Flick: A Flexible, Optimizing IDL Compiler

Eric Eide Kevin Frei Bryan Ford Jay Lepreau Gary Lindstrom

University of Utah, Department of Computer Science
3190 M.E.B., Salt Lake City, Utah 84112

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

Abstract

An interface definition language (IDL) is a nontraditional language for describing interfaces between software components. IDL compilers generate "stubs" that provide separate communicating processes with the abstraction of local object invocation or procedure call. High-quality stub generation is essential for applications to benefit from component-based designs, whether the components reside on a single computer or on multiple networked hosts. Typical IDL compilers, however, do little code optimization, incorrectly assuming that interprocess communication is always the primary bottleneck. More generally, typical IDL compilers are "rigid" and limited to supporting only a single IDL, a fixed mapping onto a target language, and a narrow range of data encodings and transport mechanisms.

Flick, our new IDL compiler, is based on the insight that IDLs are true languages amenable to modern compilation techniques. Flick exploits concepts from traditional programming language compilers to bring both flexibility and optimization to the domain of IDL compilation. Through the use of carefully chosen intermediate representations, Flick supports multiple IDLs, diverse data encodings, multiple transport mechanisms, and applies numerous optimizations to all of the code it generates. Our experiments show that Flick-generated stubs marshal data between 2 and 17 times faster than stubs produced by traditional IDL compilers, and on today's generic operating systems, increase end-to-end throughput by factors between 1.2 and 3.7.

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 Rome Laboratory, Air Force Material Command, 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.

To appear in *Proceedings of the ACM SIGPLAN '97 Conference on Programming Language Design and Implementation* (PLDI), Las Vegas, NV, June 15–18, 1997.

1 Introduction

An *interface definition language* (IDL) is a special-purpose language for describing the interfaces of a software component. An IDL specification declares one or more interfaces; each interface declares a set of operations that may be invoked on objects implementing the interface. The input and output behavior of each operation is given by the IDL specification. For example, the following CORBA [18] IDL program declares a simple interface to an electronic mail service:

```
interface Mail {
    void send(in string msg);
};
```

A largely equivalent mail system interface would be defined in the ONC RPC[1] [23] IDL by this program:

```
program Mail {
    version MailVers {
        void send(string) = 1;
    } = 1;
} = 0x20000001;
```

As shown by these examples, an IDL program declares a set of functions or methods but does not describe the computations that those functions and methods perform. IDLs are typically independent of the programming language in which the components are themselves implemented, further decoupling interface from implementation.

An IDL compiler accepts an IDL interface specification and outputs an implementation of that specification. Typically, the implementation is a set of data type declarations and "stubs" written in a conventional programming language such as C, C++, or Java. The stubs encapsulate the communication that must occur between the entity that invokes an operation (i.e., the *client*) and the entity that implements the operation (i.e., the *server*). The stubs that are output by the

[1] ONC RPC was previously known as Sun RPC, and Sun's `rpcgen` is the standard compiler for the ONC RPC IDL. The numbers in the example ONC RPC IDL program are chosen by the programmer to identify components of the interface.

IDL compiler hide the details of communication and allow the client and server to interact through a procedural interface. Traditionally, stubs have implemented *remote procedure calls* (RPC) [3] or *remote method invocations* (RMI): the client and server are located in separate processes, and the stubs in each process communicate by exchanging messages through a transport medium such as TCP/IP. More recently, IDLs have become popular for defining high-level interfaces between program modules within a single process.

IDLs and IDL compilers arose for reasons familiar to any programming language veteran: descriptive clarity, programmer productivity, assurance of consistency, and ease of maintenance. Performance of IDL-generated code, however, has traditionally not been a priority. Until recently, poor or mediocre performance of IDL-generated code was acceptable in most applications: because interprocess communication was generally both expensive and rare, it was not useful for an IDL compiler to produce fast code. For performance critical applications, implementors resorted to hand-coded stubs — tolerating the accompanying greater software engineering costs. Some IDL compilers such as MIG [20] struck a middle ground by providing a language with a restricted set of structured data types, blended with programmer control over implementation details. This compromise could be likened to that provided by a traditional compiler that permits embedded assembly language. Although embedded "hints" can lead to performance gains, reliance on hints moves the burden of optimization from the compiler to the programmer, and has the additional effect of making the language nonportable or useful only within restricted domains.

Today, in almost every respect, IDL compilers lag behind traditional language compilers in terms of flexibility and optimization. IDL compilers such as Sun's `rpcgen` [25] are generally written "from scratch" and are implemented without incorporating modern compiler technologies such as multiple, flexible intermediate representations. The result is that today's IDL compilers are "rigid": they accept only a single IDL, they implement only a single, fixed mapping from an IDL specification to a target language, and they generate code for only one or two encoding and transport mechanisms. Today's IDL compilers still assume that the transport medium is inherently slow, and therefore, that optimization of the stubs will not yield significant speed increases. Modern network architectures, however, have moved the performance bottlenecks for distributed applications out of the operating system layers and into the applications themselves [5, 12, 13].

In this paper we show that in order to solve the problems inherent to existing IDL compilers, IDL compilation must evolve from an ad hoc process to a principled process incorporating techniques that are already well-established in the traditional programming language community. Although IDL compilation is a specialized domain, IDL compilers can be greatly improved through the application of concepts and technologies developed for the compilation of general programming languages. *Flick*, our Flexible IDL Compiler Kit, exploits this idea. Flick is designed as a "toolkit" of reusable components that may be specialized for particular IDLs, target language mappings, data encodings, and transport mechanisms. Flick currently has front ends that parse the CORBA [18], ONC RPC [23], and MIG [20] IDLs. Flick compiles an interface specification in any of these languages through a series of intermediate representations to produce CORBA-, `rpcgen`-, or MIG-style C stubs communicating via TCP, UDP, Mach [1] messages, or Fluke [10] kernel IPC. Flick's compilation stages are implemented as individual components and it is easy for a system designer to mix and match components at IDL compilation time in order to create the high-performance communication stubs that he or she needs. Further, the organization of Flick makes it easy to implement new component front ends, "presentation generators," and back ends.

Flick's design as a traditional language compiler promotes not only flexibility but also optimization. Flick implements techniques such as code inlining, discriminator hashing, and careful memory management to maximize the speed at which data can be encoded and decoded (*marshaled* and *unmarshaled*) for communication. Flick's optimization techniques are similar to those provided by modern optimizing compilers, but its domain-specific knowledge allows Flick to implement important optimizations that a general-purpose language compiler cannot. Most of Flick's techniques are implemented by an abstract C++ base class for code generators, and therefore, all back ends inherit the optimizations provided by the large code base. The results presented in Section 4 show that Flick-generated communication stubs are up to 3.7 times faster than those generated by other IDL compilers.

