The Design and Performance of a OMG CORBA IDL Compiler Optimized for Embedded Systems

Aniruddha Gokhale*
gokhale@research.bell-labs.com
Bell Labs, Lucent Technologies
600 Mountain Ave Rm 2B-434
Murray Hill, NJ 07974

Douglas C. Schmidt, Carlos O'Ryan, Alexander Arulanthu
{schmidt,coryan,alex}@cs.wustl.edu
Washington University
Campus Box 1045
St. Louis, MO 63130†

This paper was submitted to the LCTES '99 workshop at PLDI, Atlanta, Georgia, May 1-4, 1999.

Abstract

First-generation CORBA middleware was reasonably successful at meeting the demands of request/response, non-embedded applications with best-effort quality of service (QoS) requirements. However, supporting embedded systems with more stringent requirements, such as memory footprint limitations and real-time latency requirements, poses new challenges for next-generation CORBA middleware.

This paper provides two contributions to the design and optimization of OMG IDL compilers for embedded systems. First, we illustrate how the IDL compiler for TAO, our real-time CORBA implementation, has been optimized to support stringent time and space requirements of real-time embedded systems. Second, we describe the results of benchmarks that compare the impact of alternative IDL compiler code optimization strategies on the efficiency, predictability, and scalability of real-time ORBs.

Our results indicate that an ORB's IDL compiler must be highly configurable and adaptable to meet the QoS requirements for a wide range of embedded applications. In addition, we show how TAO's IDL compiler can be configured to perform predictably and scalably, which is essential to support real-time embedded applications.

Keywords: CORBA Middleware, Design Patterns, Tools, Performance.

1 Motivation

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, such as the Object Management Group's (OMG) Common Object Request Broker Architecture (CORBA) [1], 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.

The footprint of CORBA middleware is determined by (1) the size of the Object Request Broker (ORB) Core, and (2) the size of the stubs and skeletons generated by the Interface Definition Language (IDL) compiler. The issue of limitations on the footprint of the ORB Core for embedded systems is addressed by the OMG's Minimum CORBA [2] specification. However, there is no standard specification for minimizing the footprint of IDL compiler-generated stubs and skeletons. In this paper, therefore, we describe the optimization techniques we use to address this issue in the IDL compiler implementation for *The ACE ORB* (TAO), which is an open source, high-performance, and real-time CORBA implementation developed at Washington University [3].

A compiler for the OMG IDL parses IDL files containing CORBA interfaces and data types, and generates *stubs* and *skeletons* that marshal and demarshal data types, respectively, as shown in Figure 1. In addition, generated skeletons are also responsible for demultiplexing operation names carried in client requests to the associated methods on the servants, which implement target objects in a CORBA server. For embedded applications, particularly ones with real-time QoS requirements, it is essential for ORB operation demultiplexing to be efficient, predictable, and scalable.

Stubs/skeletons that use compiled (de)marshaling can achieve high efficiency at the cost of increased memory footprint [4]. Conversely, interpretive (de)marshaling is useful for applications that can trade less efficiency for a smaller memory footprint [5]. Since neither approach is optimal for all applications, an IDL compiler for embedded systems should be capable of producing stubs and skeletons that can use compiled and/or in-

*Work done by the author while at Washington University.

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

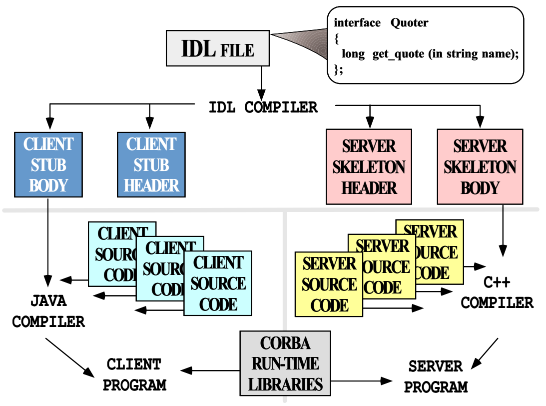

Figure 1: Components and Output of an OMG IDL Compiler

terpretive (de)marshaling.

This paper describes how TAO's IDL compiler compiler can selectively generate compiled and/or interpreted stubs and skeletons to enhance the optimization alternatives available to developers of CORBA-based embedded applications. In addition to describing performance optimizations, we describe the patterns [6] used to simplify the configuration of TAO's IDL compiler to produce stubs/skeletons using different code generation and operation demultiplexing strategies.

The remainder of the paper is organized as follows. Section 2 provides an overview of the TAO IDL compiler and outlines its interpretive and compiled (de)marshaling strategies; Section 3 describes the optimizations we employed in TAO's IDL compiler, which reduce the size of the generated interpreted stubs/skeletons without unduly degrading performance, and compares the performance of the stubs/skeletons using optimized interpretive and compiled marshaling; Section 4 describes optimizations we developed to make TAO's IDL compiler operation demultiplexing predictable and scalable; Section 5 describes the patterns used in TAO's IDL compiler design; Section 6 describes related work; and Section 7 provides concluding remarks.

