Compiler Optimization Techniques for Improving Distributed Object Computing Framework Performance over High-speed ATM Networks

Aniruddha Gokhale (gokhale@cs.wustl.edu)
Douglas C. Schmidt[1] (schmidt@cs.wustl.edu)
Ron Cytron (cytron@cs.wustl.edu)
George Varghese (varghese@cs.wustl.edu)
Department of Computer Science
Washington University
St. Louis, MO 63130

An earlier version of this paper has been submitted to the Workshop on Compiler Support for System Software, Tucson, Arizona, February 23-24, 1996.

Abstract

This paper demonstrates why the performance of flexible Distributed Object Computing (DOC) frameworks, such as the Common Object Request Broker Architecture (CORBA), must be improved to satisfy the requirements of next-generation bandwidth-intensive and delay-sensitive applications. In addition, the paper outlines a software toolkit we are building to integrate various compiler optimization techniques to improve the performance of implementations of DOC frameworks .

To illustrate the overhead imposed by conventional DOC implementations, we present performance results from using CORBA to transfer richly-typed data between hosts on a high-speed ATM network. We compare these results with those obtained using lower-level socket-based C interfaces and C++ wrappers for sockets. Our results indicate that both the C and C++ wrapper implementations outperform the CORBA implementation significantly. We analyze the CORBA performance and pinpoint specific areas in which compiler optimizations are desired. We describe how we are solving performance problems of CORBA using compiler techniques to automate *adaptive* object request demultiplexing, *Integrated Layer Processing* to reduce data copying and *Flow Analysis* and *Shapes Choice* techniques to produce efficient marshalling/demarshalling code.

Keywords: Distributed Object Computing, Common Object Request Broker Architecture, Compiler Optimizations, Frameworks, High-speed networks.

1 Introduction and Motivation

Despite dramatic increases in the performance of networks and computers, designing and implementing flexible and efficient communication software remains hard. Substantial time and effort has traditionally been required to develop this type of software. Moreover, all too frequently communication software fails to achieve its performance and functionality requirements.

Distributed Object Computing (DOC) frameworks such as the Common Object Request Broker Architecture (CORBA) [12] are a promising approach for improving the flexibility of communication software. CORBA is designed to enhance application extensibility and portability by automating common networking tasks such as parameter marshalling, object location and object activation. However, empirical studies [23, 22] show that current CORBA implementations incur significant overhead when used to implement performance-sensitive applications over high-speed networks.

As high-speed networks are increasingly deployed, the performance overhead of higher-level tools like CORBA may encourage developers to continue to use lower-level tools (like sockets). Using low-level tools in performance-sensitive, mission/life-critical applications (such as electronic medical imaging) increases the development efforts and reduces system reliability and flexibility. Therefore, it is imperative that performance of high-level, but inefficient, DOC frameworks be improved to match that of low-level, but efficient, tools.

Earlier studies [23, 22], and our results shown below, demonstrate that the factors responsible for the poor performance of CORBA implementations over ATM include excessive data copying, inefficient presentation layer conversions, and inefficient demultiplexing of object requests. Compiler technology has matured to the point where tools can be built to incorporate sophisticated optimizations automatically in order to improve the performance of DOC frameworks without requiring tedious and error-prone manual tuning. For example, advanced code flow analysis techniques can be used to achieve Integrated Layer Processing [5] to reduce the number of operations that manipulate data.

Existing implementations of network protocols are optimized for high performance. But these optimizations have been manually hand-crafted into the code. Such code is hard to comprehend, debug, maintain, and extend. In contrast, efficient and verifiable compiler optimization algorithms exist that can produce optimized code that is reliable and often better than or competitive with the hand-crafted code. For

[1] contact author

example, it is quite possible that hand-crafted optimizations overlook subtle target machine details such as register allocation or filling delay or branch slots. These low-level details are best left to the compilers, which is precisely why we are employing advanced compiler techniques to improve performance of DOC frameworks.

In this paper Section 2 demonstrates the key sources of overhead in conventional CORBA implementations over ATM; Section 3 describe how our current research is applying compiler optimizations to alleviate CORBA overhead on ATM network; Section 4 describes related work; and Section 5 presents concluding remarks.

