Techniques for Optimizing CORBA Middleware for Distributed Embedded 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 appeared in the proceedings of IEEE INFOCOM '99, New York, New York, March 21–25th, 1999.

Abstract—**The distributed embedded systems industry is poised to leverage emerging real-time operating systems, such as Inferno, Windows CE, EPOC, and Palm OS to support mobile communication applications, 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 memory available 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, embedded 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 CORBA Object 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. However, the code size for the interpreted stubs and skeletons was 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.**

I. 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, EPOC, and Palm OS. Analysts estimate that the shipment of embedded systems will grow five-fold from about 7 million units in 1995 to about 36 million units in 1999 [1]. However, many PDAs exhibit a dearth of communications software and applications, such as 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 they can possess.

The Common Object Request Broker Architecture (CORBA) is a distributed object computing middleware standard defined by the Object Management Group (OMG) [3]. 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 [4]. In addition, 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 the core communication infrastructure for distributed embedded systems.

A key research challenge for distributed embedded systems, 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 memory footprint constraints. To address these issues, the OMG has recently approved the *minimum CORBA* specification [5]. This specification is 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 many dimensions of real-time ORB endsystem design including real-time scheduling [6], real-time request demultiplexing [7], real-time event processing [8], real-time I/O subsystems [9] and operating systems [10], real-time ORB Core connection and concurrency architectures [11], and performance comparisons of various commercial ORBs [12]. 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 (de)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 (de)marshaling technique generated by the TAO [6] IDL compiler. TAO (The ACE ORB) is a real-time ORB that complies with OMG's recently adopted Portable Object Adapter standard [3]. TAO's IDL compiler-generated stubs

This work was supported in part by NSF grant NCR-9628218.
Work done by the first author while at Washington University.

use a highly optimized interpretive strategy [13] to (de)marshal data types.

Typically the code size for stubs/skeletons that use interpretive strategies is smaller in size compared to compiled strategies. In addition, interpreted stubs/skeletons are slightly slower than their counterparts. Since TAO's interpretive (de)marshaling engine is highly optimized, however, the performance of the stubs/skeletons is almost comparable to that of the compiled stubs for most data types. Moreover, the footprint of TAO IDL compiler-generated interpreted stubs and skeletons is significantly smaller than the compiled version, which makes it applicable for PDAs and other distributed embedded systems that have stringent memory restrictions.

This remainder of this paper is organized as follows: Section II describes the techniques used by TAO's IDL compiler so it generates stubs and skeletons that have a small footprint; Section III 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 IV describes related work; and Section V summarizes the results.

II. The Design of TAO's IDL Compiler

A. *Motivation*

A CORBA IDL compiler is responsible for generating *stubs* and *skeletons* that marshal and demarshal data types, respectively. In general, compiled (de)marshaling achieve higher runtime 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 applications, an 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.

B. *Overview of TAO's IDL Compiler*

Figure 1 illustrates the interaction between the key components in the TAO's IDL Compiler. The TAO IDL compiler is based on the freely available SunSoft IDL compiler[1] 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.

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 [13]. In addition to generating interpreted stubs and skeletons, TAO's back-end can also produce compiled stubs and skeletons.