2 Overview of TAO's IDL Compiler

Figure 2 illustrates the interaction between the components in the TAO's IDL Compiler. The TAO IDL compiler is a heavily modified set of extensions to the freely available SunSoft IDL compiler[1] front-end, with many portability enhancements and many defects removed.

The front-end of TAO's IDL compiler parses OMG IDL input files and generates an abstract syntax tree (AST), which is stored in main memory. The back-end of the compiler processes the

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

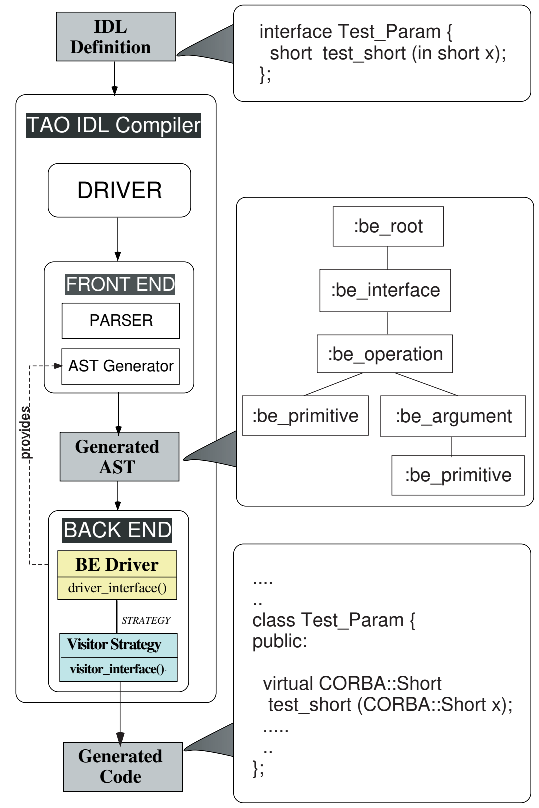

Figure 2: Interactions Between Components in TAO's IDL Compiler

AST and generates CORBA 2.2-compliant [1] C++ source code. The back-end of TAO's IDL compiler can selectively generate C++ code that is optimized for (1) TAO's Internet Inter-ORB Protocol (IIOP) protocol interpreter [5] and/or (2)

TAO's compiled stubs and skeletons.

2.1 The Design of TAO's IDL Compiler Front-end

TAO's IDL compiler front-end contains the following components:

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 OMG IDL. The front-end defines a base class called `AST_Decl` that maintains information common to all AST node types. Specialized AST node classes like `AST_Interface` inherit from this base class, as shown in Figure 3. The front-end also defines the `UTL_Scope` class, which maintains scoping information, such as the nesting level and each component of the fully scoped name. All AST nodes representing OMG IDL constructs that can define

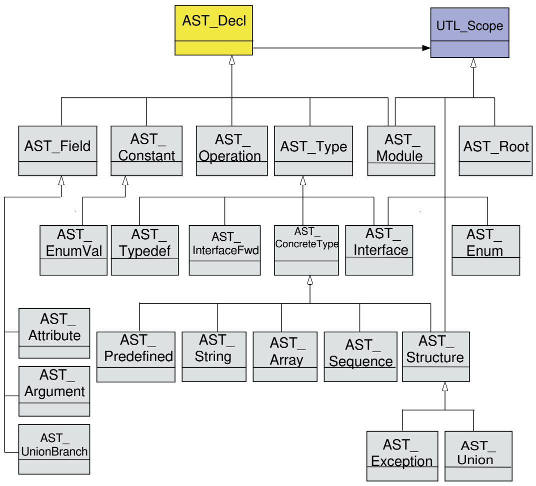

Figure 3: TAO's IDL Compiler AST Class Hierarchy

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 to create the in-memory AST and pass it to the back-end code generator.

2.2 The Design of TAO's Back-end Code Generator

The original SunSoft IDL compiler front-end only parses OMG IDL and generates a corresponding abstract syntax tree (AST). To create a complete OMG IDL compiler for TAO, we developed a back-end for the OMG IDL-to-C++ mapping. TAO's IDL compiler back-end uses several design patterns [6], such as *Abstract Factory*, *Strategy*, and *Visitor*, which are described further in Section 5 to show how these patterns were applied to simplify and enhance the design and implementation of the compiler. As a consequence of using patterns, TAO's IDL compiler back-end can be reconfigured readily to produce stubs/skeletons that use either compiled and/or interpretive (de)marshaling, as described below.

