Flexible and Optimized IDL Compilation for Distributed Applications

Eric Eide, Jay Lepreau, and James L. Simister

University of Utah Department of Computer Science
3190 MEB, Salt Lake City, Utah 84112

flick@cs.utah.edu http://www.cs.utah.edu/projects/flux/flick/

Abstract. The author of a distributed system is often faced with a dilemma when writing the system's communication code. If the code is written by hand (e.g., using Active Messages) or partly by hand (e.g., using MPI) then the speed of the application may be maximized, but the human effort required to implement and maintain the system is greatly increased. On the other hand, if the code is generated using a high-level tool (e.g., a CORBA IDL compiler) then programmer effort will be reduced, but the performance of the application may be intolerably poor. The tradeoff between system performance and development effort arises because existing communication middleware is inefficient, imposes excessive presentation layer overhead, and therefore fails to expose much of the underlying network performance to application code. Moreover, there is often a mismatch between the desired communication style of the application (e.g., asynchronous message passing) and the communication style of the code produced by an IDL compiler (synchronous remote procedure call). We believe that this need not be the case, but that established optimizing compiler technology can be applied and extended to attack these domain-specific problems.

We have implemented *Flick*, a flexible and optimizing IDL compiler, and are using it to explore techniques for producing high-performance code for distributed and parallel applications. Flick produces optimized code for marshaling and unmarshaling data; experiments show that Flick-generated stubs can marshal data between 2 and 17 times as fast as stubs produced by other IDL compilers. Further, because Flick is implemented as a "kit" of components, it is possible to extend the compiler to produce stylized code for many different application interfaces and underlying transport layers. In this paper we outline a novel approach for producing "decomposed" stubs for a distributed global memory service.

This research was supported in part by the Defense Advanced Research Projects Agency, monitored by the Department of the Army under contract number DABT63–94–C–0058, and Air Force Research Laboratory, Rome Research Site, USAF, under agreement number F30602–96–2–0269. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation hereon.

© Springer-Verlag. Published in David O'Hallaron, editor, *Languages, Compilers, and Run-Time Systems for Scalable Computers (LCR '98)*, volume 1511 of *Lecture Notes in Computer Science*, pages 288–302. Springer, May 1998.

1 Introduction

Modern computer networks are becoming increasingly fast, and modern operating systems are now supporting efficient, lightweight communication mechanisms such as shared memory-based intra-node communication channels [1], highly optimized kernel IPC paths [11, 22], and new inter-node communication models such as active messages [31] and sender-based protocols [5, 30]. As Clark and Tennenhouse predicted in 1990 [8], these improvements in low-level communication systems have moved the performance bottlenecks for distributed applications out of the network and operating system layers and into the applications themselves [19, 20].

It is now possible to write high-performance, distributed applications that efficiently utilize network bandwidth. However, in order to *realize* the benefits of modern communication networks, the authors of parallel and distributed applications are generally forced to write their applications' communication code by hand. In order to squeeze the necessary performance out of the communication code, programmers must often write their own routines to marshal and unmarshal data, to dispatch and receive messages, and to handle communication failures. Writing this code requires a great amount of effort, is error-prone, and places a significant maintenance burden on the application programmer. Unfortunately, all this effort is necessary because today's high-level tools for producing communication code are inadequate for the demands of high-performance distributed applications.

An *interface definition language* (IDL) compiler is one tool for producing communication code with minimal programmer effort. An IDL compiler reads a high-level specification of an interface to a software component, and from that produces "stubs": functions that encapsulate the communication between the *client* that invokes an operation and the *server* that implements the operation. The stubs output by the IDL compiler hide the details of communication and allow the client and server to interact through a procedural interface. For example, consider the following CORBA [24] IDL specification of a simple file server:

```
interface FS {
  typedef sequence<octet> content;

  content read(in string filename);
  void    write(in string filename, in content data);
};
```

Given this interface, a CORBA IDL compiler for C will generate the following stub prototypes for the `read` and `write` operations:[1]

[1] For clarity, we have omitted the declaration of the `FS` object type and the declaration of the `FS_content` type.

```
FS_content *FS_read(FS obj,
                    CORBA_char *filename,
                    CORBA_Environment *env);

void FS_write(FS obj,
              CORBA_char *filename, FS_content *data,
              CORBA_Environment *env);
```

The compiler will also output implementations of the `FS_read` and `FS_write` stubs that implement *remote procedure call* (RPC) [4] or *remote method invocation* (RMI) semantics: the client that invokes `FS_read` or `FS_write` will block until the server replies to the client's request. From the client's point of view, the `FS_read` and `FS_write` functions behave like normal, local procedure calls.

