Principles for Optimizing CORBA
Internet Inter-ORB Protocol Performance

Aniruddha Gokhale and Douglas C. Schmidt

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

Abstract

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

This paper provides three contributions to the study and design of efficient CORBA IIOP implementations. First, we outline the software architecture of the SunSoft’s public domain implementation of IIOP version 1.0. Second, we pinpoint the key sources of overhead in the SunSoft IIOP implementation (which is an implementation of IIOP written in C++) by measuring its performance for transferring richly-typed data over a high-speed ATM network. Third, we empirically demonstrate the benefits of systematically applying protocol optimizations to SunSoft IIOP.

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

Keywords: Distributed object computing, CORBA, IIOP performance, communication middleware protocol optimizations, high-speed networks.

1 Motivation

An increasingly important class of distributed applications require high bandwidth and low latency. These applications include telecommunication systems (e.g., call processing and switching), avionics control systems (e.g., operational flight programs for fighter aircraft), multimedia (e.g., video-on-demand and teleconferencing), and simulations (e.g., battle readiness planning). In addition to requiring good performance, these applications must be flexible and re usable.

The Common Object Request Broker Architecture (CORBA) is a distributed object computing middleware standard defined by the Object Management Group (OMG) [13]. CORBA is intended to support the production of flexible and reusable distributed services and applications. Many implementations of CORBA are now available.

The CORBA 2.0 specification requires Object Request Brokers (ORBs) to support a standard interoperability protocol. IIOP is the standard for interoperability for distributed object computing over TCP/IP.²

[7, 8, 9] show that the performance of conventional CORBA middleware implementations is relatively poor compared to lower-level implementations using C/C++ since the ORBs incur a significant amount of data copying, marshaling, demarshaling, and demultiplexing overhead. These results, however, focused entirely on the communication performance between homogeneous ORBs. They do not measure the run-time costs of interoperability between heterogeneous ORBs. In addition, earlier work on measuring CORBA performance did not present the results of optimizations to reduce key sources of ORB overhead.

In this paper, we measure the performance of an implementation of IIOP, freely available from SunSoft, using a CORBA/ATM testbed environment similar to [7, 8, 9]. We measure the performance of SunSoft IIOP and precisely pinpoint its performance overheads. In addition, we describe the results of systematically applying seven principle-driven optimizations [16] that substantially improve the performance of SunSoft IIOP. These optimizations include: optimizing for the common case; eliminating gratuitous waste; replacing general purpose methods with specialized, efficient

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

²Netscape 4.0 integrates IIOP into its Web browser, making IIOP the most widely available protocol for interoperability between heterogeneous ORBs.
The results of applying these optimization principles to SunSoft IIOP improved its performance 1.9 times for doubles, 3.3 times for longs, 4 times for shorts, 5 times for chars/octets, and 6.7 times for richly-typed structs over ATM networks. Our optimized version of SunSoft IIOP is now comparable to existing commercial ORBs [7, 8, 9] using the static invocation interface (SII) and around 2 to 4.5 times (depending on the data type) faster than commercial ORBs using the dynamic skeleton interface (DSI).

This paper is organized as follows. Section 2 outlines the SunSoft IIOP reference implementation; Section 3 presents the results of our performance optimizations of SunSoft IIOP over a high-speed ATM network; Section 4 compares our research with related work; and Section 5 provides concluding remarks. The Appendices describe CORBA IIOP in greater depth and present a detailed view of SunSoft IIOP’s protocol engine structure and run-time functionality.

2 Overview of SunSoft IIOP

2.1 CORBA Features Supported by SunSoft IIOP

SunSoft IIOP is a freely available implementation of an ORB Core that comprises an implementation of the IIOP version 1.0 protocol. SunSoft IIOP is written in C++ and provides many features of a CORBA 2.0 ORB, except for an IDL compiler, an interface repository, and a complete Basic Object Adapter (BOA).

On the client-side, SunSoft IIOP provides a static invocation interface (SII) and a dynamic invocation interface (DII). The SII is used by the client-side stubs. Since SunSoft IIOP does not provide an IDL compiler, client stubs using the SII API must be generated manually.

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

2.2 Overview of SunSoft IIOP Components

The components in SunSoft IIOP are shown within the ORB Core in Figure 1. The TypeCode marshaling/demarshaling interpreter is the primary component of the SunSoft IIOP implementation. SunSoft IIOP’s interpretive marshaling scheme provides a mechanism that encodes or decodes parameter data by identifying their TypeCodes at run-time. In

SunSoft IIOP, this is done using the .kind field of the SunSoft’s TypeCode object. The TypeCode pseudo-object maintains an internal representation of the OMG IDL type information for a data type as it is marshaled and transmitted over a network.