2.2.1 Supporting Interpreted Stubs and Skeletons

Algorithm overview for interpreted stubs: The interpreted stubs produced by TAO's IDL compiler use a table-driven technique that passes parameters to TAO's highly optimized IIOP protocol interpreter [5]. The basic structure of an interpretive stub is shown in Figure 4. Each step is described briefly below:

1. Initialize table entries describing each parameter's type via its `TypeCode`, and its parameter passing mode.

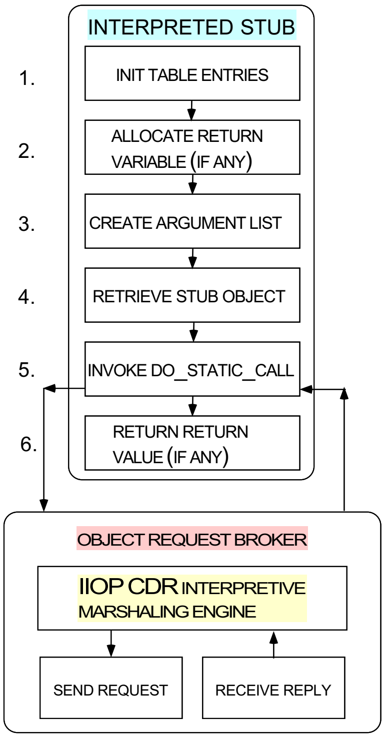

Figure 4: Interpreted Stubs Generated by TAO's IDL Compiler

2. Initialize a table describing the operation including its name, whether it is oneway 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 in which 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. This method is the stub's interface to TAO's IIOP protocol interpreter.

The table-driven technique and the `do_static_call` interface were provided in the original SunSoft IIOP implementation. Since the SunSoft IIOP implementation did not have an IDL compiler, however, each stub had to be hand-crafted, which was extremely tedious and error-prone. In contrast, the TAO IDL compiler automatically generates stubs that use interpretive (de)marshaling.

Algorithm overview for interpreted skeletons: The SunSoft IIOP skeletons use a dynamic skeleton interface (DSI) [1] approach to unmarshal parameters. An intuitive, though nonoptimal, algorithm for an interpreted skeleton is shown in Figure 5. Each step is described briefly below:

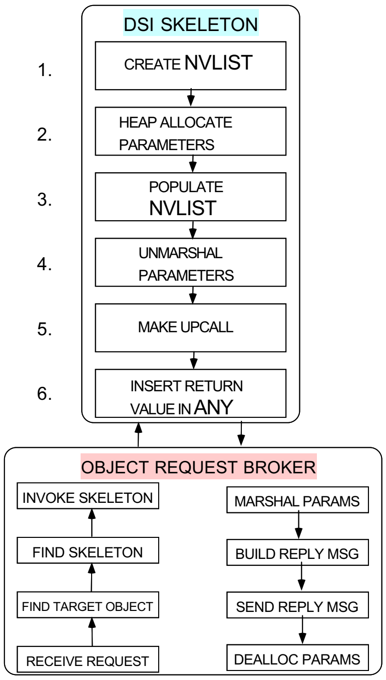

Figure 5: Unoptimized Skeletons

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 have 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.

Evaluation: In the DSI algorithm described above, memory is allocated from the heap for `inout`, `out`, and `return` values. Heap allocation is necessary because these parameters are marshaled into the outgoing IIOP Reply message after the call to the skeleton has returned. As a result, it is not possible to allocate the parameters on the run-time stack of the skeleton. The heap allocated data structures are owned by the ORB and freed using an interpretive scheme similar to the interpretive (de)marshaling scheme [5].

The original version of the skeletons used the algorithm described above, which caused excessive dynamic memory allocation and degraded performance [5] and predictability [7] significantly. Therefore, the version of TAO's IDL compiler described in this paper automatically reduces memory allocation overhead and reduces the size of the skeleton, as described in Section 3.

2.2.2 Supporting Compiled Stubs and Skeletons

Algorithm overview for compiled stubs and skeletons: The basic structure of a compiled stub is shown in Figure 6. The

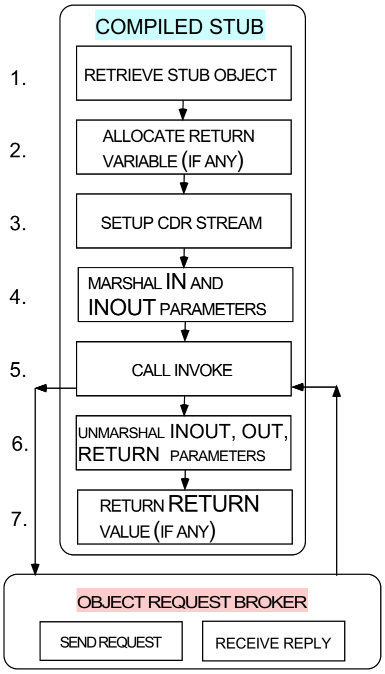

Figure 6: Compiled Stubs

compiled stub's algorithm works as follows:

1. Retrieve the stub object from the object reference.
2. Setup a Common Data Representation (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 compiled skeleton generated by TAO's IDL compiler works in much the same way as the stub. However, the incoming `in` and `inout` parameters are first demarshaled from the request, an upcall is made, and the resulting `return`, `inout`, and `out` parameters are marshaled into the reply request.