2 Experimental Results of CORBA over ATM

2.1 CORBA/ATM Testbed

This section describes our experimental testbed features and provides results of an experiment. The testbed hardware uses a Bay Networks LattisCell 10114 ATM switch connected to two uni-processor SPARCstation 20 Model 50s. The LattisCell 10114 is a 16 Port, OC3 155Mbs/port switch. The SPARCstations contain 100 MIP Super SPARC CPUs running SunOS 5.4. The SunOS 5.4 TCP/IP protocol stack is implemented using the STREAMS communication framework [21]. Each SPARCstation 20 has 64 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 64K). This allows up to 8 connections per card.

2.2 Traffic Generators

[23] describes results of experiments performed using this testbed. These experiments tested the performance of transferring 64MB of untyped bytestream data between hosts using a "flooding model." This type of traffic is representative of performance-sensitive applications such as high-speed electronic medical imaging [10]. The tests were conducted using an enhanced version of the TTCP [27] protocol benchmarking tool. Implementations of TTCP included a low-level C socket version, a C++ wrappers version, and two implementations of CORBA: Orbix and ORBeline versions.

The results presented in this paper extend our earlier results. The new experiments further enhanced TTCP to support 64MB transfers of "richly typed" data between remote hosts over a high-speed ATM network. This type of traffic is representative of applications such as transferring the contents of a large database of patient medical records to support remote teleradiology in a large-scale distributed health care delivery system [2]

Three implementations of TTCP were measured in our latest tests: a low-level C socket version, a C++ wrapper version, and an Orbix version of CORBA.[2]. The following data types were tested in the `sequences`: the scalars (`short, long, float, double, char`) and a `struct` composed of all five scalars. The CORBA implementation transferred the data types using IDL `sequences`, which are essentially dynamically-sized arrays. The tests illustrated the overhead of marshalling and demarshalling all the types of data supported by CORBA.

Two parameters were varied for each data type:

- Receiver socket queue sizes used were 8K and 64K bytes.[3]
- Sender buffers were incremented by multiple of two, starting at 1K bytes upto a maximum of 128K bytes.

2.3 Performance Results

Figures 4 and 5 depict the time spent by the senders and receivers of the three versions of TTCP for a 128K sender buffer, 64K receiver socket buffer and sequence of *structs*.

The figures indicated that for transferring CORBA **structs**, the Orbix version spent a significant amount of time marshalling and demarshalling in the presentation layer. Note, for instance, the high level of overhead for the encoding and decoding methods and the various CORBA::Request::operator methods.

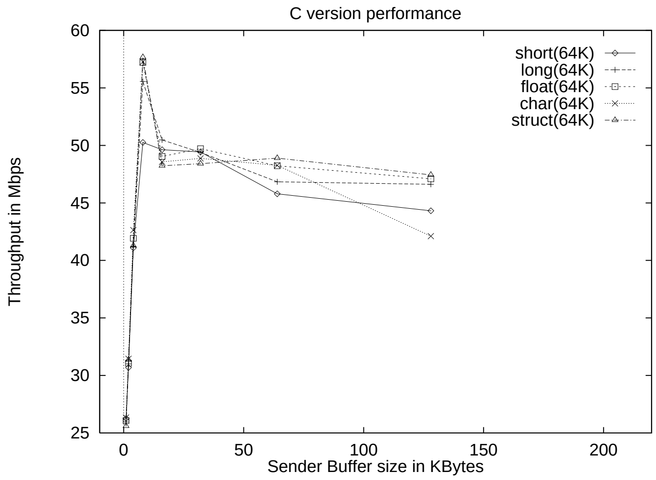

Figure 1: C version Performance

Figures 1, 2 and 3 depict performance of the three versions of TTCP for 64KB receiver socket queue size.

A comparison of the results for richly-typed data with those obtained for the untyped data presented in [23] reveal that the

[2]The ORBeline CORBA implementation consistently performed worse than Orbix and was therefore omitted from the test results presented in this paper.

[3]Performance with the 8K socket queues was consistently one-half to two-thirds slower than using the 64K queues. Therefore, we omitted the 8K results from the figures below.