2 Flick

The Flick IDL compiler is divided into three phases as illustrated in Figure 1. These phases are analogous to those in a traditional language compiler and correspond to separable aspects of IDL compilation. Each phase is primarily implemented by a large, shared library of C and C++ code that provides abstractions for such things as IDL source constructs, target language data types, and "on the wire" message data types. Each of Flick's libraries implements a generic set of methods to manipulate these abstractions. The libraries are the bases for specializations that override the generic methods as necessary in order to implement behaviors peculiar or specific to a single IDL, language mapping, message format, or transport facility.

The first phase of the compiler is the *front end*. The front end reads an IDL source file and produces an abstract representation of the interface defined by the IDL input. This representation, called an Abstract Object Interface (AOI), describes the high-level "network contract" between a client

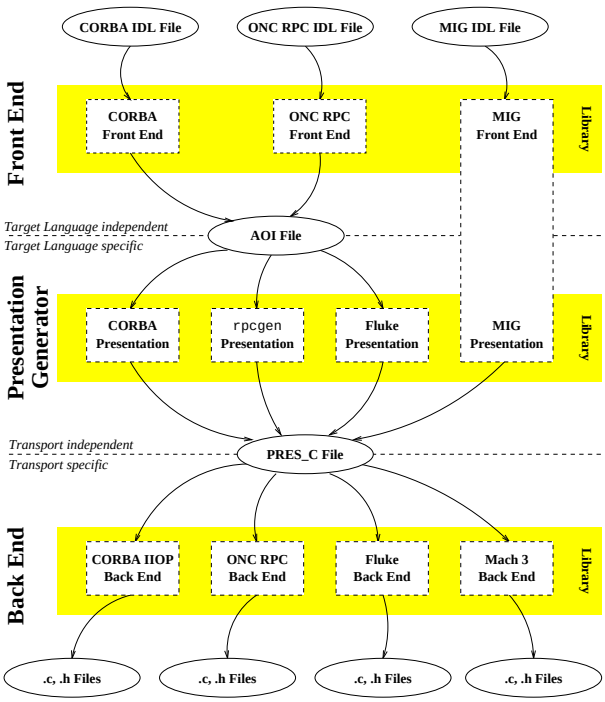

Figure 1: Overview of the Flick IDL Compiler. Flick is divided into three compilation phases, and each phase is implemented by a large library of code. Specialized components are derived from the Flick libraries in order to parse different IDLs, implement different target language mappings, and produce code for a variety of message formats and transport systems.

and a server: the operations that can be invoked and the data that must be communicated for each invocation.

Flick's second compilation phase, the *presentation generator*, reads the network contract produced by the front end and outputs a separate and lower-level "programmer's contract." The programmer's contract defines the interface between the programmer's client or server code and the stubs, e.g., how parameters are passed between them.

For example, consider the CORBA IDL input shown in Section 1 that defines a network contract between the client and server of a `Mail` interface. Given that input, a CORBA IDL compiler for C will always produce the following prototype describing the programmer's contract:[2]

```
void Mail_send(Mail obj, char *msg);
```

This programmer's contract declares the C functions and data types that will connect the client or server code to the stub: we say that this contract is a *presentation* of the interface in the C language.

The presentation shown above conforms to the CORBA specification for mapping IDL constructs onto the C programming language. However, it is not the only possible presentation of the `Mail` interface. For instance, if we depart from the CORBA mapping rules, the `Mail_send` function could be defined to take a separate message length argument:

```
void Mail_send(Mail obj, char *msg, int len);
```

This presentation of the `Mail` interface could enable optimizations because `Mail_send` would no longer need to count the number of characters in the message [8, 9]. This change to the presentation would *not* affect the network contract between client and server; the messages exchanged between client and server would be unchanged. The addition of a separate `len` parameter changes *only* the calling conventions for the `Mail_send` function. Flick's ability to handle different presentation styles can be important for optimization as just described, but it is also essential for supporting multiple IDLs in a reasonable way.

To summarize, a *presentation* describes everything that client or server code must understand in order to use the function and data type declarations output by an IDL compiler: this includes the names of the functions, the types of their arguments, the conventions for allocating memory, and so on. Because there can be many different presentations of a single interface, Flick provides multiple, different *presentation generators*, each implementing a particular style of presentation for a particular target programming language. When C is the target language, the presentation is described in an intermediate format called PRES_C. Because the presentation of an interface may differ between client and server, a presentation generator creates separate PRES_C files for the client- and server-side presentations of an interface.

The third and final phase of the compiler is the *back end*. The back end reads a presentation description (PRES_C) and produces the source code for the C functions that will implement client/server communication. The generated functions are specific to a particular message format, message data encoding scheme, and transport facility.

Table 1 compares the number of substantive C and C++ source code lines in each of Flick's libraries with the number of lines particular to each of Flick's specialized components. The number of lines specific to each presentation generator and back end is extremely small when compared to the size of the library from which it is derived. Front ends have significantly greater amounts of specialized code due to the need to scan and parse different IDL source languages.

2.1 Front Ends

As just described, the purpose of a Flick front end is to translate an interface description (source IDL program) to an intermediate representation. Each of Flick's front ends is specific to a particular IDL. However, each is completely *independent* of the later stages of IDL compilation: the presentation

[2] For clarity, we have omitted the declaration of the `Mail` object type and the `CORBA_Environment` parameter to the `Mail_send` function.

Phase	Component	Lines	
Front End	Base Library	1797	
	CORBA IDL	1661	48.0%
	ONC RPC IDL	1494	45.4%
Pres. Gen.	Base Library	6509	
	CORBA Library	770	10.6%
	CORBA Pres.	3	0.0%
	Fluke Pres.	301	4.0%
	ONC RPC `rpcgen` Pres.	281	4.1%
Back End	Base Library	8179	
	CORBA IIOP	353	4.1%
	ONC RPC XDR	410	4.8%
	Mach 3 IPC	664	7.5%
	Fluke IPC	514	5.9%

Table 1: Code Reuse within the Flick IDL Compiler. Percentages show the fraction of the code that is unique to a component when it is linked with the code for its base library. The CORBA presentation library is derived from the generic presentation library; the CORBA and Fluke presentation generators are derived from the CORBA presentation library.

of the interface that will be constructed, the target programming language that will implement the presentation, the message format and data encodings that will be chosen, and the transport mechanism that will be used. In sum, the output of a Flick front end is a high-level "network contract" suitable for input to any presentation generator and any back end.

Flick's MIG front end, however, is a special case. A MIG interface definition contains constructs that are applicable only to the C language and to the Mach message and IPC systems [20]. Therefore, as illustrated in Figure 1, Flick's MIG front end is conjoined with a special MIG presentation generator that understands these idioms. Flick's MIG components translate MIG interface descriptions directly into PRES_C representations; this is different than Flick's CORBA and ONC RPC front ends, which produce AOI. This difference reveals a strength: Flick's multiple intermediate representations provide the flexibility that is necessary for supporting a diverse set of IDLs.