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 CDR stream. TAO's ORB Core provides these operators for primitive types; user-defined types are generated by the IDL compiler.

Evaluation: 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 marshals and unmarshals. Moreover, all unmarshaling and marshaling of the parameters occur in the scope of the skeleton. In contrast, the interpretive skeletons in the DSI scheme 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.

3 Optimizing Generated Stubs and Skeletons for Embedded Systems

As described in Section 1, IDL compilers for embedded systems must typically generate stubs and skeletons with small memory footprints. Therefore, we devised a technique to minimize the code size of the interpreted stubs/skeletons generated by TAO's IDL compiler, without unduly reducing performance. Our code-size reduction techniques for interpreted stubs/skeletons are guided by the following three optimization principles:

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

Implementing these optimizations required several new features to be added 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. This allows parameters to be allocated on the stack instead of from the heap, thereby eliminating the overhead of dynamic memory allocation and locking, which is essential for embedded applications with real-time performance requirements [7]. Below, we describe the specific techniques we applied in TAO's IDL compiler to optimize the generated stubs and skeletons.

3.1 Optimizing DSI-style Interpretive Skeletons

Motivation: As shown in Figure 5, each interpretive skeleton is required to create an `NVList` and populate it with parameters. In addition, memory is allocated dynamically from the heap, rather than on the run-time stack, for the `inout`, `out`, and `return` types. Dynamic memory management is necessary in this scheme since the marshaling of these parameters in the outgoing stream takes place *after* the call to the skeleton has returned.

Applications using the CORBA DSI must comply with the programming style described above. However, for applications using the static skeleton interface (SSI), the ORB Core can be modified to provide the necessary operations used by the IDL generated code. Applications cannot directly access these operations since they are protected.

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

Optimization: Based on this observation, we implemented a table-driven technique similar to the one used for the interpretive stubs described in Section 2.2.1. This table-driven approach defines two new interfaces in TAO's (de)marshaling engine that are modeled after 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 optimized interpretive skeleton is shown in Figure 7.

Benefits: The main benefit of this optimized 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 used by TAO's compiled skeletons, outlined in Section 2.2.2. In addition, if any outgoing types are mapped into C++ pointers, we can directly invoke the C++ `delete` operator rather than deallocating the memory interpretively.

In summary, our optimizations eliminate the need for excessive dynamic memory allocations, thereby improving performance and predictability, while minimizing the code size.

3.2 Controlling Code Generation at Finer-Granularity

Motivation: The TAO IDL compiler provides options that make it possible to select the type of (de)marshaling to be used on a per file basis. For some real-time applications it is beneficial to choose interpretive or compiled stubs and skeletons at a finer granularity than an entire IDL file. Having finer grained

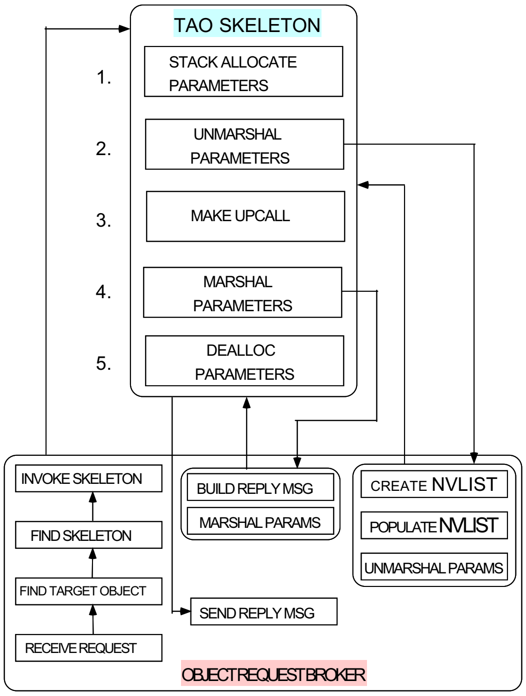

Figure 7: Optimized Interpretive Skeletons

control over the code generation strategy helps embedded applications achieve (1) high performance for time-critical operations using compiled (de)marshaling while (2) maintaining small code size for less time-critical operations that use interpretive (de)marshaling.

