure 2 illustrates the design of the TAO IDL Compiler. The TAO IDL compiler is based on the freely available SunSoft IDL compiler front-end.[1] The front-end of the compiler parses OMG IDL input and generates an in-memory abstract syntax tree (AST). We customized the back-end to process the AST and generate C++ source code that is optimized for the interpretive IIOP protocol engine [9].

#### 3.1.1 The Design of TAO's IDL Compiler Front-end

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

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

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

In addition, the TAO IDL compiler defines a class called `UTL_Scope`, which maintains scoping information such as the nesting level and each component of the fully scoped name. All AST nodes representing CORBA IDL constructs that can define scopes (such as `structs` and `interfaces`) also inherit from the `UTL_Scope` class.

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

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

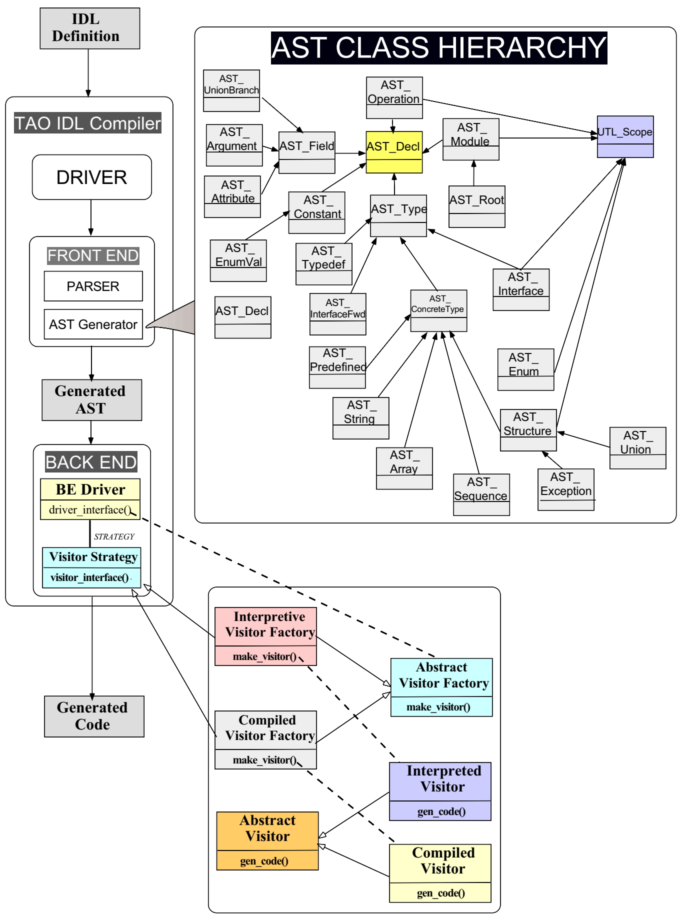

Figure 2: The TAO IDL Compiler

#### 3.1.2 The Design of TAO's Back-end Code Generator

The original SunSoft IDL compiler front-end parses OMG IDL and generates the abstract syntax tree (AST). To create a complete CORBA IDL compiler for TAO, we developed a back-end for the OMG IDL to C++ mapping. TAO's IDL compiler back-end uses several design patterns [3] such as *Abstract Factory*, *Strategy*, and *Visitor*. As a consequence, TAO's back-end can be reconfigured to produce stubs/skeletons that use either compiled and/or interpretive marshaling.

The back-end of TAO's IDL compiler produces stubs and skeletons that integrate with TAO's highly optimized IIOP interpretive protocol engine [9]. The interpreted stubs and skeletons generated by TAO's IDL compiler are explained below. We use the `test_short` method from the `Param_Test` interface shown in Appendix A to explain the behavior of the stubs and skeletons.

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

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

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

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

**Interpreted skeletons:** The skeletons use a Dynamic Skeleton Interface (DSI) approach for marshaling parameters. The basic (non-optimized) algorithm for a skeleton is shown in Figure 3 and described below:

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

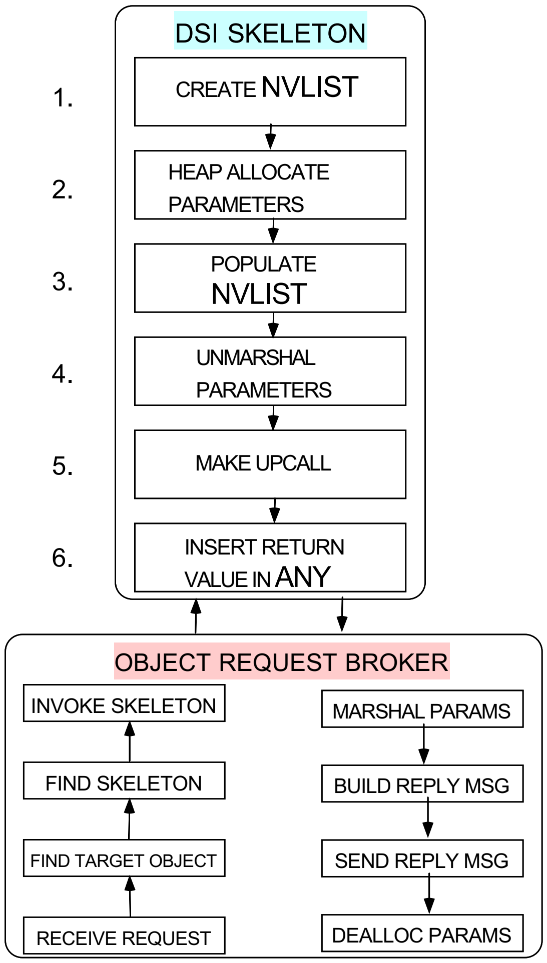

Figure 3: Unoptimized skeletons

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

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

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

**Compiled stubs and skeletons:** For this paper, the compiled stubs and skeletons are hand-crafted.[2] The basic structure of