2.1.1 AOI: The Abstract Object Interface

AOI is Flick's intermediate representation language for describing interfaces: the data types, operations, attributes, and exceptions defined by an IDL specification. AOI is applicable to many IDLs and represents interfaces at a very high level. It describes constructs independently of their implementation: for instance, AOI has separate notions of object methods, attributes, and exceptions, although all of these things are generally implemented as kinds of messages. AOI supports the features of typical existing IDLs such as the CORBA and ONC RPC IDLs, and Flick's front ends produce similar AOI representations for equivalent constructs across different IDLs. This "distillation process" is what makes it possible for Flick to provide a large and general library for the next stage of compilation, presentation generation.

2.2 Presentation Generators

Presentation generation is the task of deciding how an interface description will be mapped onto constructs of a target programming language. Each of Flick's presentation generators implements a particular mapping of AOI constructs (e.g., operations) onto target language constructs (e.g., functions). Therefore, each presentation generator is specific to a particular set of mapping rules and a particular target language (e.g., the CORBA C language mapping).

A presentation generator determines the appearance and behavior (the "programmer's contract") of the stubs and data types that present an interface — but *only* the appearance and behavior that is exposed to client or server code. The unexposed *implementation* of these stubs is determined later by a Flick back end. Therefore, the function definitions produced by a presentation generator may be implemented on top of any available transport facility, and each presentation generator is independent of any message encoding or transport. Moreover, each of Flick's presentation generators (except for the MIG generator as described previously) is independent of any particular IDL. A single presentation generator can process AOI files that were derived from several different IDLs.[3]

Flick currently has two presentation generators that read AOI files: one that implements the C mapping specified by CORBA [18] and a second that implements the C mapping defined by Sun Microsystems' `rpcgen` program [25]. Each of these presentation generators outputs its presentations in an intermediate representation called PRES_C (Presentation in C). PRES_C is a fairly complex description format containing three separate sublanguages as illustrated in Figure 2 (and described separately below): a MINT representation of the messages that will be exchanged between client and server, a CAST representation of the output C language declarations, and a set of PRES descriptions that connect pieces of the CAST definitions with corresponding pieces of the MINT structures. Of the three intermediate representations within a PRES_C file, only CAST is specific to the C language; MINT and PRES are applicable to any programming language. We plan to create intermediate representation languages for C++ and Java presentations, for example, by replacing CAST with intermediate representation languages for C++ and Java source code.

2.2.1 MINT: The Message Interface

The first step of presentation generation is to create an abstract description of all messages, both requests and replies,

[3] Naturally, the ability to process AOI files generated from different IDLs is somewhat restricted due to the limitations of particular presentations. For example, the presentation generator that implements the `rpcgen` presentation style cannot accept AOI files that use CORBA-style exceptions because there is no concept of exceptions in standard `rpcgen` presentations. Similarly, the CORBA presentation generator cannot handle self-referential type definitions that may occur in an AOI file produced from an ONC RPC IDL input because CORBA does not support self-referential types.

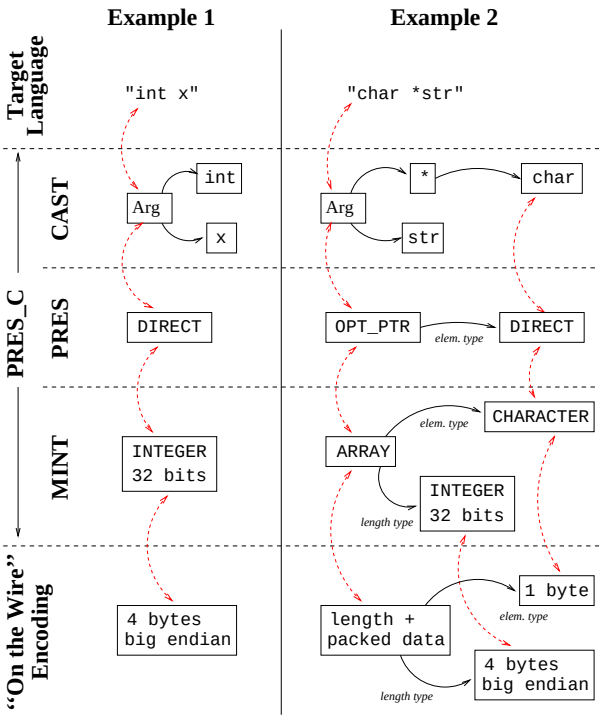

Figure 2: Two Examples of PRES_C. PRES_C is the intermediate representation that connects C target language data with "on the wire" data encodings. The first example links a C language `int` with a 4-byte, big endian encoding. The second example associates a C string (`char *`) with a counted array of packed characters.

that may be exchanged between client and server as part of an interface. These messages are represented in a type description language called MINT. A MINT representation of a data type is a directed graph (potentially cyclic) with each node representing an atomic type (e.g., an integer), an aggregate type (e.g., a fixed- or variable-length array, structure, or discriminated union), or a typed literal constant.

MINT types do not represent types in the target programming language, nor do they represent types that may be encoded within messages. Rather, MINT types represent high-level message formats, describing all aspects of an "on the wire" message *except* for low-level encoding details. MINT types serve as glue between transport encoding types and target language types as illustrated in Figure 2. The first example in Figure 2 utilizes a MINT integer type that is defined to represent signed values within a 32-bit range. The MINT integer type does not specify any particular encoding of these values, however. Target language issues are specified by the representation levels above MINT in the figure; "on the wire" data encodings are specified by the representation level below MINT. The second example in Figure 2 illustrates a MINT array type containing both a length and a vector of characters. Again, MINT specifies the ranges of the values within the type but does not specify any encoding or target language details.

2.2.2 CAST: The C Abstract Syntax Tree

The second portion of a PRES_C file is a description of the C language data types and stubs that present the interface. These constructs are described in a language called CAST, which is a straightforward, syntax-derived representation for C language declarations and statements. By keeping an explicit representation of target language constructs, Flick can make associations between CAST nodes and MINT nodes described previously. Explicit representation of target language constructs is critical to flexibility; this is the mechanism that allows different presentation generators and back ends to make fine-grain specializations to the base compiler libraries. Similarly, explicit representation is critical to optimization because Flick's back ends must have complete associations between target language data and "on the wire" data in order to produce efficient marshaling and unmarshaling code.

Although Flick's explicit representation for C language constructs is ordinary in comparison to the intermediate representations used by traditional language compilers, it is unique in comparison to traditional IDL compilers because most IDL compilers including `rpcgen` and ILU [15] maintain no explicit representations of the code that they produce.

2.2.3 PRES: The Message Presentation

PRES, the third and final component of PRES_C, defines the mapping between the message formats defined in MINT and the target language-specific, application-level formats defined in CAST. Like MINT and CAST, PRES is a graph-based description language. A node in a PRES tree describes a relationship between a MINT node and a CAST node: the data described by the MINT and CAST nodes are "connected" and marshaling and unmarshaling of data will take place as determined by the connecting PRES node. In language terms, a PRES node defines a *type conversion* between a MINT type and a target language type.