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

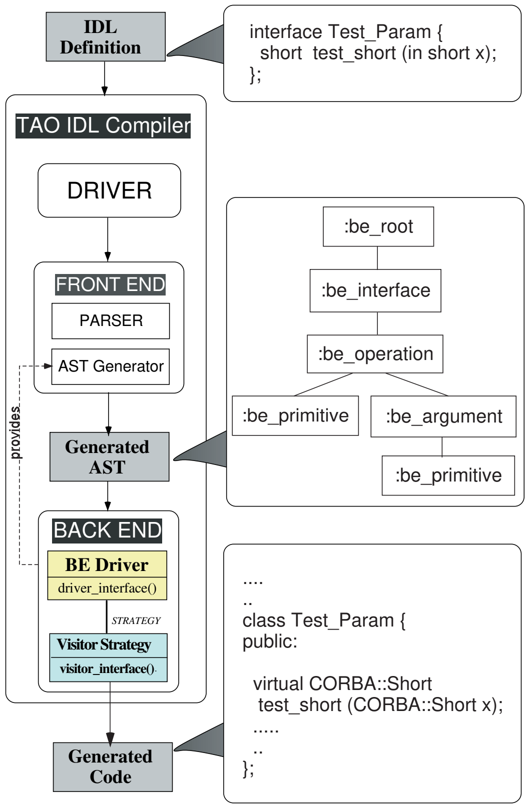

Fig. 1. Interactions Between Components in TAO's IDL Compiler

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 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 like `AST_Interface` inherit from this base class, as shown in Figure 2.

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 CORBA 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.

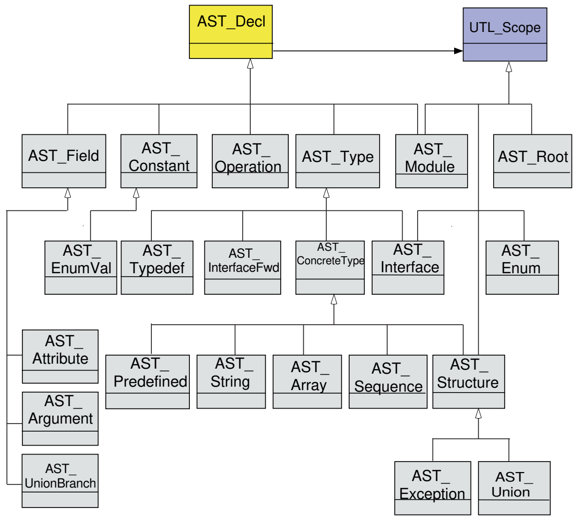

Fig. 2. TAO's IDL Compiler AST Class Hierarchy

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 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 [14], 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 [13]. 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 3. 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.

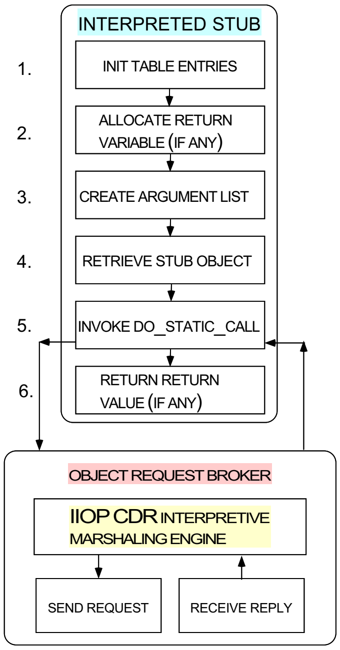

Fig. 3. Interpreted Stubs Generated by TAO's IDL Compiler

The table-driven technique and the `do_static_call` interface were provided in the original SunSoft IIOP implementation. 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 4. 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 call to the skeleton has returned. Therefore, it is not possible

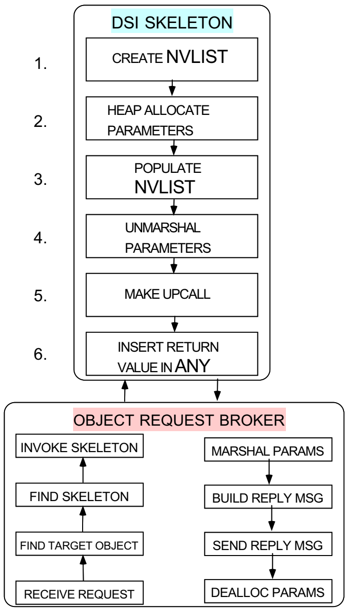

Fig. 4. Unoptimized Skeletons

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 [13].

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 II-C.