The motivation for using an interpreter is to reduce the space utilization of the marshaling/demarshaling engine. This is important for embedded systems that cannot afford large memory footprint. SunSoft IIOP requires less than 100 Kbytes on a real-time operating system like VxWorks. A small memory footprint is also useful for general-purpose computing systems since it allows the interpreter to fit entirely within a processor cache.

Each component of the SunSoft IIOP architecture is described below:

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

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

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

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

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

- **The deep_free method** – The deep_free method is used by the DSI server to free allocated memory after incoming data has been demarshaled and passed to a server application.

**The utility methods:** SunSoft IIOP provides several methods that perform various utility tasks, including:

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

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

### 3 Experimental Results of CORBA IIOP over ATM

#### 3.1 CORBA/ATM Testbed Environment

**3.1.1 Hardware and Software Platforms**

The experiments in this section were conducted using a FORE systems ASX-1000 ATM switch connected to two dual-processor UltraSPARC-2s running SunOS 5.5.1. The ASX-1000 is a 96 Port, OC12 622 Mbs/port switch. Each UltraSparc-2 contains two 168 MHz Super SPARC CPUs with a 1 Megabyte cache per-CPU. The SunOS 5.5.1 TCP/IP protocol stack is implemented using the STREAMS communication framework. Each UltraSparc-2 has 256 Mbytes of RAM and an ENI-155s-MF ATM adaptor card, which supports 155 Megabits per-sec (Mbps) SONET multimode fiber. The Maximum Transmission Unit (MTU) on the ENI ATM adaptor is 9,180 bytes. Each ENI card has 512 Kbytes of on-board memory. A maximum of 32 Kbytes is allotted per ATM virtual circuit connection for receiving and transmitting frames (for a total of 64 K). This allows up to eight switched virtual connections per card.

**3.1.2 Traffic Generator for Throughput Measurements**

Traffic for the experiments was generated and consumed by an extended version of the widely available ttcp [15] protocol benchmarking tool. We extended ttcp for use with SunSoft IIOP. We hand-crafted the stubs and skeletons for the different methods defined in the interface. Our hand-crafted client-side stubs use the SII API provided by SunSoft IIOP. On the server-side, the Object Adaptor uses a callback method supplied by the ttcp server application to dispatch incoming requests and their parameters to the target object.

Our ttcp tool measures end-to-end data transfer throughput in Mbps from a transmitter process to a remote receiver process across an ATM network. The flow of user data for each version of ttcp is uni-directional, with the transmitter flooding the receiver with a user-specified number of data buffers. Various sender and receiver parameters may be selected at run-time. These parameters include the number of data buffers transmitted, the size of data buffers, and the type of data in the buffers. In all our experiments the underlying socket queue sizes were enlarged to 64 Kbytes (which is the maximum supported on SunOS 5.5.1).

The following data types were used for all the tests: primitive types (short, char, long, octet, double) and a C++ struct composed of all the primitives (BinStruct). The size of the BinStruct is 32 bytes. SunSoft IIOP transferred the data types using IDL sequences, which are dynamically-sized arrays. The IDL declaration is shown in the Appendix A. The sender-side transmitted data buffer sizes of a specific data type incremented in powers of two, ranging from 1 Kbytes to 128 Kbytes. These buffers were repeatedly sent until a total of 64 Mbytes of data was transmitted.

#### 3.1.3 Profiling Tools

The profile information for the empirical analysis was obtained using the Quantify [10] 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.

#### 3.2 Performance Results and Benefits of Optimization Principles

**3.2.1 Methodology**

CORBA implementations like SunSoft IIOP are representative of complex communication software. Optimizing such software is hard, particularly since seemingly minor “mistakes” (such as copying data excessively) can significantly reduce performance [7]. Therefore, developing high performance ORBs requires an iterative, multi-step process. The first step involves measuring the performance of the system and pinpointing the sources of overhead. The second step involves a

---

4 We are developing an IDL compiler for TAO that generates stubs and skeletons automatically.
This section describes the optimizations we applied to SunSoft IIOP to improve its throughput performance over ATM networks. First, we show the performance of the original SunSoft IIOP for the IDL data types defined in Appendix A. Next, we use Quantify to illustrate the key sources of overhead in SunSoft IIOP. Finally, we describe the benefits of the optimization principles we applied systematically to improve the performance of SunSoft IIOP.