For example, an embedded implementation of an Event Channel [8] may require highly optimized event dispatching for `push` operations. However, administrative operations for connecting and disconnecting consumers and suppliers from the Event Channel are not in the critical path. Therefore, they can be implemented using a less efficient, but smaller, interpretive approach. Note that operations in this example cannot be spread over several IDL files because they are grouped in a single interface, *i.e.*, `CosEventComm::PushConsumer`, which is standardized in the OMG Common Object Services specification [9].

Optimization: To support fine-grained tradeoffs between time and space, we have devised a scheme where developers embed `#pragma` directives into an IDL file. TAO's IDL compiler can use these `#pragmas` to select the desired code generation strategy. For instance, in the Event Channel example above, developers could specify the IDL interface as follows:

```
module CosEventComm {
  interface PushConsumer {
    #pragma tao_compiled
    void push (in any event);
      raises(Disconnected);
    #pragma tao_interpreted
    void disconnect_push_consumer ();
    // ...
```

Benefits: The `#pragmas` serve as fine-grained instructions to TAO's IDL compiler, indicating when it should optimize for size and when to optimize for speed. This feature allows developers to override the default IDL compiler settings for each IDL file. As described in Section 5, TAO's IDL compiler back-end uses the Visitor pattern to generate the C++ code. The AST nodes representing operations carry pointers to the Visitor factory that corresponds to the code generation strategy specified by the `#pragmas` defined in each scope.

3.3 Benchmarking Compiled vs. Interpreted Stubs/Skeleton Performance

3.3.1 Benchmarking Testbed Design

The throughput benchmark described below was conducted using a Bay Networks LattisCell 10114 ATM switch connected to two dual-processor UltraSparc-2s running SunOS 5.5.1. The LattisCell 10114 is a 16 Port, OC3 155 Mbs/port switch. Each UltraSparc-2 contains two 300 MHz Super SPARC CPUs with a 1 Megabyte cache per-CPU. The SunOS 5.5.1 TCP/IP protocol stack is implemented using the `STREAMS` communication framework. Each UltraSparc-2 has 512 Mbytes of RAM and an ENI-155s-MF ATM adaptor card, which supports 155 Megabits per-sec (Mbps) SONET multimode fiber. The Maximum Transmission Unit (MTU) on the ENI ATM adaptor is 9,180 bytes. Each ENI card has 512 Kbytes of on-board memory. A maximum of 32 Kbytes is allotted per ATM circuit connection for receiving and transmitting frames (for a total of 64 K). This allows up to eight switched virtual connections per card.

In our experiment,we sent a large `sequence` of various data types from a client program to a server program. For our experiment, we sent `sequences` of `long`, `octet`, and the following simple C++ `struct`:

```
struct BinStruct
{
  short s; char c; long l; octet o; double d;
};
```

The `sequence` was not sent in a single CORBA invocation, however. Instead, the data was split into a number of fixed-sized blocks. Varying the block size illustrates how the invocation overhead is minimized as the block size increases.

3.3.2 Empirical Results

Run-time performance tradeoffs: Figure 8 illustrates the run-time performance of compiled vs. interpretive (de)marshaling. For basic data types, such as `long` and `char`, the 155 mbps ATM network bandwidth limits end-to-end throughput. In this case, the differences between the

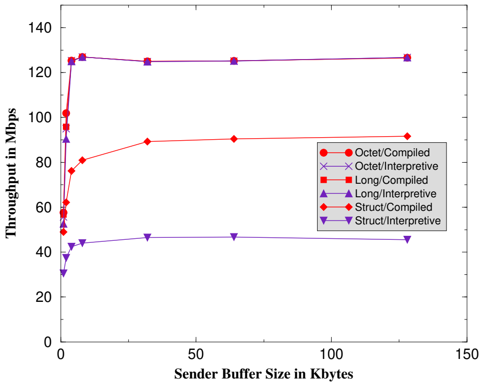

Figure 8: Throughput for TAO's Compiled vs. Interpretive (De)Marshaling (ATM)

(de)marshaling approaches are relatively minor, though there is a significant performance improvement (100%) when compiled marshaling is used for `struct`s.

Figure 9 shows the results of the same experiment over the loopback interface. In this case, the performance of compiled

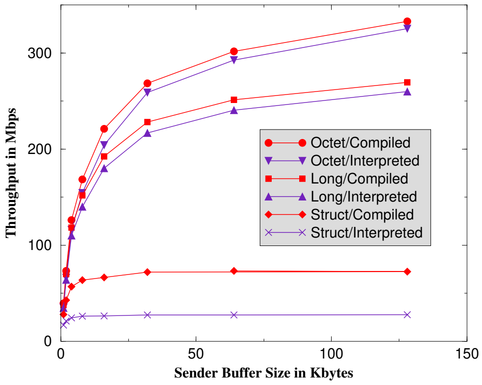

Figure 9: Throughput for TAO's Compiled vs. Interpretive (De)Marshaling (loopback)

marshaling is improved slightly for basic data types, such as `long`. This is due partly to our optimizations for the interpretive (de)marshaling case for primitive types. However, the cost of interpretive (de)marshaling increases with complexity of the data type. For instance, Figure 9 illustrates how compiled (de)marshaling outperforms interpretive (de)marshaling by over 250% (72 Mbps vs. 27 Mbps, respectively) when sending `sequence`s of `struct`s.