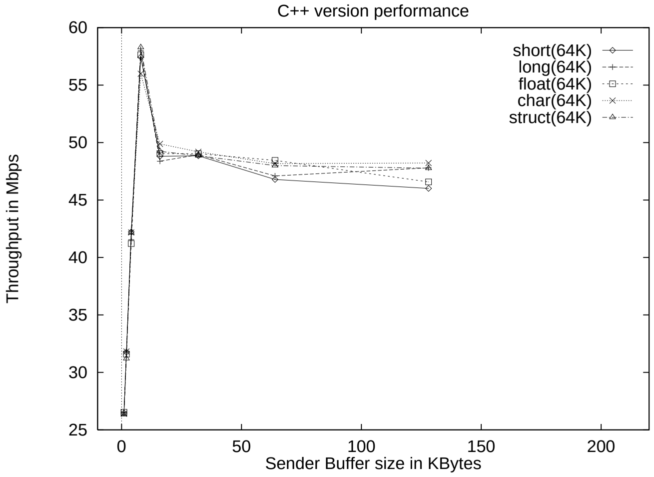

Figure 2: C++ wrappers version Performance

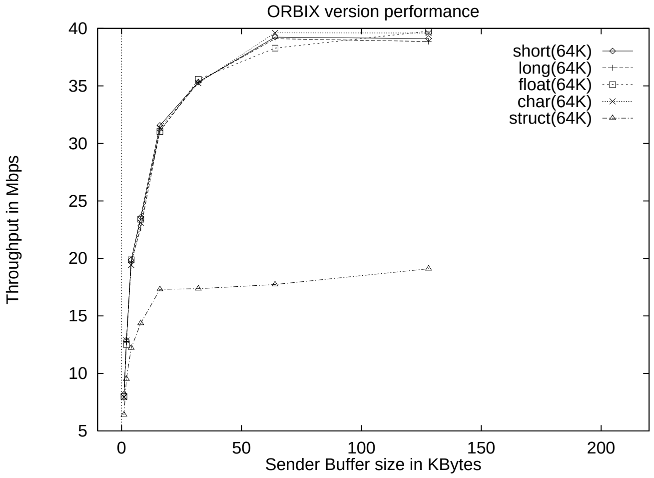

Figure 3: Orbix version Performance

low-level C socket version and the C++ socket wrapper version perform almost the same for a given socket queue size. Likewise, the performance of Orbix for sequences of scalar data types is almost the same as that reported for untyped data sequences in [23]. However, the performance of transferring sequences of CORBA *structs* for 64K and 8K was much worse than those for the scalars. As shown in the figures, this overhead is due to the significant amount spent by Orbix in marshalling/demarshalling the structs.

In all the experiments, the CORBA implementation performed consistently worse than the C and C++ wrapper versions. On average, the performance was around two-thirds the level of the C and C++ versions. As shown in the figures, this difference is due to presentation layer conversion overhead, data copying, and inefficient buffer management.

The experimental results and their analysis have provided sufficient insight into the specific areas causing performance degradation for the CORBA implementations. Thus it is

Program	%Time	Method Name
C sender	98.10	write
	0.31	close
	0.31	open
C++ sender	97.59	write
	0.80	_lrw_unlock
	0.37	open
Orbix sender	84.54	write
	2.68	CHECK(unsigned long, cursor)
	1.34	CORBA::Request::operator <<(const long&)
	1.34	CORBA::Request::operator <<(const double&)
	1.34	CORBA::Request::operator <<(const char&)
	1.34	CORBA::Request::operator <<(const short&)
	1.34	CORBA::Request::operator <<(const float&)
	1.02	NullCoder::codeChar(char&)
	0.95	NullCoder::codeDouble(double&)
	0.91	_IDL_SEQUENCE_PerfStruct::encodeOp (CORBA::Request&) const
	0.87	NullCoder::codeLong(long&)
	0.87	NullCoder::codeFloat(float&)
	0.87	NullCoder::codeShort(short&)

Figure 4: Sender side overheads

essential that these problem areas be eliminated.

3 Compiler Optimizations for Improving DOC Performance

The previous section and our earlier work [23, 22] measure several key sources of overhead incurred by conventional CORBA implementations. In general, lower-level mechanisms like C and C++ versions of sockets perform significantly better than CORBA. This performance gap presents a serious problem for mission/life-critical applications (such as medical imaging), where the use of low-level tools increases development effort and reduces system reliability, flexibility, and reuse.

Figure 6 identifies the primary sources of CORBA overhead addressed by our current research. These include integrated-GIOP (General Inter-ORB Protocol) transport protocol performance [17], data copying and data inspection, presentation layer conversions, and demultiplexing of CORBA remote operation requests. We are developing an optimization framework that will reduce these key high-cost sources of overhead for CORBA implementations over high-speed ATM networks. At the core of our work is a CORBA optimization compiler that employs the solutions proposed below.