[2]We are currently implementing a back-end for TAO IDL compiler that contains strategies for producing stubs and skeletons using compiled marshaling. This enhancement will be complete in time for the final version of the

a compiled stub is shown below. The skeleton's algorithm is very similar to the stub:

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

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

The compiled stubs and skeletons use overloaded C++ iostream insertion and extraction operators, *i.e.*, `operator<<` and `operator>>`, respectively, to marshal data types to/from the underlying CORBA Common Data Representation (CDR) stream. TAO's ORB Core provides these operators for primitive types. However, user-defined types are generated by the IDL compiler. In this paper, we hand-crafted these overloaded operators for the different user-defined types we tested.

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

## 3.2 Techniques for Reducing Stub/Skeleton Code Size

As described in Section 3.1 and shown in Appendix B, the size of the interpreted stubs and skeletons is large. However, due to stringent constraints on the available memory in hand-held devices, it is imperative to maintain a small footprint of the stubs and skeletons as well as the ORB core.

Therefore, we devised a technique to reduce the code size in TAO. Our code-size reduction techniques for interpreted stubs/skeletons are guided by the following three optimization principles shown in Table 1. 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 unmarshal parameters while the activation record of the stub and skeleton is active. Thus, parameters can now be allocated on the stack instead of from the heap, which reduces dynamic memory allocation and locking.

| **Optimization Principles** |
|---|
| Factor out all common features |
| Avoid unnecessary heap allocation |
| Leverage compile time knowledge of data types |

Table 1: Optimization Principles

### 3.2.1 Reducing the Size of Interpretive Stubs

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

### 3.2.2 Reducing the Size of Interpretive Skeletons

As shown in Figure 3, each interpretive skeleton is required to create an `NVList` and populate it with parameters. In addition, memory is allocated from the heap rather than on the call stack for the `inout`, `out`, and `return` types. This is necessary since the marshaling of these parameters in the outgoing stream takes place after the call to the skeleton has returned. Applications using DSI must comply with this style. However, the ORB Core can be modified to provide the necessary operations that is used on by the IDL generated code. Applications cannot directly access these operations since they are protected.

Close scrutiny of the skeleton code reveals that each skeleton must create an `NVList` and populate it with parameters. Similarly, any return value must be stored in a `CORBA::Any` data structure. These are common features that can be factored out into a method provided by the ORB Core. Based on this observation, we implemented a table-driven technique similar to the one used in the stubs. We defined two new interfaces to the marshaling engine that are similar to the `do_call` method. We could not reuse the `do_call` method since these two interfaces were required on the server-side "request" object. The space-efficient skeleton is shown in Figure 4. The correspond-

paper.

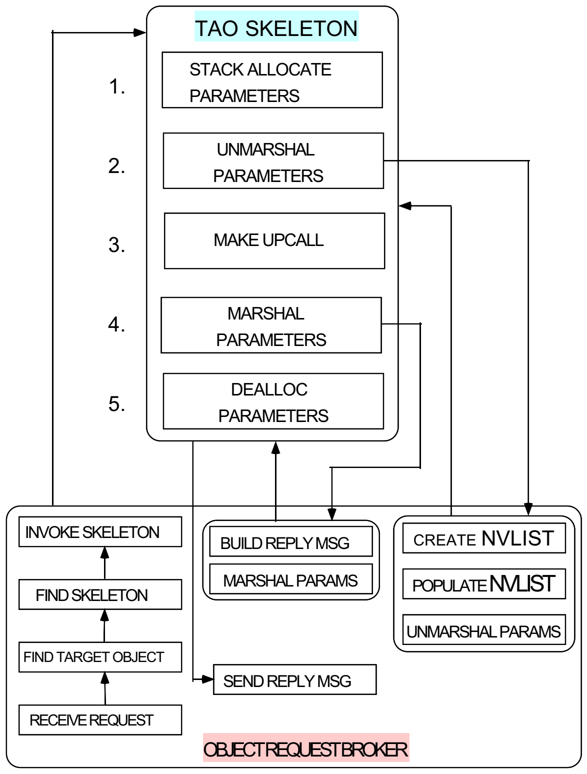

Figure 4: Optimized skeletons

ing code using this approach is shown Appendix B.5.