IDLs and IDL compilers can increase programmer productivity and eliminate the errors that are common in hand-written communication code. Traditionally, however, this benefit has come at a cost: poor optimization of the IDL-derived code. The code generated by existing IDL compilers like `rpcgen` [29] contains a great deal of overhead; Schmidt et al. [19] and others have quantified this overhead for a number of existing CORBA implementations. This overhead is often not a problem for applications that make infrequent RPC calls; similarly, it is often not a problem if the underlying network layers are slow (thus causing the network or operating system layers to be the performance bottleneck). However, the inefficiency of IDL-derived code can make it completely unsuitable for use in an application that demands maximum RPC performance. For this reason, the designers of highly distributed and parallel applications have been unable to take advantage of tools that ease the creation of communication code.

Beyond the substantial concerns about performance, there is a further and perhaps more significant barrier to the use of IDL compilers for many distributed applications: namely, that traditional IDL compilers can produce only RPC or RMI stubs. Many distributed applications require greater control over their communication; other applications simply have communication patterns that do not easily mesh with traditional RPC semantics. In order to support these kinds of applications, an IDL compiler must be flexible: it must give the application writer a great deal of control over the stubs that are generated by the IDL compiler. This level of control is also required in order to provide a migration path for existing applications already written to other communication models such as MPI [23].

We believe that parallel and distributed applications would be improved if they were designed to use either IDLs or other languages incorporating optimized presentation layer code, and that the barriers described above can be eliminated by bringing well-established compiler techniques to bear on the domain of IDL compilation. To explore these ideas we have implemented *Flick* [10], the Flexible IDL Compiler Kit. Flick is a "toolkit" of compiler components that may be specialized for particular IDLs, stub styles, and transport systems. Flick has three separate front ends that parse interface specifications written in the CORBA [24], ONC RPC (a.k.a. Sun RPC) [28], and MIG [26] IDLs. A specification in any of these

languages is sent through a series of separate compiler passes to produce optimized CORBA-, `rpcgen`-, or MIG-style C stubs communicating via TCP, Mach [1] messages, Trapeze [2] messages, or Fluke [15] kernel IPC. Section 2 summarizes the structure of Flick, and Sec. 3 describes how we are extending Flick to create specialized, "decomposed" stubs for a class of distributed applications heretofore unsupported by traditional IDL compilers. Our new IDL compilation techniques provide fine-grained control over communication and will soon be applied to generating stubs for Khazana [6], a distributed global memory service.

2 Flick

Figure 1 shows the design of the Flick IDL compiler. Flick is organized like a traditional language compiler with separate front ends, intermediate code generators, and back ends.

The first phase of the compiler is the *front end*, which reads an IDL specification and produces an intermediate file called an Abstract Object Interface (AOI). The second phase, the *presentation generator*, reads the AOI file produced by the front end and outputs a file that describes the programmer's interface to the stubs that will be generated by the compiler. This is called the *presentation* of the interface. A presentation is a "programmer's contract" that describes everything an application programmer must know in order to use the functions and data type declarations created by an IDL compiler. This contract includes such things as the names of the generated functions, the types of their arguments, the memory allocation conventions, and so on.

Consider the stub prototypes shown previously for the `read` and `write` operations of our simple filesystem interface. Those prototypes conform to the CORBA specification for mapping IDL constructs onto the C programming language. However, that mapping is not the only possible mapping, nor it is necessarily the most useful one. Previous experiments [13, 14] have shown that by modifying the mapping from IDL to stub, a programmer can adapt the stubs to better meet the needs of a particular system. Section 3 will explore this idea further.

Flick provides support for different presentation styles through its multiple presentation generators. Each presentation generator implements a specific presentation style for a particular programming language (e.g., the CORBA C language mapping). The output of the presentation generator is another intermediate file, called a Presentation in C (PRES_C) file.