Different PRES node types describe different styles of data presentation as illustrated in Figure 2. In the first example, a MINT integer is associated with a C language integer through a direct mapping: no special data transformation is specified. In the second example, a MINT variable-length array is associated with a C pointer. The PRES node is an OPT_PTR node and specifies that a particular kind of transformation must occur for both data marshaling and unmarshaling. Consider the unmarshaling case. The OPT_PTR node defines that when the MINT array size is non-zero, the array elements will be unmarshaled and the C pointer will be set to point at the decoded array elements — in this example, characters. If the MINT array size is zero, the C pointer will be set to null. Reverse transformations occur when the C pointer

data must be marshaled into a message.

Other PRES node types define similar kinds of presentation styles, and the set of PRES node types is designed to cover all of the transformations required by existing presentation schemes. PRES is not specific to any one programming language, although certain node types depend on certain language features. For instance, OPT_PTR nodes only make sense for target languages that have pointers.

2.2.4 PRES_C: The C Presentation

PRES_C combines the intermediate representations described above to create a complete description language for C language interface presentations. A PRES_C file contains the array of stub declarations that will present the interface (to the client or server, not both). Each stub is associated with its declaration in CAST, the MINT description of the messages it receives, the MINT description of the messages it sends, and two PRES trees that associate pieces of the two MINT trees with the function's CAST declaration.

In total, a PRES_C file is a complete description of the presentation of an interface — it describes everything that a client or server must know in order to invoke or implement the operations provided by the interface. The only aspect of object invocation not described by PRES_C is the transport protocol (message format, data encoding, and communication mechanism) that will be used to transfer data between the client and the server. This final aspect of IDL compilation is the domain of Flick's back ends.

2.3 Back Ends

A Flick *back end* inputs a description of a presentation and outputs code to implement that presentation in a particular programming language. For presentations in C, the input to the back end is a PRES_C file and the output is a "`.c`" file and a corresponding "`.h`" file. The output C code implements the interface presentation for either the client or the server. Because the output of a presentation generator completely describes the appearance and exposed behavior of the stubs that implement an interface, Flick's back ends are entirely independent of the IDL and presentation rules that were employed to create a presentation.

Each back end is, however, specific to a single programming language, a particular message encoding format, and a particular transport protocol. All of the currently implemented back ends are specific to C, but Flick's "kit" architecture will support back ends specific to other languages such as C++ or Java in the future. Each of Flick's C back ends supports a different communication subsystem: the first implements the CORBA IIOP (Internet Inter-ORB Protocol) [18] on top of TCP; the second sends ONC RPC messages [23, 24] over TCP or UDP; the third supports MIG-style typed messages sent between Mach 3 ports; and the fourth implements a special message format for the fast Fluke kernel IPC facility [10]. Although these four communication subsystems are all very different, Flick's back ends share a large library of code to optimize the marshaling and unmarshaling of data. This library operates on the MINT representations of the messages. Whereas a presentation generator creates associations between MINT types and target language types (through PRES), a back end creates associations between MINT types and "on the wire" encoding types. The mapping from message data to target language is therefore a chain: from encoded type to MINT node, from MINT node to PRES node, and from PRES node to CAST. Flick's library for C back ends operates on these chains and performs optimizations that are common to all transport and encoding systems.

3 Optimization

Flick's back ends apply numerous domain-specific optimization techniques to address the performance problems that typically hinder IDL-based communication. Flick's optimizations are complementary to those that are generally implemented by traditional language compilers. While many of Flick's techniques have counterparts in traditional compilers, Flick is unique in that it has knowledge of its specialized task domain and has access to many different levels of information through its multiple intermediate representations. This allows Flick to implement optimizations that a general language compiler cannot. Conversely, Flick produces code with the expectation that general optimizations (e.g., register allocation, constant folding, and strength reduction) will be performed by the target language compiler. In summary, Flick implements optimizations that are driven by its task domain and delegates general-purpose code optimization to the target language compiler.

3.1 Efficient Memory Management

Marshal Buffer Management Before a stub can marshal a datum into its message buffer, the stub must ensure that the buffer has at least enough free space to contain the encoded representation of the datum. The stubs produced by typical IDL compilers check the amount of free buffer space before *every* atomic datum is marshaled, and if necessary, expand the message buffer. These repeated tests are wasteful, especially if the marshal buffer space must be continually expanded. The stubs produced by Flick avoid this waste.

Flick analyzes the overall storage requirements of every message that will be exchanged between client and server. This is accomplished by traversing the MINT representation of each message. The storage requirements and alignment constraints for atomic types are given by the "on the wire" data types that are associated with the various MINT nodes. The storage requirements for aggregate types are determined by working backward from nodes with known requirements.

Ultimately, Flick classifies every type into one of three storage size classes: fixed, variable and bounded, or variable and unbounded.

From this information, Flick produces optimized code to manage the marshal buffer within each stub. Before marshaling a fixed-size portion of a message, a Flick-generated stub will ensure that there is enough free buffer space to hold the *all* of the component data within the fixed-size message segment. The code that actually marshals the data is then free to assume that sufficient buffer space is available. In cases in which an entire message has a fixed size, Flick generates a stub that checks the size of the marshal buffer exactly once.[4] A different message segment may be variable in size but bounded by a limit known at stub generation time or by a limit known at stub execution time. In this case, if the range of the segment size is less than a predetermined threshold value (e.g., 8KB), Flick produces code similar to that for fixed-size message fragments: the generated stub will ensure that there is enough space to contain the maximum size of the message segment. If the range is above the threshold, however, or if the message fragment has no upper bound at all, then Flick "descends" into the message segment and considers the segment's subcomponents. Flick then analyzes the subcomponents and produces stub code to manage the largest possible fixed-size and threshold-bounded message segments as described above. Overall, our experiments with Flick-generated stubs have shown that this memory optimization technique reduces marshaling times by up to 12% for large messages containing complex structures.

Parameter Management Another optimization that requires domain knowledge is the efficient management of memory space for the parameters of client and server stubs. Just as it is wasteful to allocate marshal buffer space in small pieces, it is similarly wasteful to allocate memory for unmarshaled data on an object-by-object or field-by-field basis. Therefore, Flick optimizes the allocation and deallocation of memory used to contain the unmarshaled data that will be presented to clients and servers. For example, Flick-generated stubs may use the runtime stack to allocate space for parameter data when this is allowed by the semantics of the interface presentation. In some situations, Flick-generated stubs use space within the marshal buffer itself to hold unmarshaled data — this optimization is especially important when the encoded and target language data formats of an object are identical. Generally, these optimizations are valid only for `in` (input) parameters to the functions in a server that receive client requests. Further, the semantics of the presentation must forbid a server function from keeping a reference to a parameter's storage after the function has returned. Our experiments have shown that stack allocation is most important for relatively modest amounts of data — stack allocation for small data objects can decrease unmarshaling time by 14% — and that reuse of marshal buffer space is most important when the amount of data is large. However, the behavior of stack and marshal buffer storage means that it is suitable only in certain cases. Flick can identify these cases because it has access to the behavioral properties of the presentations that it creates.