- **CORBA GIOP Protocol Optimizations:** The General Inter-ORB Protocol (GIOP) [17] provides the basic functionality necessary for interoperability between different ORB implementations. We are implementing a lightweight transport protocol implementation of the CORBA GIOP for ATM LANs. We will implement this lightweight GIOP

Program	%Time	Method Name
C	64.65	read
receiver	35.08	getmsg
C++	50.43	read
receiver	49.38	getmsg
Orbix	19.11	read
receiver	14.44	CHECK(unsigned long, cursor*)
	7.22	CORBA::Request::operator >>(float&)
	7.22	CORBA::Request::operator >>(double&)
	7.22	CORBA::Request::operator >>(short&)
	7.22	CORBA::Request::operator >>(char&)
	7.22	CORBA::Request::operator >>(long&)
	5.10	NullCoder::codeChar(char&)
	4.88	_IDL_SEQUENCE_PerfStruct::decodeOp (CORBA::Request&)
	4.88	NullCoder::codeDouble(double&)
	4.46	NullCoder::codeLong(long&)
	4.46	NullCoder::codeFloat(float&)
	4.46	NullCoder::codeShort(short&)
	0.19	write
	0.15	_free_unlocked
	0.14	ioctl
	0.10	putmsg

Figure 5: Receiver side overheads

protocol implementation with flexible, application-tailored transport functionality [24]. For cases where the application traffic characteristics do not require complete reliability (which is the case for teleconferencing for example), we will omit transport layer retransmission and error handling altogether to run directly atop ATM. Our GIOP transport layer will be implemented into the Solaris 2.x OS as a kernel-level STREAMS module that is tightly integrated to the underlying ATM infrastructure via techniques Application Layer Framing/Integrated Layer Processing [5, 1, 4, 11, 20] techniques. ILP requires ordering constraints and hence it is necessary to perform control and data flow analysis of the code to extract the dependencies. This information will be used to automatically incorporate ILP into the implementation.

- **Remote operation demultiplexing optimizations:** The remote object implementation is typically represented by an object reference and the operation is typically represented as a string or as a binary value. The type of demultiplexing scheme used by an ORB can significantly impact performance. Some ORBs demultiplex incoming messages by linearly searching through the list of object implementations and operations in the IDL interface. Linear search does not perform well for interfaces that define a large number of methods. Good hashing schemes can reduce the search time, but these schemes are not well suited for smaller interfaces. Thus we plan to develop a set of *adaptive* optimizations for CORBA request demultiplexing. In this case, it is necessary to analyze the CORBA IDL interface and decide which strategy provides optimal performance. We are also incorporating a strategy that orders the requests according to the frequency of their usage. Thus, demultiplexing these requests can be performed efficiently depending on information collected dynamically. We are using compiler techniques (such as *header prediction* [6]) to automate this. Principles underlying header prediction can be used by the receiver side to predict the incoming request and hence can efficiently demultiplex it. This prediction can be based on factors such as frequency of use and locality of reference.

- **Data copying optimizations:** IDL skeletons generated automatically by a CORBA IDL compiler do not know how the user-supplied upcall will use the parameters passed to it from the request message. Thus, they use conservative memory management techniques that dynamically allocate and release copies of messages before and after an upcall, respectively. However, this strategy needlessly increases processing overhead for streaming applications like `ttcp` that consume their data immediately without modifying it. Therefore, we are using Integrated Layer Processing (ILP) [5] to reduce the data copy operations. Since ILP requires that ordering constraints be maintained, this requires other compiler techniques (such as control and data flow analysis) to provide insight into where to employ ILP.

- **Presentation layer optimizations:** Our framework will automatically produce and configure multiple encoding/decoding strategies for CORBA IDL definitions, each amenable for different conditions (such as time/space tradeoffs between compiled vs. interpreted CORBA IDL stubs and skeletons). Using dynamic linking strategies, it would be possible to include an appropriate marshalling stub for a given data type based on its run time usage by the application. Marshalling involves accessing data and moving data. For these reasons, it becomes necessary to employ efficient buffer and memory management schemes that reduce the amount of data movement.