Compile-time size tradeoffs: Our experiment used a simple interface, with only a few operations, a simple `struct`, and `sequence`s of the types shown in the figures. Therefore, we discovered that the code size for this experiment is not affected significantly by the choice of interpreted vs. compiled code. However, the difference in size increases with the number and complexity of the data types used, *e.g.,* deeply nested data structures, the number of operations defined, and the number of parameters used by these operations.

For complex interfaces, such as TAO's implementation of the CORBA Trading Service, the time/space tradeoff between compiled and interpreted (de)marshaling becomes clear, as shown in the table below:

File Type	Compiled (De)marshaling	Interpretive (De)marshaling	% Change
Stub	675,779 bytes	625,060 bytes	8.1%
Skeleton	272,209 bytes	270,477 bytes	0.6%
Total	947,988 bytes	895,537 bytes	5.8%

We are working on optimizations that will significantly reduce the size of the interpreted stubs and skeletons for all data types and IDL interfaces, while improving the efficiency. These optimizations include a hybrid form of marshaling where the stubs and skeletons use interpreted marshaling, with additional compile-time knowledge of data types. Having compile-time knowledge will benefit (de)marshaling of deeply nested data types since this will eliminate the need to interpret their nested structure.

4 Optimizing Operation Demultiplexing in Server Skeletons

4.1 Overview

Once an ORB's Object Adapter identifies the correct *servant* for an incoming CORBA request, the next step is that the skeleton must demultiplex the request to the correct operation within the servant. This step involves locating the C++ method that matches the operation name passed with the incoming client request. Figure 10 illustrates the steps and alternative strategies supported by TAO for operation demultiplexing.

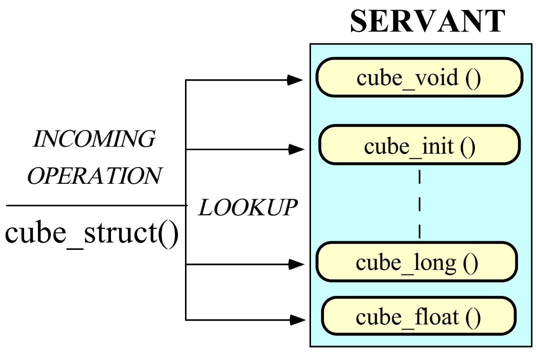

Figure 10: Operation Demultiplexing in Server Skeletons

For ORBs like TAO that target real-time embedded systems, operation demultiplexing must be efficient, scalable, and predictable. Therefore, we generate efficient operation lookup using GPERF [10], which is a freely available perfect hash function generator we developed.

GPERF automatically constructs perfect hash functions from a user-supplied list of keywords. In addition to the perfect hash functions, GPERF can generate linear and binary search strategies from a user-supplied list of keywords. Developers can use TAO IDL compiler command-line options to select between various operation lookup strategies such as linear search, binary search, dynamic hashing, and perfect hashing.

Figure 11 illustrates the interaction between the TAO IDL compiler and GPERF. When perfect hashing, linear search and binary search operation demultiplexing strategies are selected, TAO's IDL compiler invokes GPERF as a co-process to generate an optimized lookup strategy for operation names in IDL interfaces.

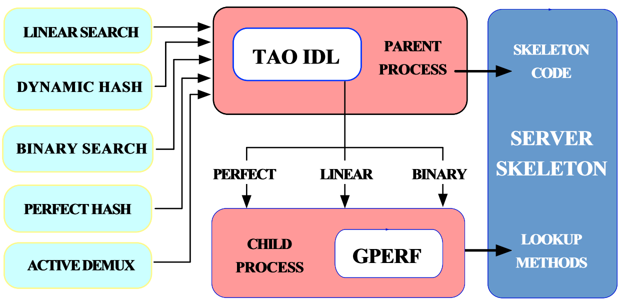

Figure 11: Integrating TAO's IDL Compiler and GPERF

4.2 Operation Demultiplexing Performance

The final step at a server-side ORB's Object Adapter involves demultiplexing a request to the appropriate skeleton, which demarshals the request and dispatches the associated operation in the servant as an upcall. To measure operation demultiplexing overhead, our experiments described below defined a range of operations, 1 through 100, in the IDL interface.

TAO's IDL compiler can be configured to generate skeletons that demultiplex operations using linear search, binary search, dynamic hashing, or perfect hashing. The lookup key for this phase is the operation name, which is a `string`. Therefore, perfect hashing is well-suited to operation demultiplexing since all operation names are known at compile time.