B.2.c Compiled stubs and skeletons. The basic structure of a compiled stub is shown in Figure 5. The compiled stub's

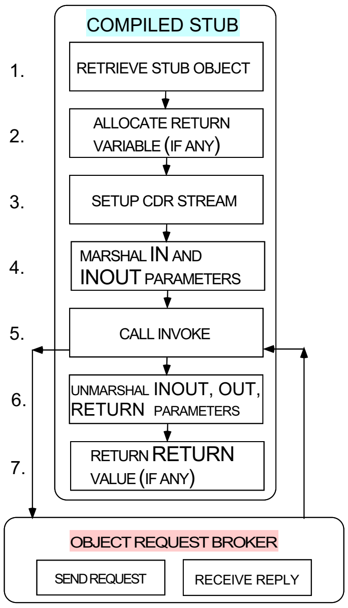

Fig. 5. Compiled Stubs/Skeletons

algorithm is very similar to the 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 II-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 following three optimization principles:

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

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 4, 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 4. 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, return value were 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 6.

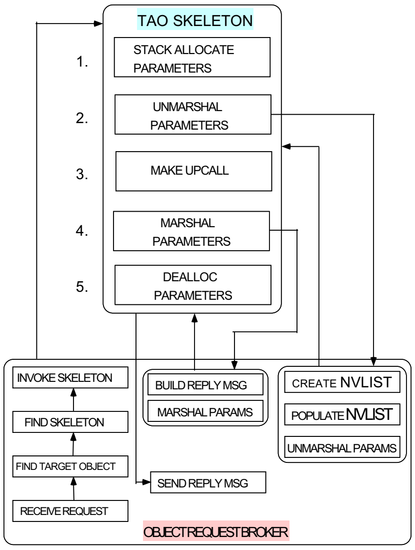

Fig. 6. Optimized DSI-interpretive Skeletons

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.

III. Benchmarks Comparing 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.

A. 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.

B. Profiling Tools

The code size information for various methods reported in Section III 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.

C. Parameter Types for Stubs/Skeletons

We defined an *interface* in OMG IDL called `Param_Test`. 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. 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 III-B.

E. 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 III-D, 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.

E.1 Comparing Twoway Average Throughput

Figures 7, 8, and 9 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.

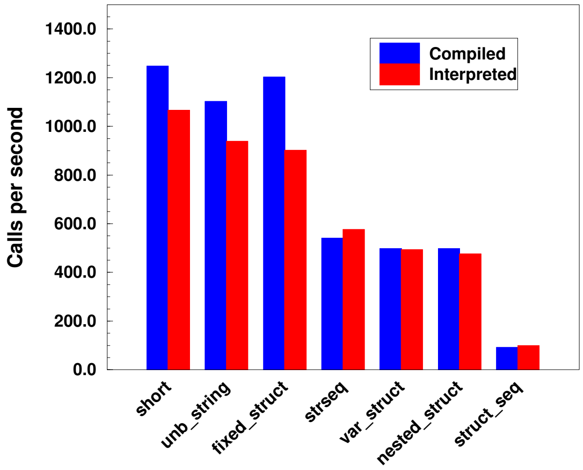

Fig. 7. UltraSPARC Performance

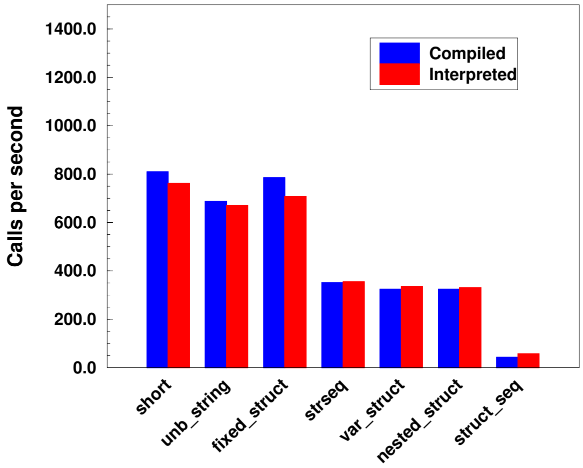

Fig. 8. NT Performance

These figures 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 a number of optimizations we developed for the TAO ORB core and its interpretive IIOP (de)marshaling engine [13].