To reduce marshalling overhead, our framework automatically *caches* certain types of request information. Caching is employed when certain types of application data units (ADUs) are transferred sequentially in "request chains." In cases where ADUs contain a large number of subparts that remain constant, only a few vary from one transmission to the other. By having the framework cache the marshalled information for the largely constant subparts and only allowing marshalling of the varying quantities, the marshalling overhead can be reduced significantly. This optimization requires flow analysis [3, 7] of the application code to determine which information can be cached.

Another scenario results from the fact that in some cases, it may not be necessary to convert some data structures into the host format at all. In such cases, greater efficiency can result if marshalling is not done for such data types. An efficient technique called the *Shapes Choice* problem [14] can be used to solve the marshalling overhead in situations described above. In the Shapes Choice problem, a cost is associated with the cross product of each kind of field reference and the two shapes (*e.g.,* wire format and host machine for-

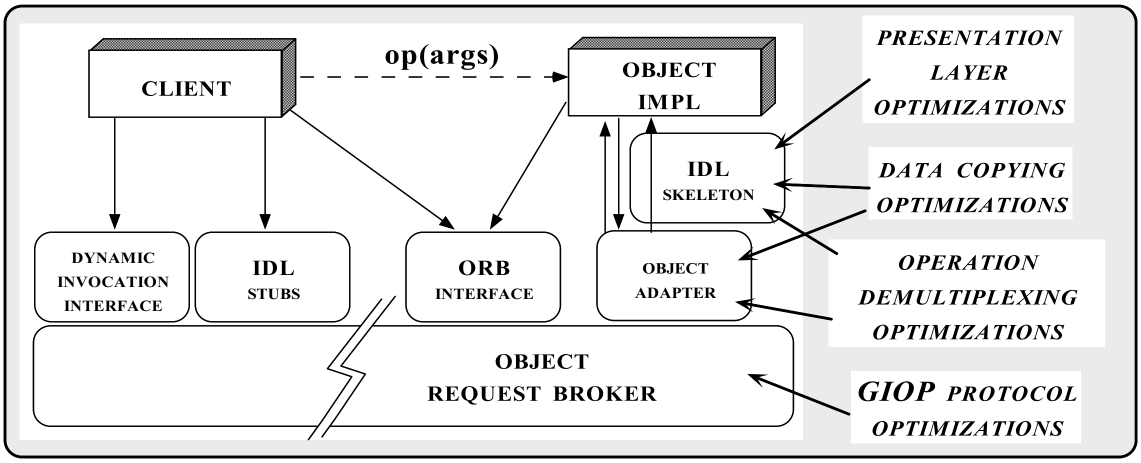

Figure 6: CORBA Components and Optimization Opportunities

mat). A cost is associated with converting in each direction. Conversions of only those data types are done for which the global cost is minimized.

4 Related Work

Our work on distributed object computing focuses on topics related to the transport, presentation, and application layers of the protocol stack. Much of the existing work has focussed on enhancing the TCP/IP protocol suite. Less attention has been paid to performance of DOC frameworks over high-speed networks. We classify the related work as follows.

• **Transport Protocol Performance over ATM networks:** [10, 15, 8] present results on performance of TCP/IP (and UDP/IP [8]) on ATM networks by varying a number a parameters (such as TCP window size, socket queue size, and user data size). This work indicates that in addition to the host architecture and host network interface, parameters configurable in software (like TCP window size, socket queue size and user data size) significantly affect TCP throughput. [8] also shows that UDP performs better than TCP over ATM networks, which is attributed to redundant TCP processing overhead on highly-reliable ATM links.

• **Demultiplexing:** Demultiplexing is a task that 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. Conventional CORBA implementations utilize several additional levels of demultiplexing at the application layer to associate incoming CORBA requests with the appropriate object implementation and method. Layered multiplexing and demultiplexing is generally disparaged for high-performance communication systems [26] due to the additional overhead incurred at each layer. Our framework uses a delayered demultiplexing architecture to select optimal demultiplexing strategies based on compile-time and run-time analysis of CORBA IDL interfaces.

• **Presentation layer and data copying:** The presentation layer is a major bottleneck in high-performance communication subsystems [5]. This layer transforms typed data objects from higher-level representations to lower-level representations (marshalling) and vice versa (demarshalling). In both RPC and DOC frameworks, this transformation process is performed by client-side stubs and server-side skeletons that are generated by interface definition language (IDL) compilers. IDL compilers translate interfaces written in an IDL (such as XDR [25], NDR [9], or CDR [12]) to other forms such as a network wire format. A significant amount of research has been devoted to developing efficient stub generators. We cite a few of these and classify them as below.