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

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

## 4 Experimental Results Comparing Interpreted and Compiled Marshaling

### 4.1 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.

### 4.2 Profiling Tools

The code size information for various methods reported in Section 4 is obtained using the GNU `objdump` binary utility on SunOS 5.5.1 and Linux. On Window NT, we used the `dumpbin` binary utility. In both cases, we used the *disasm* and *linenumbers* options to disassemble the object code and insert line numbers in the assembly listing, respectively. Code size for individual stubs/skeletons is reported by counting the total number of bytes of assembly level instructions produced. In addition, we used the UNIX `strip` utility to measure the total size of the object code after removing the symbols and other debug information.

The profile information for the empirical analysis was obtained using the `Quantify` [14] 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.

## 4.3 Parameter Types for Stubs/Skeletons

Appendix A provides the `Param_Test` IDL interface and its operations. All the operations test the four parameter passing modes (1) `in`, (2) `inout`, (3) `out`, and (4) `return` for a wide range of data types. The data types we tested include primitives such as `short`s, 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 twoway. Sequences are limited to a length of 9 elements and strings are 128 chars long.

## 4.4 Methodology

Each operation of the `Param_Test` interface is invoked 2,000 times. We measure the average latency for making the twoway call for the 2,000 iterations. Since we are interested in measuring the performance of the stubs and skeletons, all tests were run in the loopback mode. This way we avoided any network transfer overhead. However, OS effects such as paging, context switching, and interrupts are measured. In addition, delays incurred due to the run-time costs of the implementation of the operation by the servant object are also measured. The servant object implements each operation by copying its `in` parameter into the `inout`, `out`, and `return` parameters. For complex data types such as `struct_sequence`, this overhead becomes significant compared to the others.

One approach to eliminating OS effects in the measurements is to use co-located objects. However, even though TAO supports co-located objects, we cannot use them in our measurements. This is due to the fact that operations on co-located objects result in a direct C++ method call on the target object. Co-located objects entirely bypass the stubs/skeleton that perform the marshaling of data types. However, we are interested in measuring the overhead of marshaling. Forcing the co-located objects to use the stubs/skeletons required significant reengineering of the TAO ORB Core and the IDL Compiler. Therefore, we decided to approximate this behavior by

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

Table 2: Twoway Latency of Stubs/Skeletons on UltraSPARC Running SunOS5.5.1

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

Table 3: Twoway Latency of Stubs/Skeletons on Pentium Pro 200 Running Windows NT 4.0

running the tests in loopback mode. We configured our profiling tool `Quantify` (explained in Section 4.2) to measure only the overhead of the stubs and skeletons.

The code size of individual stubs and skeletons is measured using the GNU binary utility *objdump* and Windows NT's *dumpbin* as explained in Section 4.2.

## 4.5 Comparing Interpreted versus Compiled Marshaling

This section describes the results comparing the performance and code size of stubs and skeletons using interpretive and compiled form of marshaling. As explained in Section 4.4, 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 twoway average latency of invoking the operations is reported. First, we report the performance results followed by comparison of the code sizes.

### 4.5.1 Comparing Twoway Average Latencies

Tables 2, 3, and 4 depict the twoway average latency for invoking different methods of the `Param_Test` for 2,000 iterations for the UltraSPARC, a PC running NT, and PC running Linux, respectively. Figure 5 illustrates this information graphically for UltraSPARC.

These tables indicate that the twoway latency of the interpreted stubs and skeletons is within 75 to 95 % of the compiled stubs for primitive types such as `shorts`,

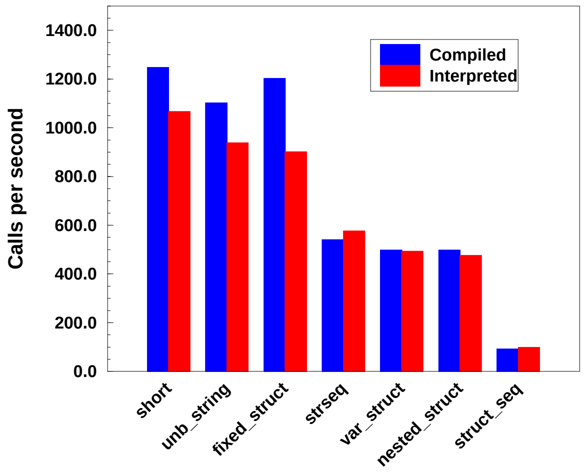

Figure 5: SPARC Performance

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

Table 4: Twoway Latency of Stubs/Skeletons on Pentium Pro 180 Running Linux

| 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 5: Whitebox Analysis of Performance of Stubs/Skeletons on UltraSPARC

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

As mentioned in Section 4.4, these measurements do not exclude the effects of the OS, as well as the runtime costs of the implementation of the operations. The runtime costs of the implementation of the operations is more significant for the `test_struct_seq` case. Each `sequence` of `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. The costs of copying the *in* parameter into the *inout*, *out*, and *return* is significant. However, irrespective of the type of marshaling used by the stubs and skeletons, the implementation of the operations is same in both cases. Hence, our comparisons of twoway latency are valid.