The third and final phase of the compiler, the *back end*, transforms the presentation description into the source code that will implement client/server communication. Each of Flick's back ends is specialized for a particular communications infrastructure (e.g., CORBA IIOP or Mach typed messages). The back end decides the internals of the stubs — message format, message data encoding, and underlying transport facility — and finally produces the optimized stub code.

Flick's back ends apply numerous domain-specific optimizations to reduce the overheads that typically hinder IDL-based communication. For instance, Flick analyzes the overall storage requirements of every message and for each produces

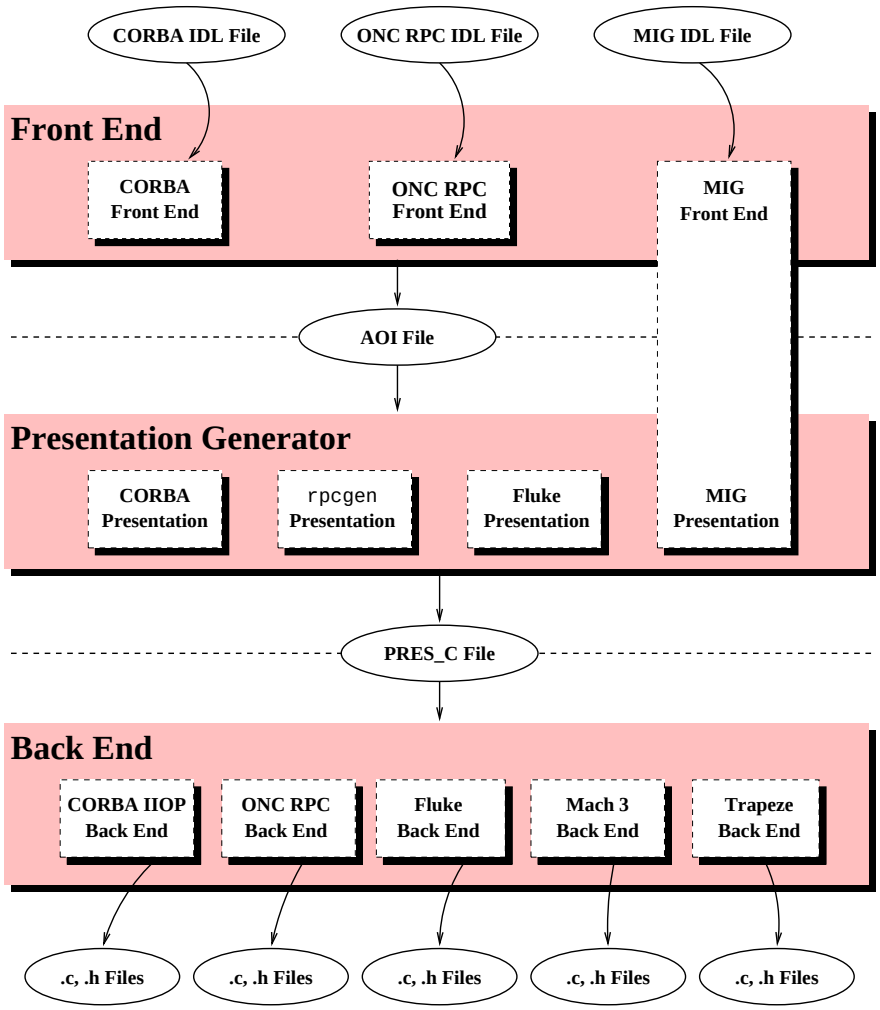

Fig. 1. The components of the Flick IDL compiler. Flick is implemented as three separate compiler passes which communicate through intermediate files. Each pass is primarily implemented by a large library of code which is specialized in order to create particular compiler components.

efficient memory allocation code. Flick also exploits opportunities to copy objects with efficient copy routines; for example, arrays of integers may be copied with `memcpy` when Flick is able to determine that no byte-swapping or other presentation layer conversion is required. Previous experiments with Flick [10] show that Flick-generated stubs can marshal data between 2 and 17 times as fast as stubs generated by other IDL compilers. This reduction in presentation layer overhead results in significant end-to-end throughput improvements for applications running on stock operating systems, communicating over TCP.