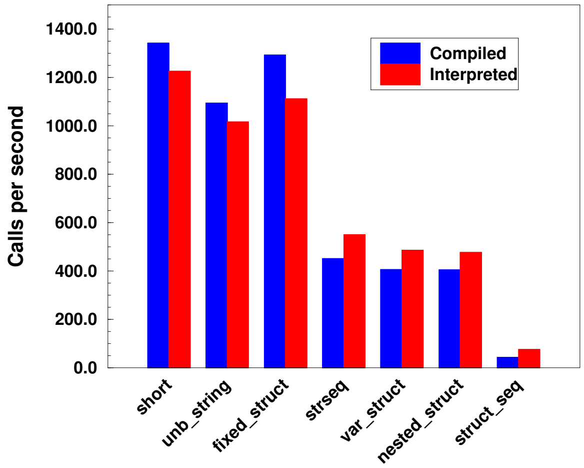

Fig. 9. Linux Performance

As mentioned in Section III-D, 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 7, 8, and 9 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 7, 8, and 9 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 I illustrates the Quantify analysis for the `test_fixed_struct` and the `test_strseq` tests on the UltraSPARC platform.[2]

Table I 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

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

TABLE I

WHITEBOX ANALYSIS OF PERFORMANCE OF STUBS/SKELETONS ON ULTRASPARC

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

`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 [13].

E.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 III-B, we used the GNU binary utility called `objdump` and NT's `dumpbin` to measure the individual code sizes.

Table II depicts the code sizes for the overloaded operators used for (de)marshaling user-defined IDL data types. The code

TABLE II

SIZES OF OVERLOADED OPERATORS FOR COMPILED STUBS/SKELETONS ON ULTRASPARC

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

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

Tables III and IV illustrate the code sizes for the stubs and skeletons, respectively. We account for the size of the ta-

TABLE III

STUB SIZES ON ULTRASPARC

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 IV

SKELETON 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

bles 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 10 and 11 illustrate this information graphically for the UltraSPARC platform. The helper operators for primitive

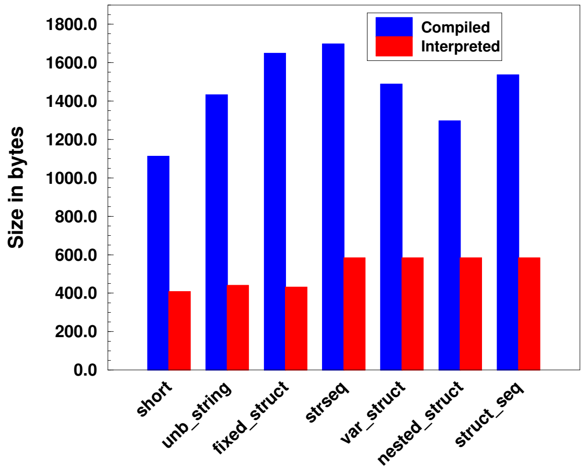

Fig. 10. UltraSPARC Stub Sizes

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

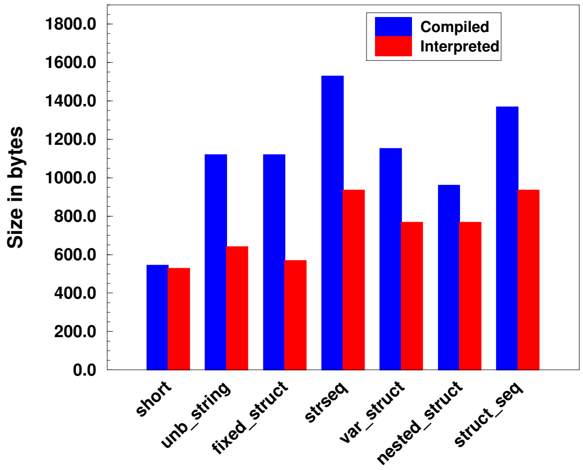

Fig. 11. UltraSPARC Skeleton Sizes

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

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

Tables III and IV indicate that the stubs for interpretive (de)marshaling are much smaller than the ones for compiled (de)marshaling. As shown in Section III-F, the interpretive stub sizes are ∼26-45% of the size of the compiled stubs. As shown in Section II, 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.[3]

F. Summary of Comparisons

This section summarizes the results of Sections III-E.1 and III-E.2. Table V 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.