The blackbox results presented in Tables 2, 3, and 4 do *not* convey the effects of the OS, as well as runtime costs of the implementation of the operations. To pinpoint precisely the runtime costs of the stubs and skeletons in marshaling and demarshaling, we configured our profiling tool `Quantify` to measure only these costs. Table 5 illustrates the Quantify analysis for the `test_fixed_struct` and the `test_strseq` tests on the UltraSPARC platform.[3]

Table 5 indicates that for `fixed_struct`, the compiled stubs and skeletons accounted for 47.76 msec compared to 283.56 msec required for the interpretive stubs and skeletons. This explains why the compiled marshaling is significantly better than the interpretive marshaling for fixed size structs. On the other hand, for `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 explains why the interpretive stubs perform better than

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

| 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 6: Sizes of Overloaded Operators for Compiled Stubs/Skeletons on UltraSPARC

the compiled stubs for all the data types that are `sequence`s or have `sequence`s as their members.

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

### 4.5.2 Comparing Code Size for Stubs and Skeletons

This section describes the measurements of code size we did for the stubs and skeletons. As mentioned in Section 4.2, we used the GNU binary utility called `objdump` and NT's `dumpbin` to measure the individual code sizes.

Table 6 depicts the code sizes for the overloaded operators used for marshaling and demarshaling user-defined IDL data types. The code size of the `nested_struct` is only 88 bytes since internally it calls the overloaded operator for `var_struct`.

Tables 7 and 8 illustrate the code sizes for the stubs and skeletons, respectively. We account for the size of the tables in the size of the stubs and skeletons using interpreted marshaling. Hence the total size of the stub/skeleton is the size of the stub/skeleton and the size of the statically allocated tables. Similarly, for the compiled marshaling, we account for the size of helper overloaded operator methods used to marshal/demarshal user-defined data types. Since these helper methods are not inlined by the compiler, we account for them only once. Thus, although the `nested_struct`'s

| 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 7: Stub Sizes on UltraSPARC

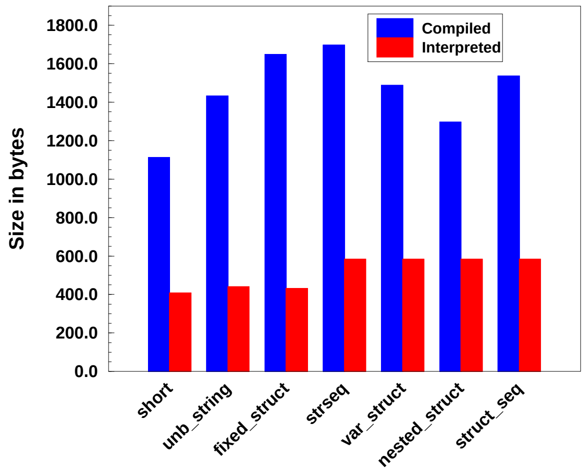

Figure 6: SPARC Stub Sizes

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 6 and 7 illustrate this information graphically for the UltraSPARC platform.

Tables 7 and 8 indicate that the stubs for interpretive marshaling are much smaller than the ones for compiled marshaling. As shown in Section 4.6, the interpretive stub sizes are roughly 26-45% of the size of the compiled stubs. As shown in Section 3, in addition to explicitly marshaling and demarshaling parameters, the compiled stub must initialize a GIOP/IIOP request message and invoke it. On the contrary, for the interpretive stubs the `do_call` method provided by the ORB Core handles all this. The size of skeletons for compiled marshaling is relatively smaller since the `ServerRequest` object is already available. The skeleton code size for primitives such

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

Table 8: Skeleton Sizes on UltraSPARC

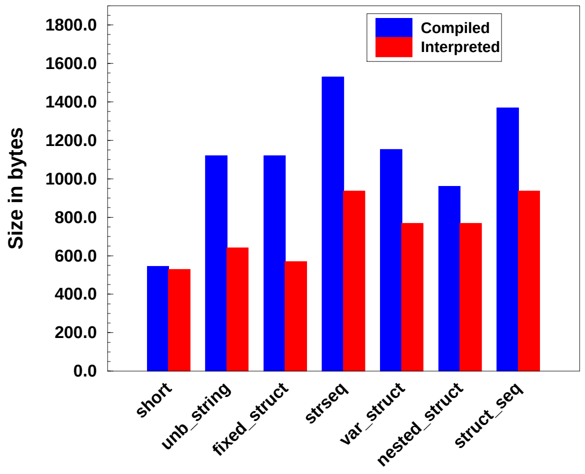

Figure 7: SPARC Skeleton Sizes

| 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 9: Comparison of Interpretive with Compiled Code on UltraSPARC in Percentages

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

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

## 4.6 Summary of Comparisons

This section summarizes the results of Sections 4.5.1 and 4.5.2. Table 9 illustrates how interpreted stubs and skeletons compared with the compiled versions for the UltraSPARC platform. Similar results are observed for the other two platforms. Appendix C provides the detailed measurements. All values are in percentages.

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

## 4.7 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 10 depicts a number of important CORBA higher-level services. We provide details on the number of user-defined structures, sequences, and total number of operations and/or attributes defined by their IDL definitions. We have not reported other data types such as unions, enums, and exceptions defined by these IDLs.

As shown in Sections 3 and 4, the total size of all the stubs and skeletons using compiled form of marshaling will exceed that of interpretive marshaling as the number of operations and user-defined types increase.