3 IDL Compilation for a Global Memory Service

As described in Sec. 2, the *presentation* of an interface is a description of everything that client or server code must know in order to use the stubs and other code output by an IDL compiler. There may be many different presentations of a single interface; that is, there are many different ways of translating an IDL specification into bits of code to implement the interface. Traditional IDL compilers produce RPC stubs, but this is not the only possible way to implement an interface, and may in fact be a bad match with the needs of particular distributed or parallel applications. This section presents one such application and describes a novel presentation style for CORBA IDL that we are implementing within Flick, thus allowing Flick to accommodate the application in a way that traditional IDL compilers cannot.

3.1 Khazana

The application is Khazana [6], a "global memory service" being developed by Carter et al. at the University of Utah. Khazana is a middleware layer for distributed applications that provides the abstraction of a distributed, persistent, globally shared store in which applications may save state. Khazana is implemented as a collection of peer processes; there is no single "server" process. Instead, all of the Khazana processes cooperate in order to provide the illusion of a single shared resource.

Khazana processes communicate by exchanging asynchronous messages. Although there are conceptual requests and replies, Khazana nodes must be able to process events after a request is sent but before the corresponding reply is received. When responding to a request, a Khazana node may discover that it needs to forward the message to another node (e.g., the home node of some requested Khazana memory page). A node may also discover that it can only service part of a request, and that the request needs to be "pushed back" onto the queue of incoming requests along with the partial results that have been so far computed.

The current implementation of Khazana uses hand-coded functions to handle the exchange of messages between nodes, perform asynchronous sends and receives, and save partially handled requests. The Khazana implementors, however, anticipate that as development continues, the number of message types will

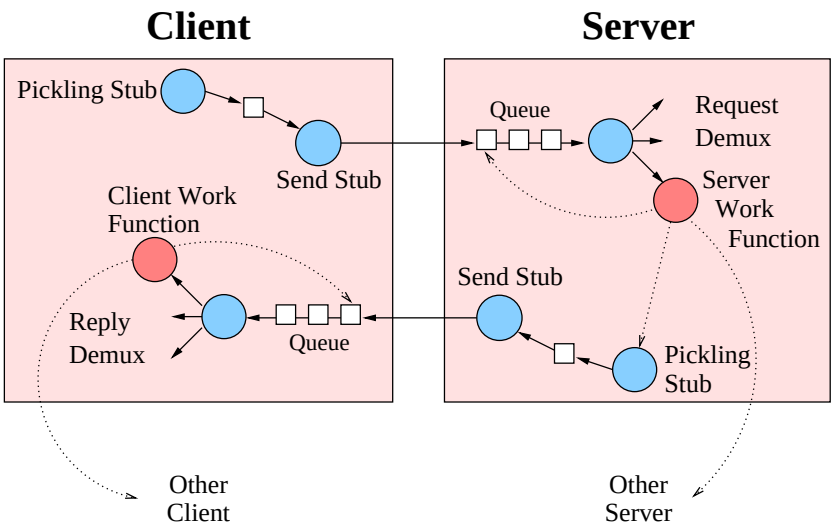

Fig. 2. A new presentation style for CORBA interfaces. Unlike the standard presentation of CORBA interfaces, this new presentation allows for asynchronous requests and replies, efficient message forwarding, and saving intermediate results.

increase and that the burden of writing all the messaging code will be overwhelming. For this reason they are interested in using Flick to generate the Khazana communication stubs — but the generated code must meet the requirements outlined above. The design of Khazana is incompatible with the traditional presentation of IDL interfaces as simple, synchronous RPC stubs.

3.2 Decomposed Stubs: A New Presentation Style for CORBA

We have therefore designed a new presentation style for CORBA IDL specifications that breaks apart traditional RPC stubs and better meets the needs of Khazana. This new presentation style is illustrated in Fig. 2 and consists of:

- *pickling stubs*, to marshal (encode) request and reply messages;
- *unpickling stubs*, to unmarshal (decode) request and reply messages;
- *send stubs*, to transmit pickled messages to other nodes;
- *server work functions*, to handle received requests;
- *client work functions*, to handle received replies; and
- *continuation stubs*, to postpone the processing of messages.

The stub and function prototypes generated by Flick for this new "decomposed" presentation style are summarized in Table 1.

Table 1. Summary of the stubs and functions that are part of Flick's "decomposed" CORBA presentation style. The italicized elements (e.g., *Interface*) are replaced with the appropriate names and elements from the input IDL file.