[3]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.

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.

G. 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 VI 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 II and III, 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 VI 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 [6]. 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 great than that of interpretive (de)marshaling.

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

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

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

IV. Related Work

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

A. Related work on interpretive and compiled forms of marshaling:

TAO uses an interpretive (de)marshaling engine for its interpretive stubs. 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.

Reference [15] 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. Reference [15] describes a hybrid scheme that combines compiled and interpretive marshaling to achieve better performance. This work was done in the context of the ASN.1/BER encoding [16].

Our TAO IDL compiler can also generate compiled stubs and skeletons. Our goal is to generate efficient stubs and skeletons by extending optimizations provided in USC [17] 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. Related work on CORBA performance measurements:

Our earlier benchmarking experiments [12], [19], [7] showed that the performance of first-generation CORBA middleware implementations was relatively poor, compared with lower-level implementations using C/C++. The primary source of ORB-level overhead stems from marshaling and demarshaling.

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 [13] 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.

V. 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, EPOC, and Palm OS, that provide support for mobile communication 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 impose a stringent 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 and compiled stubs and skeletons are generated by the TAO IDL compiler.

Our results comparing the performance of the compiled and interpretive stubs indicate that the interpretive stubs perform $\sim$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 $\sim$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 size of the interpreted was $\sim$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. Thus, 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 OMG Minimum CORBA specification.

References

[1] Richard Comerford, "Pocket Computers Ignite OS Battle," *IEEE Spectrum*, vol. 35, no. 5, pp. 43–48, May 1998.

[2] G. Forman and J. Zahorhan, "The Challenges of Mobile Computing," *IEEE Computer*, vol. 27, no. 4, pp. 38–47, April 1994.

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

[4] Steve Vinoski, "CORBA: Integrating Diverse Applications Within Distributed Heterogeneous Environments," *IEEE Communications Magazine*, vol. 14, no. 2, February 1997.

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

[6] Douglas C. Schmidt, David L. Levine, and Sumedh Mungee, "The Design and Performance of Real-Time Object Request Brokers," *Computer Communications*, vol. 21, no. 4, pp. 294–324, Apr. 1998.

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

[8] 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.

[9] 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.

[10] David Levine, Sergio Flores-Gaitan, and Douglas C. Schmidt, "Measuring OS Support for Real-time CORBA ORBs," in *Proceedings of the* 4^{th} *Workshop on Object-oriented Real-time Dependable Systems*, Santa Barbara, CA, January 1999, IEEE.

[11] Douglas C. Schmidt, Sumedh Mungee, Sergio Flores-Gaitan, and Aniruddha Gokhale, "Software Architectures for Reducing Priority Inversion and Non-determinism in Real-time Object Request Brokers," *Journal of Real-time Systems*, To appear 1999.

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

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

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

[15] Phillip Hoschka and Christian Huitema, "Automatic Generation of Optimized Code for Marshalling Routines," in *IFIP Conference of Upper Layer Protocols, Architectures and Applications ULPAA'94*, Barcelona, Spain, 1994, IFIP.
[16] International Organization for Standardization, *Information processing systems - Open Systems Interconnection - Specification of Basic Encoding Rules for Abstract Syntax Notation One (ASN.1)*, May 1987.
[17] Sean W. O'Malley, Todd A. Proebsting, and Allen B. Montz, "USC: A Universal Stub Compiler," in *Proceedings of the Symposium on Communications Architectures and Protocols (SIGCOMM)*, London, UK, Aug. 1994.
[18] 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.
[19] Aniruddha Gokhale and Douglas C. Schmidt, "The Performance of the CORBA Dynamic Invocation Interface and Dynamic Skeleton Interface over High-Speed ATM Networks," in *Proceedings of GLOBECOM '96*, London, England, November 1996, IEEE, pp. 50–56.