[4] Flick-generated stubs use dynamically allocated buffers and reuse those buffers between stub invocations. This is generally preferable to allocating a new buffer for each invocation. However, it means that stubs that encode fixed-size messages larger than the minimum buffer size must verify the buffer size once.

3.2 Efficient Copying and Presentation

Data Copying By comparing the encoded representation of an array with the representation that must be presented to or by a stub, Flick determines when it is possible to copy arrays of atomic types with the C function `memcpy`. Copying an object with `memcpy` is often faster than copying the same object component-by-component, especially when the components are not the same size as machine words. For instance, our measurements show that this technique can reduce character string processing times by 60–70%. In order for this optimization to be valid, the encoded and target language data formats must be identical, and this can be determined by examining the type chains constructed by the presentation generator and back end as described in Section 2.3. A more flexible copy optimizer that allows for byte swapping and word copying of other aggregate types — similar to the optimizer in USC [19] — will be implemented in a future version of Flick.

Even when an object cannot be copied with `memcpy`, Flick performs an optimization to speed component-by-component copying. As part of the analysis performed for optimizing marshal buffer allocation described above, Flick identifies portions of the message that have fixed layouts. A message region with a fixed size and a fixed internal layout is called a *chunk*. If Flick discovers that a stub must copy data into or out of a chunk, Flick produces code to set a *chunk pointer* to the address of the chunk. Subsequent stub accesses to components of the chunk are performed by adding a constant offset to the chunk pointer. The chunk pointer itself is not modified; rather, individual statements perform pointer arithmetic to read or write data. Flick assumes that the target language compiler will turn these statements into efficient pointer-plus-offset instructions. Chunking is a kind of common subexpression elimination that would not ordinarily be performed by the target language compiler itself due to the general difficulty of optimizing pointer-based code. Chunk-based code is more efficient than code produced by traditional IDL compilers, which generally increments a read or write pointer after each atomic datum is processed. Our experiments with Flick-generated stubs show that chunking can reduce some data marshaling times by 14%.

Compiler	Size of Client		Size of Server	
	Stubs	Library	Stub	Library
Flick	2800	0	2116	0
PowerRPC	2656	2976	2992	2976
rpcgen	2824	2976	3796	2976
ILU	7148	24032	6628	24032
ORBeline	14756		16208	

Table 2: Object Code Sizes in Bytes. Each IDL compiler produced stubs for the directory interface described in Section 4 and the generated stubs were compiled for our SPARC test machines. The sizes of the compiled stubs, along with the sizes of the library code required to marshal and unmarshal data, were determined through examination of the object files. Numbers for MIG are not shown because the MIG IDL cannot express the interface. Library code for ORBeline is not shown because we had limited access to the ORBeline runtime.

Compiler	Origin	IDL	Encoding	Transport
rpcgen	Sun	ONC	XDR	ONC/TCP
PowerRPC	Netbula	~CORBA	XDR	ONC/TCP
Flick	Utah	ONC	XDR	ONC/TCP
ORBeline	Visigenic	CORBA	IIOP	TCP
ILU	Xerox PARC	CORBA	IIOP	TCP
Flick	Utah	CORBA	IIOP	TCP
MIG	CMU	MIG	Mach 3	Mach 3
Flick	Utah	ONC	Mach 3	Mach 3

Table 3: Tested IDL Compilers and Their Attributes. rpcgen, PowerRPC, and ORBeline are commercial products, while ILU and MIG are well known compilers from research organizations. The PowerRPC IDL is similar to the CORBA IDL. The target language was C, except for ORBeline which supports only C++.

Specialized Transports Because Flick is a toolkit, it is straightforward to implement back ends that take advantage of special features of a particular transport system. For example, Flick's Mach 3 back end allows stubs to communicate out-of-band data [20] and Flick's Fluke [10] back end produces stubs that communicate data between clients and servers in machine registers. A Fluke client stub stores the first several words of the message in a particular set of registers; small messages fit completely within the register set. When the client invokes the Fluke kernel to send the message, the kernel is careful to leave the registers intact as it transfers control to the receiving server. This optimization is critical for high-speed communication within many microkernel-based systems.

3.3 Efficient Control Flow

Inline Code The stubs produced by many IDL compilers are inefficient because they invoke separate functions to marshal or unmarshal each datum in a message. Those functions in turn may invoke other functions, until ultimately, functions to process atomic data are reached. This type of code is straightforward for an IDL compiler to generate. However, these chains of function calls are expensive and impose a significant runtime overhead. Not only are the function calls wasteful, but reliance on separate, type-specific marshal and unmarshal functions makes it difficult for an IDL compiler to implement memory management optimizations such as those described previously in Section 3.1. A general-purpose marshal function must always check that buffer space is available; a separate unmarshal function cannot use the runtime stack to allocate space for the unmarshaled representation of a data object. Therefore, Flick aggressively inlines both marshal and unmarshal code into both client- and server-side stubs. In general, Flick-generated stubs invoke separate marshal or unmarshal functions only when they must handle recursive types such as linked lists or unions in which one of the union branches leads back to the union type itself.[5] For a large class of interfaces, inlining actually *decreases* the sizes of the stubs once they are compiled to machine code. This effect is illustrated in Table 2. Inlining obviously removes expensive function calls from the generated code, but more importantly, it allows Flick to specialize the inlined code *in context*. The memory, parameter, and copy optimizations described previously become more powerful as more code can be inlined and specialized. In total, our experiments with Flick show that stubs with inlined code can process complex data up to 60% faster than stubs without this optimization.

Message Demultiplexing A server dispatch function must demultiplex messages received by the server process and forward those messages to the appropriate work functions. To perform this task, the dispatch function examines a discriminator value at the beginning of every message. This discriminator may be one or more integer values, a packed character string, or any other complex type, depending on the message format. Regardless of the type, Flick generates demultiplexing code that examines machine word-size chunks of the discriminator insofar as possible. The acceptable values for a discriminator word are used to produce a C `switch` statement; multi-word discriminators are decoded using nested `switch`es. When a complete discriminator has been matched, the code to unmarshal the rest of the message is then inlined into the server dispatch function.