- *Annotating high level programming languages* – The Universal Stub Compiler (USC) [18] annotates the C programming language with layouts of various data types. The USC stub compiler supports the automatic generation of device and protocol header marshalling code. The USC tool generates optimized C code that automatically aligns data structures and performs network/host byte order conversions.
- *Generating code based on Control Flow Analysis of interface specification* –

  [13] describes a technique of exploiting application-specific knowledge contained in the type specifications of an application to generate optimized marshalling code. This work tries to achieve an optimal tradeoff between interpreted code (which is slow but compact in size) and compiled code (which is fast but larger in size). A frequency-based ranking of application data types is used to decide between interpreted and compiled code for each data type. Our implementations of the stub compiler will be designed to adapt according to the run-

time access characteristics of various data types and methods. Depending on the runtime usage of a given data type or method, our framework will dynamically link in either the compiled or the interpreted version. Dynamic linking has been shown to be useful for mid-stream adaptation of protocol implementations [20].

- *Using high level programming languages for distributed applications* – [19] describes a stub compiler for the C++ language. This stub compiler does not need an auxiliary interface definition language. Instead, it uses the operator overloading feature of C++ to enable parameter marshalling. This approach enables distributed applications to be constructed in a straightforward manner. A drawback of using a programming language like C++ is that it allows programmers to use constructs (such as references or pointers) that do not have any meaning on the remote side. Instead, IDLs are more restrictive and disallow such constructs. CORBA IDL has the added advantage that it resembles C++ in many respects and a well-defined mapping from the IDL to C++ has been standardized.

• **Application Level Framing and Integrated Layer Processing on communication subsystems:** Conventional layered protocol stacks lack the flexibility and efficiency required to meet the quality of service requirements of diverse applications running over high speed networks. A remedy for this problem is to use *Application Level Framing* (ALF) [4, 11] and *Integrated Layer Processing* (ILP) [5, 1, 20]. ALF ensures that lower layer protocols deal with data in units specified by the application. ILP provides the implementor with the option of performing all data manipulations in one or two integrated processing loops, rather than manipulating the data sequentially.

None of the systems described above are targeted for the requirements and constraints of distributed object computing frameworks. DOC frameworks are characterized by an integrated approach that supports *platform heterogeneity, high system reliability, efficient marshalling/demarshalling of parameters, flexible and efficient object location and selection, and higher level mechanisms for collaboration among services* [16]. To meet these requirements and to enhance functionality provided by traditional procedural RPC toolkits (such as Sun RPC and OSF DCE), DOC frameworks support object-oriented language features. Many sophisticated components must be developed to support features such as remote method invocation, transparent object location and activation, and service selection. These components include directory name servers, object request brokers (ORBs), interface definition language compilers and object locators/traders. We plan to enhance previously described ideas and propose newer schemes for efficient object-to-object communication in DOC environments.

5 Concluding Remarks

The main thesis of our work is that advances in communication software can be achieved only by simultaneously integrating techniques and tools that simplify application development, optimize application performance, and systematically measure application behavior in order to pinpoint and alleviate performance bottlenecks. Our work is motivated by an increasing demand for efficient and flexible communication software to support next-generation multimedia applications and to leverage emerging high-speed networking technology. This paper outlines how we are applying compiler techniques to develop flexible CORBA implementations that are optimized for performance-sensitive applications running over high-speed ATM networks.

Thus far, our framework consists of two parts:

1. A testbed to perform experiments that precisely pinpoint sources of overhead of CORBA over ATM networks.

2. A tool that can use sophisticated compiler optimization techniques to incorporate different optimizations in the CORBA implementations such as efficient marshalling, reduced data copying, integrated layer processing and effective demultiplexing of object requests.

Our long-term goal is to ease the development of flexible, performance-sensitive communication software.

References

[1] M. Abbott and L. Peterson. Increasing Network Throughput by Integrating Protocol Layers. *ACM Transactions on Networking*, 1(5), October 1993.

[2] G.J Blaine, M.E. Boyd, and S.M. Crider. Project Spectrum: Scalable Bandwidth for the BJC Health System. *HIMSS, Health Care Communications*, pages 71–81, 1994.