Table 10 shows examples of CORBA services such as the trading service, naming service, and others. As shown in the table, the IDL definitions for these services define a very large number of operations and/or attributes.[4] The total size of stubs and skeletons using compiled marshaling will far exceed that of interpretive marshaling.

| Service | structures | sequences | op/attributes |
|---|---|---|---|
| Trading | 11 | 7 | 63 |
| A/V Streams | 2 | 3 | 50 |
| Property | 3 | 5 | 33 |
| Naming | 2 | 2 | 13 |

Table 10: Number of Operations and User-defined Types in Standard OMG Services

In addition, for every user-defined type, the compiled form will produce overloaded operators to marshal and demarshal these types. As shown in Section 4, the performance of the interpreted stubs and skeletons is comparable or exceeds that of the compiled ones. At the same time, their code size is much smaller than the compiled ones.

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

## 5 Related Work

This section describes related work on CORBA performance measurements and presentation layer marshaling.

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

[13] describes the tradeoffs of using compiled and interpreted marshaling schemes. Although compiled stubs are faster, they are also larger. In contrast, interpretive marshaling is slower, but smaller in size. [13] describes a hybrid scheme that combines compiled and interpretive marshaling to achieve better performance. This work was done in the context of the ASN.1/BER encoding [15].

We are currently implementing a CORBA IDL compiler [10] that can generate compiled stubs and skeletons. Our goal is to generate efficient stubs and skeletons by extending optimizations provided in USC [19] and "Flick" [1], 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++.

**Related work on CORBA performance measurements:** [5, 6, 7] show that the performance of CORBA middleware implementations is relatively poor, compared to lower-level implementations using C/C++. The primary source of ORB-level overhead stems from marshaling and demarshaling. [5] measures the performance of the static invocation interface. [6] measures the performance of the dynamic invocation interface and the dynamic skeleton interface. [7] measures performance of CORBA implementations in terms of latency and support for very large number of objects.

However, the results of earlier CORBA benchmarking experiments were restricted to measuring the performance of communication between homogeneous ORBs. These tests do not explicitly measure the cost of marshaling for all the parameter passing modes. In addition, these tests were restricted to measuring the performance of compiled stubs and skeletons. Our earlier work [9] provides a detailed analysis and optimizations for an interpretive marshaling engine. However, that study did not compare interpretive and compiled marshaling. In addition, it did not comment upon the relative code sizes.

## 6 Conclusions

The distributed embedded systems industry, particularly hand-held devices, is poised to undergo a revolution with the emergence of operating systems, such as Inferno, Windows CE 2.0, and Palm OS, that provide support for mobile communications for applications. Ideally these applications can be developed using standard middleware components like CORBA to leverage the benefits of distributed object computing. However, stringent constraints on the available memory in embedded systems imposes a severe limit on the footprint of the middleware, particularly the stubs and skeletons generated by CORBA IDL compilers.

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

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

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

## References

[1] Eric Eide, Kevin Frei, Bryan Ford, Jay Lepreau, and Gary Lindstrom. Flick: A Flexible, Optimizing IDL Compiler. In *Proceedings of ACM SIGPLAN '97 Conference on Programming Language Design and Implementation (PLDI)*, Las Vegas, NV, June 1997. ACM.

[2] G. Forman and J. Zahorhan. The Challenges of Mobile Computing. *IEEE Computer*, 27(4):38–47, April 1994.

[3] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, MA, 1995.

[4] Christopher D. Gill, David L. Levine, and Douglas C. Schmidt. Evaluating Strategies for Real-Time CORBA Dy-

namic Scheduling. *submitted to the International Journal of Time-Critical Computing Systems, special issue on Real-Time Middleware*.

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

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

[7] Aniruddha Gokhale and Douglas C. Schmidt. Evaluating Latency and Scalability of CORBA Over High-Speed ATM Networks. In *Proceedings of the International Conference on Distributed Computing Systems*, Baltimore, Maryland, May 1997. IEEE.

[8] Aniruddha Gokhale and Douglas C. Schmidt. Evaluating the Performance of Demultiplexing Strategies for Real-time CORBA. In *Proceedings of GLOBECOM '97*, Phoenix, AZ, November 1997. IEEE.

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

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

[11] Timothy H. Harrison, David L. Levine, and Douglas C. Schmidt. The Design and Performance of a Real-time CORBA Event Service. In *Proceedings of OOPSLA '97*, Atlanta, GA, October 1997. ACM.

[12] Michi Henning. Binding, Migration, and Scalability in CORBA. *Communications of the ACM special issue on CORBA*, 41(10), October 1998.

[13] Phillip Hoschka and Christian Huitema. Automatic Generation of Optimized Code for Marshalling Routines. In *IFIP Conference of Upper Layer Protocols, Architectures and Applications ULPAA'94*, Barcelona, Spain, 1994. IFIP.

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

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

[16] Object Management Group. *Minimum CORBA - Request for Proposal*, OMG Document orbos/97-06-14 edition, June 1997.

[17] Object Management Group. *Messaging Service Specification*, OMG Document orbos/98-05-05 edition, May 1998.

[18] Object Management Group. *The Common Object Request Broker: Architecture and Specification*, 2.2 edition, February 1998.