4 Experimental Results

To evaluate the impact of Flick's optimizations, we compared Flick-generated stubs to those from five other IDL compilers, including three sold commercially. The different IDL compilers are summarized in Table 3. The first compiler, Sun's rpcgen, is in widespread use. PowerRPC [17] is a

[5] A future version of Flick will produce iterative marshal and unmarshal code for "tail-recursive" data encodings in the marshal buffer.

new commercial compiler derived from `rpcgen`. PowerRPC provides an IDL that is similar to the CORBA IDL; however, PowerRPC's back end produces stubs that are compatible with those produced by `rpcgen`. ORBeline is a CORBA IDL compiler distributed by Visigenic, implementing the standard mapping for CORBA onto C++. ILU and MIG represent opposite ends of a spectrum: ILU is a very flexible compiler that produces slow stubs, whereas MIG is a very rigid compiler that produces fast stubs.

For the ONC RPC and CORBA IDL-based compilers, we measured the performance of generated stub functions communicating across three different networks: a 10Mbps Ethernet link, a 100Mbps Ethernet link, and a 640Mbps Myrinet link [4]. For MIG interfaces, we measured Mach IPC speeds between separate tasks running on a single host.[6] For each transport and compiler, we measured the costs of three different method invocations. The first method takes an input array of integers. The second takes an input array of "rectangle" structures: each structure contains two substructures, and each substructure holds two integers (i.e., a coordinate value). The third method takes an input array of variable-size "directory entry" structures: each directory entry contains a variable-length string followed by a fixed-size, UNIX `stat`-like structure containing 136 bytes of file information (30 4-byte integers and one 16-byte character array). Although the size of a directory entry is variable, in our tests we always sent directory entries containing exactly 256 bytes of encoded data.

These methods were repeatedly invoked in order to measure both marshaling speed and end-to-end throughput for a variety of message sizes. The first two methods were invoked to send arrays ranging in size from 64 bytes to 4MB. The third method was invoked to send arrays ranging in size from 256 bytes to 512KB.

Marshal Throughput Marshal throughput is a measure of the time required for a stub to encode a message for transport, independent of other runtime overhead or the time required to actually transmit the message. To measure marshal throughput, we instrumented the stubs produced by Flick, `rpcgen`, PowerRPC, ILU, and ORBeline, and the resultant throughput measurements are shown in Figure 3. The figure shows that Flick-generated marshal code is between 2 and 5 times faster for small messages and between 5 and 17 times faster for large messages. As expected, Flick-generated stubs process integer arrays more quickly than structure arrays because Flick performs its `memcpy` optimization only for arrays of atomic types. ORBeline stubs use scatter/gather I/O in order to transmit arrays of integers and thereby avoid conventional marshaling [12]; this is why data for ORBeline's performance over integer arrays are missing from Figure 3.

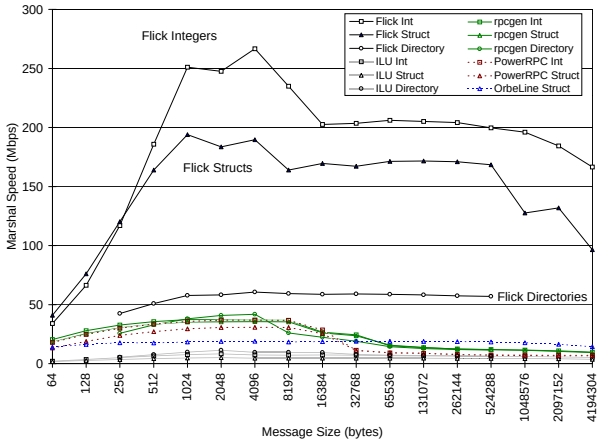

Figure 3: Marshal Throughput on a Big Endian (SPARC) Architecture. This test compares equivalent marshaling functions and avoids any transport-related bottlenecks. The performance ratios are similar when the tests are performed on a little endian (Pentium) architecture. Flick's superior throughput shows that Flick-generated stubs are suitable for use on high-performance transports.

End-to-End Throughput The gains derived from greater marshal throughput can only be realized to the extent that the operating system and network between client and server do not limit the possible end-to-end throughput of the system. To show that improved network performance will increase the impact of an optimizing IDL compiler, we measured the round-trip performance of stubs produced by the three compilers supporting ONC transports: `rpcgen`, PowerRPC, and Flick, on three different networks. The stubs produced by the three compilers all have minimal runtime overhead that is not related to marshaling, thus allowing a fair comparison of end-to-end throughput.[7] Figure 4 shows that the maximum end-to-end throughput of all the compilers' stubs is approximately 6.5–7.5Mbps when communicating across a 10Mbps Ethernet. Flick's optimizations have relatively little impact on overall throughput.

Over fast communication links, however, Flick's optimizations again become very significant. Figures 5 and 6

[6] Our hosts for the network and marshaling tests were two Sun SPARCstation 20/50 machines. Each ran at 50MHz, had 20K/16K (I/D, 5/4 set-associative) L1 caches, no L2 caches, were rated 77 on the SPECint_92 benchmark, and had measured memory copy/read/write bandwidths of 35/80/62 MBps, although the `libc bcopy` gives only 29MBps. They ran Solaris 2.5.1. One machine had 64MB DRAM, while the other had 96MB DRAM. Our host for the MIG tests was a 100MHz Pentium with an 8K/8K (I/D 2/2 assoc) L1 cache, a 512K direct-mapped L2 cache, both write-back, and 16MB of DRAM, running CMU Mach 3. It had copy/read/write bandwidths of 36/62/82MBps. All memory bandwidth tests were performed using `lmbench` 1.1 [16], and all throughput measurements were performed with the operating system socket queue size set to 64K.

[7] Unlike stubs produced by Flick, `rpcgen`, and PowerRPC, stubs generated by ORBeline and ILU include function calls to significant runtime layers. These runtime layers perform tasks that are necessary in certain environments (e.g., multi-thread synchronization) but which are not required for basic client/server communication.

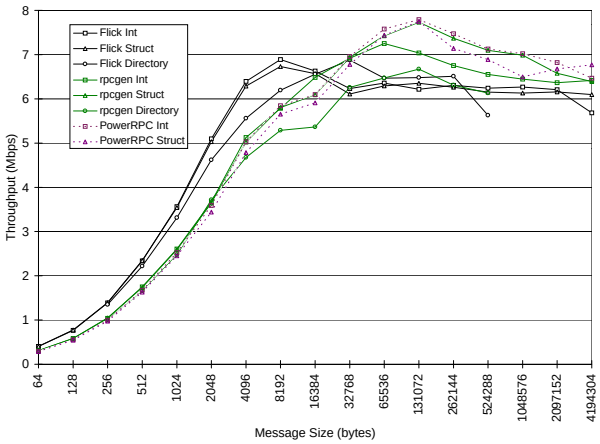

Figure 4: End-to-End Throughput Across 10Mbps Ethernet. The data for PowerRPC and `rpcgen` is incomplete because the generated stubs signal an error when invoked to marshal large arrays of integers.