The optimizations described in this section are based on fundamental principles for implementing protocols efficiently. [16] describes a collection of optimization principles in detail and illustrates how they have been applied in existing protocol implementations, e.g., TCP/IP. We focus on the principles in Table I that we used to improve IIOP performance. We chose these principles since we found them applicable to improving the SunSoft IIOP performance.

When describing our optimizations, we refer to these principles and explain how their use is justified. Thus, we describe a measurement-based, principle-driven methodology to improve the performance of SunSoft IIOP.

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

1. Aggressive inlining to optimize for the common case – which is discussed in Section 3.2.3;
2. Precomputing, adding redundant state, passing information through layers, eliminating gratuitous waste, and specializing generic methods – which is discussed in Section 3.2.4;
3. Optimizing for the processor cache – which is discussed in Section 3.2.5.

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

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

### Table I: Summary of Principles for Efficient Protocol Implementations

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

When describing our optimizations, we refer to these principles and explain how their use is justified. Thus, we describe a measurement-based, principle-driven methodology to improve the performance of SunSoft IIOP.

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

1. Aggressive inlining to optimize for the common case – which is discussed in Section 3.2.3;
2. Precomputing, adding redundant state, passing information through layers, eliminating gratuitous waste, and specializing generic methods – which is discussed in Section 3.2.4;
3. Optimizing for the processor cache – which is discussed in Section 3.2.5.

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

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

3.2.2 Performance of the Original IIOP Implementation

**Sender-side performance:** Figure 2 illustrates the sender-side throughput obtained by sending 64 Mbytes of various data types for buffer sizes ranging from 1 Kbytes to 128 Kbytes (incremented in powers of two) and compares them with a hand-optimized baseline implementation that uses TCP/IP and sockets. These results indicate that different data types achieved substantially different levels of throughput. The highest ORB throughput results from sending doubles, whereas BinStructs displayed the worst behavior. This variation in behavior stems from the marshaling and demar-
shaling overhead for different data types, as well as from the use of the interpretive marshaling/demarshaling engine in SunSoft IIOP, which incurs a large number of recursive method calls.

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

**Receiver-side performance:** The Quantify analysis for the receiver-side are shown in Figures 3 and 4. The receiver-side results for sending primitive data types indicate that most of the run-time costs are incurred by the following methods:

1. The TypeCode interpreter – i.e., the traverse method on the TypeCode object;
2. The CDR methods that retrieve the value from the incoming data – e.g., get_long and get_short;
3. The deep_free method – which deallocates memory;
4. The CDR::decoder method – The receiver spends a significant amount of time traversing the BinStruct TypeCode (struct_traverse) and calculating the size and alignment of each member in the struct.

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

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

Figures 3 and 4 illustrate the receiver is the principal performance bottleneck. Therefore, our initial set of optimizations are designed to improve receiver performance. Likewise, since the receiver is the bottleneck, we show the Quantify profile measurements only for it.

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

- **Problem:** high invocation overhead for small, frequently called methods: This subsection describes an optimization to improve the performance of IIOP receivers. We applied Principle 1 from Table 1, which optimizes for the common case. Figures 3 and 4 illustrate that the appropriate get method of the CDR class must be invoked to retrieve the data from the incoming stream into a local copy. For instance, depending on the data type, methods like CDR::get_long or CDR::get_longlong are called millions of times to decode the 64 Mbytes of data. Since these get methods are invoked so frequently they are prime targets for our first optimization step.

- **Solution:** inline method calls: Our solution to reduce invocation overhead for small, frequently called methods was to inline these methods, using C++ language features such as preprocessor macros or the inline keyword.

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

- **Solution:** replace inline methods with preprocessor macros: To ensure inlining for all small, frequently called methods, we employ a more aggressive inlining strategy. Methods like ptr_align_binary (which is used to align a pointer variable at the specified byte alignment), which became uninlined after inlining the methods that call them, are now forcibly inlined by defining them as preprocessor macros instead of as C++ inline methods.

**Optimization results:** Figure 5 illustrates the effect of aggressive inlining on the throughput of doubles and structs after applying the first optimization (inlining).
BinStructs. After aggressive inlining, the new throughput results indicate only a marginal (i.e., 4%) increase in performance. Figures 6 and 7 show profiling measurements for the receiver. The analysis of overhead for the sender-side reveals that, as before, most run-time overhead stems from write calls to the network.