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

[20] Douglas C. Schmidt, Rajeev Bector, David Levine, Sumedh Mungee, and Guru Parulkar. An ORB Endsystem Architecture for Statically Scheduled Real-time Applications. In *Proceedings of the Workshop on Middleware for Real-Time Systems and Services*, San Francisco, CA, December 1997. IEEE.

[21] Douglas C. Schmidt, David L. Levine, and Sumedh Mungee. The Design and Performance of Real-Time Object Request Brokers. *Computer Communications*, 21(4):294–324, April 1998.

[22] Douglas C. Schmidt, Sumedh Mungee, Sergio Flores-Gaitan, and Aniruddha Gokhale. Alleviating Priority Inversion and Non-determinism in Real-time CORBA ORB Core Architectures. In *Proceedings of the Fourth IEEE Real-Time Technology and Applications Symposium*, Denver, CO, June 1998. IEEE.

[23] Steve Vinoski. CORBA: Integrating Diverse Applications Within Distributed Heterogeneous Environments. *IEEE Communications Magazine*, 14(2), February 1997.

## A IDL Definition for Param_Test Example

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

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

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

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

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

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

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

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

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

  // Nested structs (We reuse the var_struct
  // definition above to make a very
  // complicated nested structure).
  struct Nested_Struct
  {
    Var_Struct vs;
  };

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

  // sequences of structs
  typedef sequence<Var_Struct> StructSeq;

  StructSeq test_struct_sequence
   (in StructSeq s1,
    inout StructSeq s2,
    out StructSeq s3);
};
```

## B Stubs and Skeletons

This section illustrates a stub and skeleton generated by TAO's IDL compiler for the `test_short` operation described in Appendix A.[5]

### B.1 Unoptimized Interpreted Stub

The `do_call` method shown below is the interface to the interpretive marshaling engine. `do_call` takes a variable number of parameters. The number of parameters is determined by the `TAO_Call_Data` argument. This supplies information such as the operation name ("test_short"), whether it is oneway or twoway, and a pointer to a table of parameters (`TAO_Param_Data`). The `TAO_Param_Data` data structure provides the `TypeCode` and parameter type for each parameter of the operation. `TypeCode`s are CORBA pseudo-objects that describe the format and layout of primitive and constructed IDL data types.

```
CORBA::Short Param_Test::test_short
  (CORBA::Short s1,
   CORBA::Short &s2,
   CORBA::Short_out s3,
   CORBA::Environment &env)
{
   static const TAO_Param_Data
   Param_Test_test_short_paramdata [] =
   {
     {CORBA::_tc_short, PARAM_RETURN, 0},
     {CORBA::_tc_short, PARAM_IN, 0},
     {CORBA::_tc_short, PARAM_INOUT, 0},
     {CORBA::_tc_short, PARAM_OUT, 0}
   };

   static const TAO_Call_Data
   Param_Test_test_short_calldata =
   {"test_short", 1, 4,
    Param_Test_test_short_paramdata,
    0, 0};

  CORBA::Short retval;
  STUB_Object *istub;

  this->QueryInterface (IID_STUB_Object,
                        (void **) &istub);
  // QueryInterface incremented refcount.
  this->Release ();
  istub->do_call
    (env,
    &Param_Test_test_short_calldata,
    &retval, &s1, &s2, &s3);
  return retval;
}
```

This stub first obtains the underlying stub object and then invokes the `do_call` method on it. The `do_call` internally creates the IIOP `Request` message and marshals all the *in* and *inout* parameters. It then invokes the remote procedure call. It blocks for the incoming reply if the IDL operation is twoway; otherwise it returns immediately. The return value, *inout*, and *out* parameters are demarshaled from the incoming IIOP `Reply` message.

### B.2 Unoptimized Skeleton

For the `test_short_skel` skeleton shown below, the skeleton first creates a `NVList`. An `NVList` is a CORBA pseudo object that holds a list of parameters. The parame-

[5]Some exception handling and error checking has been omitted to reduce space.

ter types, their `TypeCodes`, and memory to store their values are inserted into the `NVList`. The *in* and *inout* parameters are demarshaled via a call to the `params` method on `CORBA::ServerRequest`. The skeleton then makes the upcall on the target object passing the appropriate parameters to it.

```
void POA_Param_Test::test_short_skel
  (CORBA::ServerRequest &_tao_server_request,
   void *_tao_object_reference,
   void *context,
   CORBA::Environment &_tao_environment)
{
  CORBA::NVList_ptr nvlist;
  POA_Param_Test_ptr impl =
   (POA_Param_Test_ptr) _tao_object_reference;
  CORBA::Any *result;
  CORBA::Short *retval = new CORBA::Short;
  // Create an NV list and populate
  // it with typecodes.
  _tao_server_request.orb ()
    ->create_list (3, nvlist);