Pickling Stubs	*Interface*_Request *Interface_operation*_encode_request(
	  in and inout parameters,
	  CORBA_Environment *env);
	*Interface*_Reply *Interface_operation*_encode_reply(
	  return and inout and out parameters,
	  CORBA_Environment *env);
	*Interface*_Reply *Interface_operation*_encode_exception(
	  CORBA_Environment *reply_env,
	  CORBA_Environment *env);

Unpickling Stubs	void *Interface_operation*_decode_request(
	  *Interface*_Request msg,
	  in and inout parameters,
	  CORBA_Environment *env);
	void *Interface_operation*_decode_reply(
	  *Interface*_Reply msg,
	  return and inout and out parameters,
	  CORBA_Environment *env);

Send Stubs	void *Interface*_send_request(
	  Interface target, *Interface*_Request msg,
	  Invocation_id inv_id, Client client,
	  CORBA_Environment *env);
	void *Interface*_send_reply(
	  Client client, *Interface*_Reply msg,
	  Invocation_id inv_id, *Interface* target,
	  CORBA_Environment *env);

Work Functions	void *Interface_operation*_do_request(
	  Interface target, *Interface*_Request msg,
	  Invocation_id inv_id, Client client);
	void *Interface_operation*_do_reply(
	  Client client, *Interface*_Reply msg,
	  Invocation_id inv_id, *Interface* target);

Continuation Stubs	typedef void *Interface*_Request_Continuer(...);
	typedef void *Interface*_Reply_Continuer(...);
	void *Interface_operation*_continue_request(
	  Interface target, *Interface*_Request msg,
	  Invocation_id inv_id, Client client,
	  *Interface*_Request_Continuer func, void *data);
	void *Interface_operation*_continue_reply(
	  Client client, *Interface*_Reply msg,
	  Invocation_id inv_id, *Interface* target,
	  *Interface*_Reply_Continuer func, void *data);

Pickling and Unpickling Stubs. Each operation in the IDL specification results in three "pickling" stubs: one to marshal operation requests, a second to marshal ordinary operation replies, and a third to marshal exceptional (error-signaling) replies. A pickling stub takes as parameters the data to be encoded, followed by a standard CORBA environment parameter used to communicate exceptions back to the caller (i.e., errors that occur as the message is being encoded). The pickled message is returned by the stub so that the application can send the message at a later time; that is, the encoding and transmission events are decoupled. Note that the type of the pickled message is specific to a single interface (object type); this helps prevent programming errors in which a request or reply is sent to an object of an inappropriate type. Also note that the returned message is opaque. This allows each of Flick's back ends to choose a message format, data encoding, and implementation that is best suited to the application's requirements and the underlying transport facility (e.g., CORBA IIOP messages, or some more specialized system).

Each operation also results in two "unpickling" stubs: one to unmarshal requests and one to unmarshal replies (ordinary or exceptional). An unpickling stub takes a pickled message, parameters into which the message data will be decoded (the CORBA `inout`, `out`, and return values), and a CORBA environment parameter that allows the unpickling stub to report decoding errors to the application.

Pickling and unpickling stubs provide applications with greater control over the handling of messages and enable certain application-specific optimizations that are not possible within a traditional RPC model. For example, a message can be pickled once and then sent to multiple targets; also, common replies can be premarshaled and cached, thus reducing response times. An especially useful optimization for Khazana is that a message can be redirected to another node without the overhead of decoding and re-encoding the message (assuming that the message data is not needed in order to make the forwarding decision).

Send Stubs. Once a request or reply message has been marshaled, an application transmits the message by invoking the appropriate send stub. Two send stubs are defined for each CORBA interface (object type): one for requests and another for replies. Unlike a traditional RPC stub, a send stub returns immediately after the message has been sent; it does not wait for a reply.

Also unlike a traditional RPC stub, the send stubs produced by Flick take two special parameters as shown in Table 1: an *invocation identifier* and a *client reference*. These parameters are unique to Flick's "decomposed" stub presentation style and are not standard CORBA stub parameters. An invocation identifier (`Invocation_id`) corresponds to a single message transmission event and is used to connect a reply with its associated request. The application is responsible for providing invocation identifiers to the send stubs; this allows for application-specific optimizations in the generation and management of the identifiers. A client reference (`Client`) is a CORBA pseudo-object [24], managed by the communications runtime layer, that provides additional contextual information for

handling requests and replies. A client reference serves two primary functions. First, it locates the entity that will receive the reply for a given request. Keeping explicit client references makes it possible, for instance, for one node to forward a request message to another node and direct the eventual reply back to the original requester. Second, the client reference allows the application to save state from the time a request is sent until the corresponding reply is received. A process can allocate as many client references as it needs and associate them with request transmission events as it sees fit. Then, whenever a reply message is received, the runtime locates the client reference that was provided with the original request, and gives that reference to the function that will process the newly received reply.