[3] Jong-Deok Choi, Ron Cytron, and Jeanne Ferrante. Automatic Construction of Sparse Data Flow Evaluation Graphs. In *Conference Record of the Eighteenth Annual ACE Symposium on Principles of Programming Languages*. ACM, January 1991.

[4] Isabelle Chrisment. Impact of ALF on Communication Subsystems Design and Performance. In *First International Workshop on High Performance Protocol Architectures, HIPPARCH '94*, Sophia Antipolis, France, December 1994. INRIA France.

[5] D. D. Clark and D. L. Tennenhouse. Architectural Considerations for a New Generation of Protocols. In *SIGCOMM Symposium on Communications Architectures and Protocols*, Philadelphia, Pennsylvania, September 1990. ACM.

[6] David D. Clark, Van Jacobson, John Romkey, and Howard Salwen. An Analysis of TCP Processing Overhead. *IEEE Communications Magazine*, 6(23), June 1989.

[7] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing Static Single Assignment form and the Control Dependence Graph. In *ACM Transactions on Programming Languages and Systems*. ACM, October 1991.

[8] Sudheer Dharnikota, Kurt Maly, and C. M. Overstreet. Performance Evaluation of TCP(UDP)/IP over ATM networks. Department of Computer Science, Technical Report CSTR-94-23, Old Dominion University, September 1994.

[9] John Dilley. OODCE: A C++ Framework for the OSF Distributed Computing Environment. In *Proceedings of the Winter Usenix Conference*. USENIX Association, January 1995.

[10] Minh DoVan, Louis Humphrey, Geri Cox, and Carl Ravin. Initial Experience with Asynchronous Transfer Mode for Use in a Medical Imaging Network. *Journal of Digital Imaging*, 8(1):43–48, February 1995.

[11] Atanu Ghosh, Jon Crowcroft, Michael Fry, and Mark Handley. Intergrated Layer Video Decoding and Application Layer Framed Secure Login: General Lessons from Two or Three Very Different Applications. In *First International Workshop on High Performance Protocol Architectures, HIPPARCH '94*, Sophia Antipolis, France, December 1994. INRIA France.

[12] Object Management Group. Common Object Request Broker Architecture: Architecture and Specification. (draft) edition 2.0, Object Management Group, May 1995.

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

[14] Mary E. Mace. *Memory Storage Patterns in Parallel Processing*. Kluwer Academic Publishers, Norwell, Massachusetts, 1987.

[15] K. Modeklev, E. Klovning, and O. Kure. TCP/IP Behavior in a High-Speed Local ATM Network Environment. In *Proceedings of the* 19^{th} *Conference on Local Computer Networks*, pages 176–185, Minneapolis, MN, October 1994. IEEE.

[16] Object Management Group. *Common Object Services Specification, Volume 1*, 94-1-1 edition, 1994.

[17] Object Management Group. *Universal Networked Objects*, TC Document 95-3-xx edition, March 1995.

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

[19] Graham Parrington. A Stub Generation System for C++. *Computing Systems*, 8(2):135–170, Spring 1995.

[20] Antony Richards, Ranil De Silva, Anne Fladenmuller, Aruna Seneviratne, and Michael Fry. The Application of ILP/ALF to Configurable Protocols. In *First International Workshop on High Performance Protocol Architectures, HIPPARCH '94*, Sophia Antipolis, France, December 1994. INRIA France.

[21] Dennis Ritchie. A Stream Input–Output System. *AT&T Bell Labs Technical Journal*, 63(8):311–324, October 1984.

[22] Douglas C. Schmidt, Andy Gokhale, and Tim Harrison. Experience Developing an Object-Oriented Framework for High-Performance Electronic Medical Imaging using CORBA and C++. In *Software Technology Applied to Imaging and Multimedia Applications miniconference at the Symposium on Electronic Imaging in the International Symposia Photonics West*. SPIE, January 1996.

[23] Douglas C. Schmidt, Tim Harrison, and Ehab Al-Shaer. Object-Oriented Components for High-speed Network Programming. In *Conference on Object-Oriented Technologies, COOTS95*, Monterry, CA, June 1995. Usenix.

[24] Douglas C. Schmidt, Burkhard Stiller, Tatsuya Suda, Ahmed Tantawy, and Martina Zitterbart. Language Support for Flexible, Application-Tailored Protocol Configuration. In *Proceedings of the* 18^{th} *Conference on Local Computer Networks*, pages 369–378, Minneapolis, Minnesota, September 1993. IEEE.

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

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

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