The receiver-side Quantify profile output reveals that aggressive inlining does force operations to be inlined. However, this inlining increases the code size for other methods (such as struct_traverse, CDR::decoder, and calc_nested_size_and_alignment), thereby increasing their run-time costs.

Certain SunSoft IIOP methods such as CDR::decoder and TypeCode::traverse are large and general-purpose. Inlining the small methods described above causes further “code bloat” for these methods. Thus, when they call each other recursively a large number of times, very high method call overhead results.

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

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

- **Problem: too many method calls:** The aggressive inlining optimization in Section 3.2.3 did not cause substantial improvement in performance due to processor cache effects. Figure 7 reveals that for sending structs, the high cost methods are calc_nested_size_and_alignment, CDR::decoder, and struct_traverse. These methods are invoked a substantial number of times (29,367,801, 33,554,437, and 4,194,303 times, respectively) to process incoming requests.

  To see why these methods were invoked so frequently, we analyzed the calls to struct_traverse. The TypeCode interpreter invoked struct_traverse 2,097,152 times for data transmissions of 64 Mbytes in sequences of 32 byte BinStructs. In addition, the TypeCode interpreter calculated the size of BinStruct (using the calc_nested_size_and_alignment function), which called struct_traverse internally for every BinStruct. This accounted for an additional 2,097,152 calls.

  Several solutions to remedy this problem are outlined below:

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

  To eliminate this gratuitous waste, we can precompute (Principle 4) the size and alignment requirements of each member and store them using additional state (Principle 5) in order to speed up expensive operations. We store this additional state as private data members of the SunSoft’s TypeCode class. Thus, the TypeCode for BinStruct will calculate the size and alignment once and store these in the private data members. Every time the interpreter wants to traverse BinStruct, it uses the TypeCode for BinStruct that has already precomputed its size and alignment.

  **Solution 2: convert generic methods into special-purpose, efficient ones:** To further reduce method call overhead, and to decrease the potential for processor cache  

\[\]
misses, we moved the \texttt{struct.traverse} logic for handling \texttt{structs} into the \texttt{traverse} method. In addition, we introduced the \texttt{encoder}, \texttt{decoder}, \texttt{deep_copy}, and \texttt{deep_free} logic into the \texttt{traverse} method. This optimization illustrates an application of Principle 3 (\textit{Convert generic methods into special-purpose, efficient ones}).

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

\begin{itemize}
\item **Problem: expensive no-ops for memory deallocation:** Figure 6 reveals that the overhead of the \texttt{deep_free} method remains significant for primitive data types. This method is similar to the \texttt{decoder} method that traverses the \texttt{TypeCode} and deallocates dynamic memory. For instance, the \texttt{deep_free} method has the same type signature as the \texttt{decoder} method. Therefore, it can use the recursive \texttt{traverse} method to navigate the data structure and deallocate memory.

Close scrutiny of the \texttt{deep_free} method indicates that memory must be freed for constructed data structures (such as IDL sequences and \texttt{structs}). In contrast, for sequences of primitive types, the \texttt{deep_free} method simply deallocates the buffer containing the sequence.

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

\item **Solution: eliminate gratuitous waste:** To optimize this, we changed the deletion strategy for sequences so that the element’s \texttt{TypeCode} is checked first. If it is a primitive type, the traversal is not done and memory is deallocated directly.
\end{itemize}

**Optimization results:** Figure 8 illustrates the benefits of the optimizations from step 2 by comparing the throughput obtained for doubles and BinStructs, respectively, with those from the previous optimization steps. These results indicate that the second optimization step improves the performance of the original IIOP implementation substantially. The performance of doubles increased by a factor of 1.9, longs by a factor of 3.3, shorts by a factor of 4, chars and octets by a factor of 5, and BinStructs by a factor of 6.7. The maximum throughput of 115 Mbps obtained for sending doubles is comparable to the baseline TCP/IP performance.

Figures 9 and 9 depict the profiling measurements for the receiver (the sender continues to spend most of its execution time performing network \texttt{write} calls). The receiver methods accounting for most execution time for doubles include \texttt{traverse}, \texttt{decoder}, and \texttt{deep_free}. For BinStructs, the run-time costs of the \texttt{traverse} method in the receiver increases significantly compared to the previous optimization steps. This is due primarily to the inclusion of the \texttt{struct.traverse}, \texttt{encoder}, and \texttt{decoder} logic. Although, the run-time costs of the interpreter increased, the overall performance improved since the number of calls to functions other than itself decreased. As a result, this improved processor cache affinity, which yielded better performance. In addition, due to precomputation, \texttt{calc_nested.size_and_alignment} method need not be called repeatedly.