  // Add each argument according to the in,
  // out, inout semantics
  CORBA::Short s1;
  (void) nvlist->add_item
    ("s1", CORBA::ARG_IN,
     _tao_environment)->value ()
    ->replace (CORBA::_tc_short, &s1,
               0, _tao_environment);
  CORBA::Short *s2 = new CORBA::Short;
  (void) nvlist->add_item
    ("s2", CORBA::ARG_INOUT,
     _tao_environment)->value ()
    ->replace (CORBA::_tc_short,
               s2, 1, _tao_environment);
  CORBA::Short *s3 =
    new CORBA::Short;
  (void) nvlist->add_item
    ("s3",
     CORBA::ARG_OUT,
     _tao_environment)->value ()
    ->replace (CORBA::_tc_short,
               s3, 1, _tao_environment);
  // Parse the arguments.
  _tao_server_request.params
    (nvlist, _tao_environment);
  *retval = impl->test_short
    (s1, *s2, *s3, _tao_environment);

  // Store the result
  result = new CORBA::Any
   (CORBA::_tc_short, retval, 1);
  // Save the Any into the server request.
  _tao_server_request.result
    (result, _tao_environment);
}
```

## B.3 Compiled Stubs and Skeletons

This section illustrates the hand-crafted compiled stub and skeleton for the `test_short` operation.

```
CORBA::Short Param_Test::test_short
  (CORBA::Short s1,
   CORBA::Short &s2,
   CORBA::Short_out s3,
   CORBA::Environment &env)
{
  CORBA::Short retval = 0;
  IIOP_Object *istub;

  this->QueryInterface (IID_IIOP_Object,
                        (void **) &istub);
  // QueryInterface incremented refcount.
  this->Release ();

  // Set up a GIOP/IIOP message.
  TAO_GIOP_Invocation call
    (istub, ACE_OS::strdup ("test_short"), 1);
  env.clear ();
  // Setup a IIOP Request message.
  call.start (env);

  // Get the marshal stream.
  CDR &stream = call.stream ();

  // Insert parameters.
  stream << s1; stream << s2;

  // Now invoke the request.
  TAO_GIOP_ReplyStatusType status;
  CORBA::ExceptionList exceptions;

  exceptions.length = exceptions.maximum = 0;
  exceptions.buffer = (CORBA::TypeCode_ptr *) 0;

  // Send the request.
  status = call.invoke (exceptions, env);

  // Retrieve the parameter values.
  if (status == TAO_GIOP_NO_EXCEPTION)
    stream >> retval; stream >> s2; stream >> s3;
  return retval;
}
```

The skeleton is similar and is shown below.

```
void POA_Param_Test::test_short_skel
  (CORBA::ServerRequest &_tao_server_request,
   void *_tao_object_reference,
   void * context,
   CORBA::Environment &_tao_environment)
{
  POA_Param_Test_ptr impl
    = (POA_Param_Test_ptr) _tao_object_reference;
  CORBA::Short retval, s1, s2, s3;

  // Get the incoming CDR stream.
  CDR &instream = _tao_server_request.incoming ();

  // Retrieve parameters.
  instream >> s1; instream >> s2;

  // Make upcall.
  retval = impl->test_short
```

```
    (s1, s2, s3, _tao_environment);

  // Get the outgoing CDR stream.
  CDR &outstream = _tao_server_request.outgoing ();

  // Create a IIOP Reply message.
  _tao_server_request.init_reply (_tao_environment);

  // Marshal outgoing parameters.
  outstream << retval;
  outstream << s2; outstream << s3;
}
```

## B.4 Optimized Stub

```
// Call_Data and Param_Data tables are the same.
CORBA::Short  Param_Test::test_short
  (CORBA::Short s1,
   CORBA::Short &s2,
   CORBA::Short_out s3,
   CORBA::Environment &env)
{
  STUB_Object *istub = this->stubobj (env);
  if (istub) {
    CORBA::Short retval;
    istub->do_call
      (env,
      &Param_Test_test_short_calldata,
      &retval, &s1, &s2, &s3);
    return retval;
  }
  return 0;
```

## B.5 Optimized Skeletons

This section illustrates the optimized skeletons that TAO's IDL compiler generates. The skeleton for `test_short_skel` is shown.

```
void POA_Param_Test::test_short_skel
    (CORBA::ServerRequest &_tao_server_request,
     void *_tao_object_reference,
     void *context,
     CORBA::Environment &_tao_environment)
{
   static const TAO_Param_Data_Skel
   Param_Test_test_short_paramdata [] =
   {
     {CORBA::_tc_short, 0, 0},
     {CORBA::_tc_short, CORBA::ARG_IN, 0},
     {CORBA::_tc_short, CORBA::ARG_INOUT, 0},
     {CORBA::_tc_short, CORBA::ARG_OUT, 0}
   };

   static const TAO_Call_Data_Skel
   Param_Test_test_short_calldata =
   {"test_short",
     1,
     4,
     Param_Test_test_short_paramdata};

  POA_Param_Test_ptr impl
    = (POA_Param_Test_ptr) _tao_object_reference;
  CORBA::Short retval, s1, s2, s3;

  // Demarshal parameters.
  _tao_server_request.demarshal
     (_tao_environment,
      &Param_Test_test_short_calldata,
      &retval, &s1, &s2, &s3);