Server and Client Work Functions. The server and client work functions are the functions that ultimately receive and service request and reply messages, respectively. Traditional RPC presentations contain server work functions only; traditional clients process requests and replies synchronously. In contrast, Flick's decomposed stub presentation allows replies to be handled asynchronously, and therefore, clients must contain explicit functions to receive and handle incoming replies. A server work function receives the pickled representation of the request that it is to service. This makes it straightforward and efficient for a Khazana node to forward the request to another node, or to delay local processing. If the request is to be handled immediately, the server work function will invoke the unpickling stub to extract the request data. Similarly, a client work function receives a pickled reply message so that it may choose to handle the reply immediately, forward the reply (to another client), or delay processing by invoking a continuation stub.

Continuation Stubs. While handling a request or reply, a client or server work function may need to postpone processing — for instance, a server handling a request may find that it needs data from another server before it can continue. To allow applications to handle these situations gracefully, Flick produces two "continuation" stubs for each operation: one for the request and another for the reply. These are essentially normal send stubs, except that they direct the message back into the node's own message queue. Each continuation stub accepts a pickled message, a pointer to the function that will be invoked to service the continued message (i.e., a special work function), and an opaque pointer that allows the application to pass arbitrary state information along to the function that will resume processing.[2]

Dispatch Functions. Given an interface specification, Flick also produces the server dispatch function that is invoked to demultiplex incoming requests, and a client dispatch function that is invoked to demultiplex incoming replies. These

[2] We have not yet specified the interface that allows the application to tell the runtime layer when the new work function should be invoked.

functions, however, are not properly part of the presentation because they are visible only to the runtime layers, not to application code.

3.3 Decomposed Stubs for Distributed Applications

We expect that this new presentation style for CORBA interfaces will be useful not only to Khazana but also to other, similar parallel and distributed applications. Since this new style is more like traditional message-passing, we believe that it will be useful as a migration path for applications that wish to move away from hand-written communication code and toward the use of higher-level tools. This will reduce the application writers' burden and eliminate what is currently an error-prone process, without compromising application performance.

4 Related Work

Recent work by Schmidt et al. [19, 27] has quantified the impact of presentation layer overhead for `rpcgen` and two commercial CORBA implementations. On average, due to inefficiencies at the presentation and transport layers, compiler-generated stubs achieved only 16–80% of the throughput of hand-coded stubs.

To address these and similar performance issues, several attempts have been made to improve the code generated by IDL compilers. These are discussed in our earlier paper on Flick [10]. In summary, these other IDL compilers are either not very flexible (e.g., Mach's MIG compiler [26]) or they generate very slow code (e.g., ILU [21]).

Like Flick, the Concert/C distributed programming system [3] quite fully develops the concept of flexible presentation. The primary purpose of flexible presentation in Concert is to handle the vagaries of RPC mappings to different target languages, striving for a "minimal contract" in order to achieve maximal interoperability between target languages. However, this technique is not leveraged for optimizations. In earlier work [13, 14] we concentrated on leveraging Flick's explicit separation of presentation from interface in order to produce application-specialized stubs. We showed that programmer-supplied interface annotations that coerce the "programmer's contract" to applications' needs could provide up to an order of magnitude speedup in RPC performance.

The CORBA Event Service [25] is an Object Management Group standard for decoupling requests and replies between CORBA objects. The specification defines an *event channel* as an object that mediates communication between sets of suppliers and consumers. Because an event channel is a heavyweight object, it can provide many services — but these extra services may come at a price. To make use of any channel services, including asynchronous messaging, clients and servers must be specially written to communicate through event channels. This is in contrast to Flick's decomposed stubs which allow a client or server (or both) to use asynchronous messaging without cooperation or knowledge from the other side. Also, because event channels interpose on communication, they may introduce overheads that are not present in Flick's optimized stubs.