\subsection{3.2.5 Optimization Steps 3 and 4: Optimizations for Processor Caches}

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

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

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

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

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

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

  - **Using “outlining”** [12] to optimize for the frequently executed case – Outlining is used to remove gaps that are introduced in the processor cache as a result of branch instructions arising out of error handling code. Processor cache gaps are undesirable because they waste memory bandwidth and introduce useless instructions in the cache.

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

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

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

    Figure 11 illustrates the benefits of the optimizations for the cache by comparing the throughput obtained for doubles and BinStructs, respectively, with those from the previous optimization steps.
Related work based on optimization principles: [4] describes a technique called header prediction that predicts the message header of incoming TCP packets. This technique is based on the observation that many members in the header remaining constant between consecutive packets. This observation led to the creation of a template for the expected packet header. The optimizations reported in [4] are based on Principle 1, which optimizes for the common case and Principle 3, which is precompute, if possible. We present the results of applying these principles to optimize IIOP in Sections 3.2.3, 3.2.4, and 3.2.5.

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

Packet filters [11, 2, 6] are a classic example of Principle 6, which recommends passing information between layers. A packet filter demultiplexes incoming packets to the appropriate target application(s). Rather than having demultiplexing occur at every layer, each protocol layer passes certain information to the packet filter, which allows it to identify which packets are destined for which protocol layer. We applied Principle 6 for IIOP in Section 3.2.4 where we passed the TypeCode information and size of the element type of a sequence to the TypeCode interpreter. Therefore, the interpreter need not calculate the same quantities repeatedly.

[12] describes a scheme called “outlining” that when used improves processor cache effectiveness, thereby improving performance. We describe optimizations for processor cache in Section 3.2.5.

Related work on CORBA performance measurements: [7, 8, 9] 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. [7] measures the performance of the static invocation interface. [8] measures the performance of the dynamic invocation interface and the dynamic skeleton interface. [9] measures performance of CORBA implementations in terms of latency and support for very large number of objects.

However, the results of earlier CORBA benchmarking experiments were restricted to measuring the performance of communication between homogeneous ORBs. These tests do not measure the run-time costs of interoperability between ORBs from different vendors. In addition, these papers do not provide solutions to reduce these overheads. In contrast, we have provided solutions that significantly improve performance by reducing marshaling/demarshaling overhead.

4 Related Work

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

Related work on CORBA performance measurements: [7, 8, 9] 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. [7] measures the performance of the static invocation interface. [8] measures the performance of the dynamic invocation interface and the dynamic skeleton interface. [9] measures performance of CORBA implementations in terms of latency and support for very large number of objects.

However, the results of earlier CORBA benchmarking experiments were restricted to measuring the performance of communication between homogeneous ORBs. These tests do not measure the run-time costs of interoperability between ORBs from different vendors. In addition, these papers do not provide solutions to reduce these overheads. In contrast, we have provided solutions that significantly improve performance by reducing marshaling/demarshaling overhead.

5 Concluding Remarks

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

The results of applying these optimization principles to SunSoft IIOP improved its performance 1.9 times for doubles, 3.3 times for longs, 4 times for shorts, 5 times for chars/octet, and 6.7 times for richly-typed structs over ATM networks. Our optimized implementation is now competitive with existing commercial ORBs [7, 9] using the static invocation interface (SII) and 2 to 4.5 times (depending on the data type) faster than commercial ORBs using the dynamic skeleton interface (DSI) [8]. The results of our optimizations provide sufficient proof that performance of complex distributed systems software can be improved by a systematic application of principle-driven optimizations.

We are currently integrating the optimized SunSoft IIOP implementation with a complete real-time ORB called TAO [14]. A prototype of TAO is available at www.cs.wustl.edu/~schmidt/ACE_wrappers/TAO.

References


A TTCP IDL Description and Typedefs Code Layout

The following CORBA IDL interface was used in our experiments to measure the throughput of SunSoft IIOP described in Section 3.2.

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

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

    oneway void sendStructSeq (in StructSeq ts);
    // similarly for rest of the types
};
```