Figure 12 plots operation demultiplexing latency as a function of the number of operations.

This figure indicates that perfect hashing behaves predictably and efficiently, outperforming dynamic hashing. As expected, linear and binary search depend on the number of operations, thereby complicating worst-case schedulability analysis for real-time applications.

Note that all of TAO's optimized operation demultiplexing strategies described above are entirely compliant with the CORBA specification.

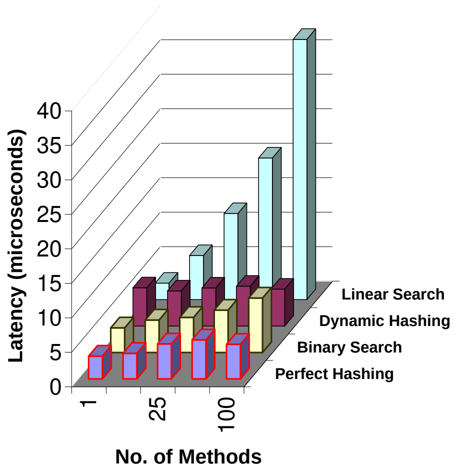

Figure 12: Operation Demultiplexing Latency with Alternative Search Techniques

5 Design Patterns Used in TAO's IDL Compiler

5.1 Motivation for Applying Patterns

As shown above, an IDL compiler is a critical component of an ORB. It is responsible for correctly mapping IDL input files into stub and skeleton classes that must be integrated with application developer code. The process of translating OMG IDL into the standard C++ mapping is a large and complex specification. Moreover, TAO's IDL compiler must be flexible to generate compiled and/or interpretive (de)marshaling code to meet the needs of various types of embedded, real-time applications. In addition, the ORB and the IDL compiler must collaborate to provide optimal performance for parameter marshaling and demarshaling, *e.g.*, optimizations in the ORB require changes to the IDL compiler so that generated code exploits the new features.

Thus, the IDL compiler must be highly maintainable, capable of being adapted to different code generation strategies. Likewise, it also must be capable of being debugged easily if the generated code deviates from the CORBA specification. Using design patterns allowed us to tackle this challenges in a systematic way, as described below.

5.2 Patterns used in TAO's IDL compiler

The SunSoft IDL compiler used as the basis for TAO's IDL compiler uses a yacc-generated parser to build an abstract syntax tree (AST) representation from IDL input files. Developers can add back-ends that generate code from the AST by providing an *Abstract Factory* [6] to the parser. Although this design simplies the task of creating multiple back-ends, and allows back-end developers to control the AST representation, it does not solve the following design challenges:

Need to know the exact type of a node: As the back-end traverses the AST to generate code, the exact type of the node being visited must be known. For example, the declaration of an input argument will change depending on the type of the argument. Basic types like `short` are passed by value, but structures are passed by reference. The SunSoft design supports this partially by using typesafe downcasting.

Dependency of the mapping on the context: The mapping for a field whose type is a `short` is different from a field whose type is an object reference. In the first case the type of the field is just `CORBA::Short`; in the second case a `T_var` (or something similar) must be used. However, the same types map to `CORBA::Short` and `T_ptr` when they appear as input arguments.

Poor scalability of virtual functions: Using virtual functions to generate the right code is an intuitive solution. Unfortunately, it does not scale well as the number of different contexts increases. For instance, the same argument type may show many times in the stub definition and many other times in the skeleton definition in different modes, *e.g.,* `in`, `inout`, `out`, and `return`. Each time a different variation is required this will increase the number of virtual functions. Moreover, the code generation logic for a particular context will spread across the compiler source, which is currently ~120,000 lines of C++ code.

Dealing with localized changes: End-users can select whether TAO's IDL compiler generates interpretive or compiled (de)marshaling on a per-file or per-operation basis. The end-user selection affects the generation of the stubs and skeleton methods, part of the `AST_Operation` mapping, and also requires the generation of insertion and extraction operators from CDR streams, *i.e.*, adding new phases to the code generation. However, most of the code to generate the stubs and skeletons *declarations* remains unchanged. Changing the AST hierarchy radically to handle this variation is not a viable solution.

We resolved the challenges outlined above by using *double dispatching* and the *Visitor* [6] pattern. These patterns are appropriate since the actions that are executed on a particular AST node depend both on the context where the actions occur and the type of the particular node.

To implement these patterns, we added methods to our back-end AST nodes so they could be traversed by visitor objects. A single visitor represents a particular context in the code generation, *e.g.*, whether to generate argument declarations or to marshal the arguments to generate a request. The visitor can consult the type of the node and generate proper code for its context and type.