Asynchronous communication and message forwarding are not new ideas. Anderson et al. [2] describe these same mechanisms for a Global Memory Service [12] built on top of Trapeze. However, their work focuses on the transport layer rather than the presentation and application layers. Further, they provide no support for automatically generating stubs to exploit asynchronous communication and message forwarding. Our work focuses on presenting these mechanisms to the application and automatically generating the appropriate stubs. We believe that our work is complementary to that of Anderson et al.; Flick can leverage the benefits of an efficient transport system to produce optimized communication stubs.

For parallel applications, there are a large number of specialized programming languages such as CC++ [7], Fortran M [16], and Split-C [9]. In most of these cases the language handles marshaling and unmarshaling of parameters. However, it is our belief that the techniques used by Flick, and possibly even its code, could be incorporated into the compilers for these languages to substantially reduce presentation layer costs, e.g., by minimizing data copying. There are also a large number of parallel runtime systems providing various levels of abstraction and functionality, such as MPI [23], PVM [18], and Nexus [17]. Typically, these systems require the programmer to write the marshaling code by hand, although they do abstract away byte-swapping in order to accommodate heterogeneous machines. We believe these provide an attractive target for optimizations provided by Flick.

5 Conclusion

High-level tools such as IDL compilers can greatly reduce the effort and time required to implement distributed and parallel systems. Unfortunately, the limits of traditional IDL compilation often prevent the use of such tools in applications that require maximum performance. Traditional IDL compilers may produce stubs that have excessive presentation layer overheads; furthermore, traditional RPC stubs are not a good match for the needs of many modern systems.

We have described Flick, a novel, modular, and flexible IDL compiler that generates optimized code for a variety of stub styles and transport mechanisms. We have described how Flick is being adapted to meet the needs of a particular distributed application, Khazana, and provide a new style of CORBA-based stubs that are appropriate for other similar systems. The work presented here is currently in progress.

Availability

Complete Flick source code and documentation are available at `http://www.cs.utah.edu/projects/flux/flick/`.

References

[1] M. Accetta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian, and M. Young. Mach: A new kernel foundation for UNIX development. In *Proc. of the Summer 1986 USENIX Conf.*, pages 93–112, June 1986.

[2] D. C. Anderson, J. S. Chase, S. Gadde, A. J. Gallatin, K. G. Yocum, and M. J. Feeley. Cheating the I/O bottleneck: Network storage with Trapeze/Myrinet. In *Proc. of the USENIX 1998 Annual Technical Conf.*, pages 143–154, New Orleans, LA, July 1998.

[3] J. S. Auerbach and J. R. Russell. The Concert signature representation: IDL as an intermediate language. In *Proc. of the Workshop on Interface Definition Languages*, pages 1–12, Jan. 1994.

[4] A. D. Birrell and B. J. Nelson. Implementing remote procedure calls. *ACM Transactions on Computer Systems*, 2(1), Feb. 1984.

[5] G. Buzzard, D. Jacobson, M. Mackey, S. Marovich, and J. Wilkes. An implementation of the Hamlyn sender-managed interface architecture. In *Proc. of the Second Symp. on Operating Systems Design and Implementation*, pages 245–259, Seattle, WA, Oct. 1996. USENIX Assoc.

[6] J. Carter, A. Ranganathan, and S. Susarla. Khazana: An infrastructure for building distributed services. In *Proc. of the Eighteenth International Conf. on Distributed Computing Systems*, May 1998.

[7] K. M. Chandy and C. Kesselman. CC++: A declarative concurrent object oriented programming notation. Technical Report CS-TR-92-01, California Institute of Technology, Mar. 1993.

[8] D. D. Clark and D. L. Tennenhouse. Architectural considerations for a new generation of protocols. In *Proc. of the SIGCOMM '90 Symp.*, pages 200–208, 1990.

[9] D. E. Culler, A. Dusseau, S. C. Goldstein, A. Krishnamurthy, S. Lumetta, T. von Eicken, and K. Yelick. Parallel programming in Split-C. In *Proceedings of Supercomputing '93*, pages 262–273, Portland, OR, Nov. 1993.