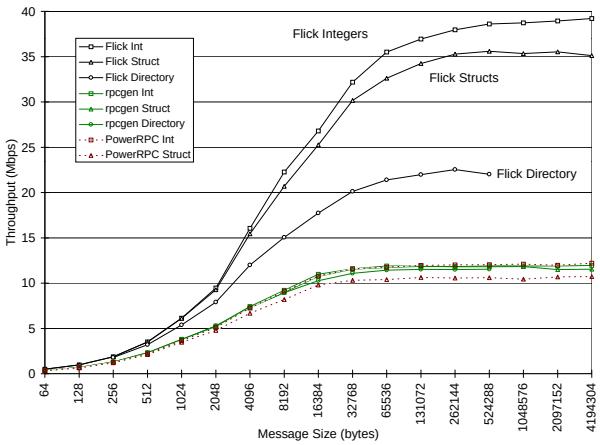

Figure 5: End-to-End Throughput Across 100Mbps Ethernet.

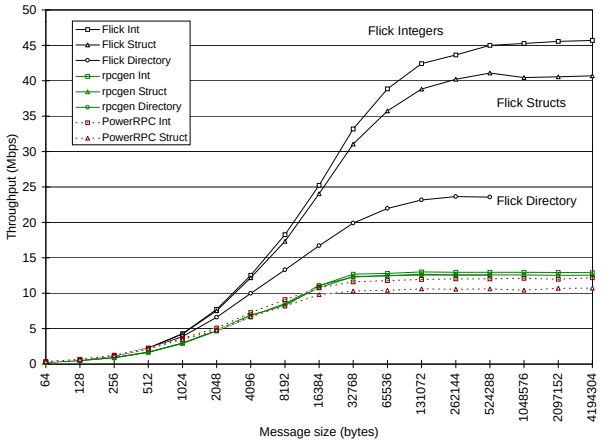

Figure 6: End-to-End Throughput Across 640Mbps Myrinet.

show that for stubs communicating across 100Mbps Ethernet and 640Mbps Myrinet, Flick's optimizations increase end-to-end throughput by factors of 2–3 for medium size messages, factors of 3.2 for large Ethernet messages, and factors of 3.7 for large Myrinet messages. With Flick stubs, both 100Mbps and 640Mbps transports yield significant throughput increases. In contrast, PowerRPC and `rpcgen` stubs did not benefit from the faster Myrinet link: their throughput was essentially unchanged across the two fast networks. This indicates that the bottleneck for PowerRPC and `rpcgen` stubs is poor marshaling and unmarshaling behavior.

Measurements show that the principal bottlenecks for Flick stubs are the memory bandwidth of the SPARC test machines and the operating system's communication protocol stack. Flick's maximum throughput is less than half of the theoretical Ethernet bandwidth and less than 10% of the theoretical Myrinet bandwidth. These results, however, must be viewed in terms of the *effective* bandwidth that is available after memory and operating system overheads are imposed. As measured by the widely available `ttcp` benchmark program, the maximum effective bandwidth of our 100Mbps Ethernet link is 70Mbps and the maximum bandwidth of our Myrinet link is just 84.5Mbps. These low measurements are due to the performance limitations imposed by the operating system's low-level protocol layers [12]. Through calculations based on these numbers and measured memory bandwidth, we have confirmed that the difference between `ttcp` throughput and the performance of Flick stubs is entirely due to the functional requirement to marshal and unmarshal message data — which requires memory-to-memory copying and is thus limited by memory bandwidth. As operating system limitations are reduced by lighter-weight transports [6, 7], Flick's ability to produce optimized marshal code will have an increasingly large impact.

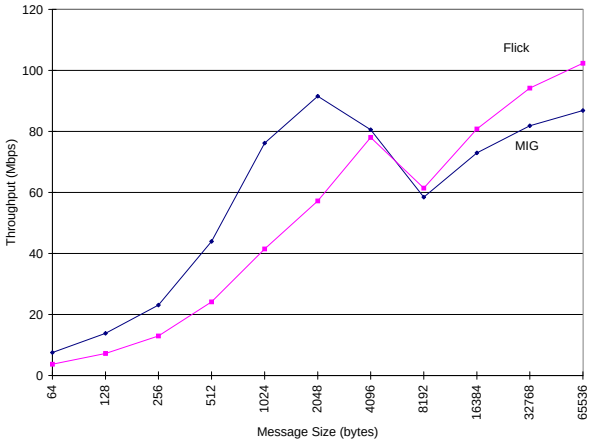

Figure 7: End-to-End Throughput for MIG and Flick Stubs. MIG is both highly specialized for optimizing Mach message communication, and is able only to support simple data types. At larger-sized messages, Flick-generated stubs achieve throughput comparable to that of MIG-generated stubs even though Flick is a much more flexible IDL compiler.

End-to-End Throughput Compared to MIG In Figure 7 we compare the end-to-end throughput of Flick-generated stubs to the throughput of stubs generated by MIG, Mach 3's native IDL compiler. In this experiment the stubs transmit arrays of integers; we did not generate stubs to transmit arrays of structures because MIG cannot express arrays of non-atomic types. MIG is a highly restrictive IDL compiler, but it is also highly specialized for the Mach 3 message communication facility. The effect of this specialization is that for small messages, MIG-generated stubs have throughput that is twice that of the corresponding Flick stubs. However, as the message size increases, Flick-generated stubs do increasingly well against MIG stubs. Beginning with 8K messages, Flick's stubs increasingly outperform MIG's stubs, showing 17% improvement at 64K. The results of this experiment demonstrate the potential for further improvements in Flick and are encouraging because they show that although Flick is much more flexible and general-purpose than MIG, Flick-generated stubs can compete against stubs produced by the operating system's own IDL compiler. At a current cost for small and moderate sized messages, Flick allows Mach programmers to use modern IDLs such as CORBA and supports many C language presentations (e.g., structures) that MIG cannot offer.

5 Related Work

Previous work has shown that flexible, optimizing compilers are required in order to eliminate the crippling communication overheads that are incurred by many distributed systems. In a seminal paper in the networking domain, Clark and Tennenhouse [5] identified *data representation conversion* as a bottleneck to many communication protocols. They emphasized the importance of optimizing the presentation layer of a protocol stack and showed that it often dominates processing time. Recent work by Schmidt et al. [12, 21] has quantified this problem 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. Mach's MIG [20] compiler generates fast code but only by restricting the types that it can handle: essentially just scalars and arrays of scalars. Hoschka and Huitema [14] studied the tradeoffs between (large, fast) compiled stubs and (small, slow) interpreted stubs and suggested that an optimizing IDL compiler should use both techniques in order to balance the competing demands of throughput and stub code size. However, their experimental results appear to apply only to the extraordinarily expensive type representations used in ASN.1, in which type encoding is dynamic even for fundamental scalar types such as integers. Of more relevance to commonly used representations is the Universal Stub Compiler (USC) work by O'Malley et al. [19]. USC does an excellent job of optimizing copying based on a user-provided specification of the byte-level representations of data types. This work is complementary to ours: as the authors state, USC may be used alone to specify simple conversion functions (e.g., for network packet headers) or it may be leveraged by a higher-level IDL compiler. By incorporating USC-style representations for all types, Flick could improve its existing copy optimizations as outlined in Section 3.2.