Each visitor usually delegates part of its work on other visitors. For example, the compiler generates the method declarations for a skeleton class using one visitor, which delegates on another visitor to generate the argument declarations. This

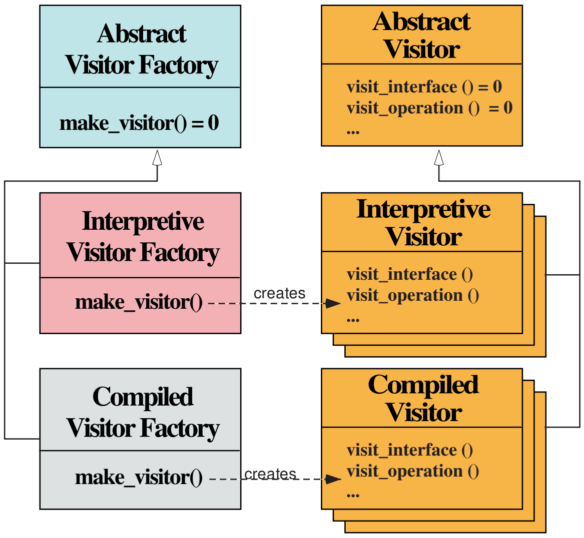

Figure 13: Creating Visitors in TAO's IDL Compiler Using Abstract Factories

decoupling between the different context or phases in the code generation and the types being processed allows us to customize a particular task without affecting other portions of the IDL compiler.

An important goal for TAO's IDL compiler was to investigate the tradeoffs between compiled vs. interpreted (de)marshaling. By using patterns, the only visitors that must be replaced to switch between compiled and interpreted schemes are those responsible for generating stub and skeleton implementations. However, the generation of IDL structures and sequences, the stubs declarations, and the skeleton declarations remain unchanged.

An *Abstract Factory* is used to localize the construction of the right visitors, as shown in Figure 13. Therefore, the visitor are used as *Strategies* [6] for different code generation algorithms.

6 Related Work

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

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

encoding [12].

According to the SunSoft IIOP developers, interpretive marshaling is preferable since it decreases code size and increases the likelihood of remaining in the processor cache. Our goal is to generate efficient stubs and skeletons by extending optimizations provided in USC [13] and Flick [4], 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++. TAO's IDL compiler implements optimizations to improve the performance of its interpretive stubs, thereby providing fine-grained tradeoffs between time and space. In contrast, the stubs and skeletons produced by USC and Flick are compiled and they optimize for time, not space.

Demultiplexing optimizations: Demultiplexing routes messages between different levels of functionality in layered communication protocol stacks. Most conventional communication models (such as the Internet model or the ISO/OSI reference model) require some form of multiplexing to support interoperability with existing operating systems and protocol stacks. In addition, conventional CORBA ORBs utilize several extra levels of demultiplexing at the application layer to associate incoming client requests with the appropriate servant and operation. Layered multiplexing and demultiplexing is generally disparaged for high-performance communication systems [14] due to the additional overhead incurred at each layer. [15] describes a fast and flexible message demultiplexing strategy based on dynamic code generation. Our earlier work [16] evaluates the performance of alternative demultiplexing strategies for real-time CORBA that were hand-crafted into the skeletons. In this paper, we describe how we enhanced the IDL compiler to automatically produce these strategies in the skeletons.

Our framework uses a delayered demultiplexing architecture to select optimal demultiplexing strategies based on compile-time and run-time analysis of CORBA IDL interfaces.

7 Concluding Remarks

The distributed embedded systems industry is poised to undergo a revolution with the emergence of operating systems, such as Inferno, Windows CE, EPOC, and Palm OS. In addition, developers of embedded systems are increasingly migrating towards the use of off-the-shelve middleware components to lower software lifecycle costs and decrease time-to-market. In this context, the flexibility and adaptability offered by CORBA makes it an attractive middleware architecture. 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.

The TAO IDL compiler described in this paper is designed to address the needs of embedded applications. In particular, it allows compiled and/or interpreted (de)marshaling to be selected on a per-file or per-operation basis. In addition, it supports optimal operation demultiplexing based on perfect hashing. This flexibility allows application developers the flexibility to selectively tradeoff time and space, which is crucial for distributed, real-time, and embedded systems. The IDL compiler optimizations and performance results presented in this paper illustrate how the next-generation of CORBA ORBs will be well-suited for distributed embedded systems that require efficient, scalable, and predictable performance.

References

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

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

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

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

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

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

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

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

[9] Object Management Group, *CORBAServices: Common Object Services Specification, Updated Edition*, 95-3-31 ed., Dec. 1998.

[10] D. C. Schmidt, "GPERF: A Perfect Hash Function Generator," in *Proceedings of the 2^{nd} C++ Conference*, (San Francisco, California), pp. 87–102, USENIX, April 1990.

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

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

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

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

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

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