[10] E. Eide, K. Frei, B. Ford, J. Lepreau, and G. Lindstrom. Flick: A flexible, optimizing IDL compiler. In *Proc. ACM SIGPLAN Conf. on Programming Language Design and Implementation*, pages 44–56, Las Vegas, NV, June 1997.

[11] D. R. Engler, M. F. Kaashoek, and J. O'Toole Jr. Exokernel: An operating system architecture for application-level resource management. In *Proc. of the 15th ACM Symp. on Operating Systems Principles*, pages 251–266, Copper Mountain, CO, Dec. 1995.

[12] M. J. Feeley, W. E. Morgan, F. H. Pighin, A. R. Karlin, and H. M. Levy. Implementing global memory management in a workstation cluster. In *Proc. of the 15th ACM Symp. on Operating Systems Principles*, pages 201–212, Copper Mountain, CO, Dec. 1995.

[13] B. Ford, M. Hibler, and J. Lepreau. Using annotated interface definitions to optimize RPC. Technical Report UUCS-95-014, University of Utah Department of Computer Science, Mar. 1995.

[14] B. Ford, M. Hibler, and J. Lepreau. Using annotated interface definitions to optimize RPC. In *Proc. of the 15th ACM Symp. on Operating Systems Principles*, page 232, 1995. Poster.

[15] B. Ford, M. Hibler, J. Lepreau, P. Tullmann, G. Back, and S. Clawson. Microkernels meet recursive virtual machines. In *Proc. of the Second Symp. on Operating Systems Design and Implementation*, pages 137–151, Seattle, WA, Oct. 1996. USENIX Assoc.

[16] I. T. Foster and K. M. Chandy. Fortran M: A language for modular parallel programming. *Journal of Parallel and Distributed Computing*, 25(1), Feb. 1995.

[17] I. T. Foster, C. Kesselman, and S. Tuecke. The Nexus task-parallel runtime system. In *Proceedings of First International Workshop on Parallel Processing*, pages 457–462, 1994.

[18] G. Geist and V. Sunderam. The PVM system: Supercomputer level concurrent computation on a heterogenous network of workstations. In *Sixth Annual Distributed-Memory Computer Conference*, pages 258–261, 1991.

[19] A. Gokhale and D. C. Schmidt. Measuring the performance of communication middleware on high-speed networks. *Computer Communication Review*, 26(4), Oct. 1996.

[20] A. Gokhale and D. C. Schmidt. Optimizing the performance of the CORBA Internet Inter-ORB Protocol over ATM. Technical Report WUCS–97–09, Washington University Department of Computer Science, St. Louis, MO, 1997.

[21] B. Janssen and M. Spreitzer. *ILU 2.0alpha8 Reference Manual*. Xerox Corporation, May 1996. `ftp://ftp.parc.xerox.com/pub/ilu/ilu.html`.

[22] J. Liedtke. Improving IPC by kernel design. In *Proc. of the 14th ACM Symp. on Operating Systems Principles*, Asheville, NC, Dec. 1993.

[23] Message Passing Interface Forum. *MPI-2: Extensions to the Message-Passing Interface*, July 1997. `http://www.mpi-forum.org/`.

[24] Object Management Group. *The Common Object Request Broker: Architecture and Specification*, 2.0 edition, July 1995.

[25] Object Management Group. Event service specification. In *CORBAservices Specification*, chapter 4. Object Management Group, Dec. 1997.

[26] Open Software Foundation and Carnegie Mellon University, Cambridge, MA. *Mach 3 Server Writer's Guide*, Jan. 1992.

[27] D. C. Schmidt, T. Harrison, and E. Al-Shaer. Object-oriented components for high-speed network programming. In *Proceedings of the First Conference on Object-Oriented Technologies and Systems*, Monterey, CA, June 1995. USENIX Assoc.

[28] R. Srinivasan. RPC: Remote procedure call protocol specification version 2. Technical Report RFC 1831, Sun Microsystems, Inc., Aug. 1995.

[29] Sun Microsystems, Inc. *ONC+ Developer's Guide*, Nov. 1995.

[30] M. R. Swanson and L. B. Stoller. Direct deposit: A basic user-level protocol for carpet clusters. Technical Report UUCS-95-003, University of Utah Department of Computer Science, Mar. 1995.

[31] T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser. Active messages: A mechanism for integrated communication and computation. In *Proc. of the 19th International Symp. on Computer Architecture*, pages 256–266, May 1992.