  // Make upcall.
  retval = impl->test_short
    (s1, s2, s3, _tao_environment);
  // Marshal outgoing parameters.
  _tao_server_request.marshal
    (_tao_environment,
     &Param_Test_test_short_calldata,
     &retval, &s1, &s2, &s3);
}
```

# C Comparison of Performance and Code Size for Windows NT and Linux

This section provides the cost of marshaling and the size of the generated stubs and skeletons for the Windows NT and Linux platforms.

| **Operator** | **Size** |
|---|---|
| **operator<< (char *)** | 111 |
| **operator>> (char *)** | 95 |
| **operator<< (fixed_struct)** | 207 |
| **operator>> (fixed_struct)** | 159 |
| **operator<< (strseq)** | 111 |
| **operator>> (strseq)** | 221 |
| **operator<< (var_struct)** | 63 |
| **operator>> (var_struct)** | 95 |
| **operator<< (nested_struct)** | 31 |
| **operator>> (nested_struct)** | 31 |
| **operator<< (struct_seq)** | 239 |
| **operator>> (struct_seq)** | 287 |

Table 11: Sizes of Overloaded Operators for Compiled Stubs/Skeletons on PC running Windows NT

| **Stub name** | **Interpreted size** | | | **Compiled size** | | |
|---|---|---|---|---|---|---|
| | stub | **table** | **total** | **stub** | **helper** | total |
| **test_short** | 205 | 88 | 293 | 570 | 0 | 570 |
| **test_ubstring** | 205 | 88 | 293 | 586 | 206 | 792 |
| **test_fixed_struct** | 205 | 88 | 293 | 570 | 366 | 936 |
| **test_strseq** | 269 | 88 | 357 | 664 | 332 | 996 |
| **test_var_struct** | 333 | 88 | 421 | 812 | 158 | 970 |
| **test_nested_struct** | 333 | 88 | 421 | 702 | 62 | 764 |
| **test_struct_seq** | 205 | 88 | 293 | 670 | 526 | 1,196 |

Table 12: Stub Sizes on PC running Windows NT

| Stub name | Interpreted size | | | Compiled size | | |
|---|---|---|---|---|---|---|
| | skel | **table** | **total** | **skel** | **helper** | total |
| **test_short_skel** | 253 | 88 | 341 | 143 | 0 | 143 |
| **test_ubstring_skel** | 333 | 88 | 421 | 239 | 206 | 445 |
| **test_fixed_struct_skel** | 285 | 88 | 373 | 191 | 366 | 557 |
| **test_strseq_skel** | 468 | 88 | 556 | 358 | 332 | 690 |
| **test_var_struct_skel** | 856 | 88 | 944 | 794 | 158 | 952 |
| **test_nested_struct_skel** | 867 | 88 | 955 | 726 | 62 | 788 |
| **test_struct_seq_skel** | 615 | 88 | 703 | 616 | 526 | 1,142 |

Table 13: Skeleton sizes on PC running Windows NT

| Operator | Size |
|---|---|
| **operator$<<$ (char *)** | 92 |
| **operator$>>$ (char *)** | 92 |
| **operator$<<$ (fixed_struct)** | 200 |
| **operator$>>$ (fixed_struct)** | 140 |
| **operator$<<$ (strseq)** | 284 |
| **operator$>>$ (strseq)** | 344 |
| **operator$<<$ (var_struct)** | 192 |
| **operator$>>$ (var_struct)** | 216 |
| **operator$<<$ (nested_struct)** | 24 |
| **operator$>>$ (nested_struct)** | 24 |
| **operator$<<$ (struct_seq)** | 168 |
| **operator$>>$ (struct_seq)** | 244 |

Table 14: Sizes of Overloaded Operators for Compiled Stubs/Skeletons on PC running Linux

| Stub name | Interpreted size | | | Compiled size | | |
|---|---|---|---|---|---|---|
| | stub | **table** | **total** | **stub** | **helper** | total |
| **test_short** | 104 | 88 | 192 | 452 | 0 | 452 |
| **test_ubstring** | 116 | 88 | 204 | 492 | 184 | 676 |
| **test_fixed_struct** | 108 | 88 | 196 | 452 | 340 | 792 |
| **test_strseq** | 212 | 88 | 300 | 576 | 628 | 1,204 |
| **test_var_struct** | 240 | 88 | 328 | 608 | 408 | 1,016 |
| **test_nested_struct** | 240 | 88 | 328 | 588 | 48 | 636 |
| **test_struct_seq** | 212 | 88 | 300 | 576 | 412 | 988 |

Table 15: Stub Sizes on PC running Linux

| Stub name | Interpreted size | | | Compiled size | | |
|---|---|---|---|---|---|---|
| | skel | **table** | **total** | **skel** | **helper** | total |
| **test_short_skel** | 136 | 88 | 224 | 180 | 0 | 180 |
| **test_ubstring_skel** | 228 | 88 | 316 | 276 | 184 | 460 |
| **test_fixed_struct_skel** | 132 | 88 | 220 | 192 | 340 | 532 |
| **test_strseq_skel** | 308 | 88 | 396 | 368 | 628 | 996 |
| **test_var_struct_skel** | 544 | 88 | 632 | 596 | 408 | 1,004 |
| **test_nested_struct_skel** | 544 | 88 | 632 | 596 | 48 | 644 |
| **test_struct_seq_skel** | 308 | 88 | 396 | 368 | 412 | 780 |

Table 16: Skeleton Sizes on PC running Linux