Recently, Gokhale and Schmidt [13] addressed performance issues by optimizing SunSoft's reference implementation of IIOP [26]. The SunSoft IIOP implementation does not include an IDL compiler but instead relies on an interpreter to marshal and unmarshal data. The authors optimized the interpreter and thereby increased throughput over an ATM network by factors of 1.8 to 5 for a range of data types. Their implementation achieved throughput comparable to that of commercial CORBA systems that utilize compiled stubs, including ORBeline [11]. However, since Flick-generated stubs typically greatly outperform stubs produced by ORBeline, Flick must also outperform the best current interpretive marshalers.

In the area of flexible IDL compilers, the Inter-Language Unification [15] (ILU) system from Xerox PARC emphasizes support for many target languages, supporting C, C++, Modula-3, Python, and Common Lisp. However, like most IDL compilers, ILU uses as its sole intermediate representation a simple AST directly derived from the IDL input file. ILU does not attempt to do any optimization but merely traverses the AST, emitting marshal statements for each datum, which are typically (expensive) calls to type-specific mar-

shaling functions. Each separate backend is essentially a full copy of another with only the `printfs` changed. For Flick to do similarly, it would simply emit marshaling code as it traversed an AOI structure. ILU does support two IDLs — its native, unique IDL and the CORBA IDL — but only by translating the CORBA language into its own IDL.

Like Flick, the Concert/C distributed programming system [2] quite fully develops the concept of flexible presentation. In Concert, the primary purpose of this separation is to handle the vagaries of RPC mapping to different target languages, striving for a "minimal contract" in order to achieve maximal interoperability between target languages. However, this separation is not leveraged for optimizations. In earlier work [8, 9] 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.

Finally, several techniques used by Flick are similar or analogous to those in traditional compilers for general purpose programming languages. In addition, it appears that our work has many similarities to type-based representation analysis [22] directed to achieving more efficient "unboxed" data representations whenever possible, and to convert between such representations.

6 Conclusion

This work exploits the fundamental and overdue recognition that interface definition languages are indeed programming languages, albeit specialized and non-traditional in their computational content. This insight is the basis for Flick, a novel, modular, and flexible IDL compiler that approaches stub generation as a programming language translation problem. This, in turn, allows established optimizing compiler technology to be applied and extended in domain-specific ways.

Flick exploits many fundamental concepts of modern compiler organization including carefully designed intermediate representations, modularized front and back ends localizing source and target language specifics, and a framework organization that encourages reuse of software implementing common abstractions and functionality. Our quantitative experimental results confirm that this approach is indeed effective for producing high-performance stubs for a wide variety of communication infrastructures.

Availability

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

Acknowledgments

We are especially indebted to Steve Clawson, who provided technical support for our Mach and Myrinet performance measurements. Chris Alfeld, Godmar Back, John Carter, Ajay Chitturi, Steve Clawson, Mike Hibler, Roland McGrath, Steve Smalley, Jeff Turner, and Kevin Van Maren all reviewed drafts of this paper and suggested numerous improvements; we wish we could have incorporated them all. Nathan Dykman and Gary Crum did much early implementation. We thank Gary Barbour for loaning us the Suns, Al Davis and Chuck Seitz for loaning the Myrinet cards, and Grant Weiler for help with Sun system software.

References

[1] ACCETTA, M., BARON, R., BOLOSKY, W., GOLUB, D., RASHID, R., TEVANIAN, A., AND YOUNG, M. Mach: A new kernel foundation for UNIX development. In *Proc. of the Summer 1986 USENIX Conf.* (June 1986), pp. 93–112.

[2] AUERBACH, J. S., AND RUSSELL, J. R. The Concert signature representation: IDL as an intermediate language. In *Proc. of the Workshop on Interface Definition Languages* (Jan. 1994), pp. 1–12.

[3] BIRRELL, A. D., AND NELSON, B. J. Implementing remote procedure calls. *ACM Transactions on Computer Systems 2*, 1 (Feb. 1984).

[4] BODEN, N., COHEN, D., FELDERMAN, R., KULAWIK, A., SEITZ, C., SEIZOVIC, J., AND SU, W.-K. Myrinet – A gigabit-per-second local-area network. *IEEE MICRO 15*, 1 (February 1995), 29–36.

[5] CLARK, D. D., AND TENNENHOUSE, D. L. Architectural considerations for a new generation of protocols. In *Proc. of the SIGCOMM '90 Symp.* (1990), pp. 200–208.

[6] DRUSCHEL, P., DAVIE, B. S., AND PETERSON, L. L. Experiences with a high-speed network adapter: A software perspective. In *Proc. of the SIGCOMM '94 Symp.* (1994), pp. 2–13.

[7] EICKEN, T. V., CULLER, D. E., GOLDSTEIN, S. C., AND SCHAUSER, K. E. Active messages: A mechanism for integrated communication and computation. In *Proc. of the 19th International Symp. on Computer Architecture* (May 1992), pp. 256–266.

[8] FORD, B., HIBLER, M., AND LEPREAU, J. Using annotated interface definitions to optimize RPC. In *Proc. of the 15th ACM Symp. on Operating Systems Principles* (1995), p. 232. Poster.

[9] FORD, B., HIBLER, M., AND LEPREAU, J. Using annotated interface definitions to optimize RPC. Tech. Rep. UUCS-95-014, University of Utah, Mar. 1995.

[10] FORD, B., HIBLER, M., LEPREAU, J., TULLMANN, P., BACK, G., AND CLAWSON, S. Microkernels meet recursive virtual machines. In *Proc. of the Second Symp. on Operating Systems Design and Implementation* (Seattle, WA, Oct. 1996), USENIX Assoc., pp. 137–151.

[11] GOKHALE, A. Personal communication, Mar. 1997.

[12] GOKHALE, A., AND SCHMIDT, D. C. Measuring the performance of communication middleware on high-speed networks. *Computer Communication Review 26*, 4 (Oct. 1996).

[13] GOKHALE, A., AND SCHMIDT, D. C. Optimizing the performance of the CORBA Internet Inter-ORB Protocol over ATM. Tech. Rep. WUCS–97–09, Washington University Department of Computer Science, St. Louis, MO, 1997.

[14] HOSCHKA, P., AND HUITEMA, C. Automatic generation of optimized code for marshalling routines. In *International Working Conference on Upper Layer Protocols, Architectures and Applications* (Barcelona, Spain, 1994), M. Medina and N. Borenstein, Eds., IFIP TC6/WG6.5, North-Holland, pp. 131–146.

[15] JANSSEN, B., AND SPREITZER, M. *ILU 2.0alpha8 Reference Manual*. Xerox Corporation, May 1996. `ftp://ftp.parc.xerox.com/pub/ilu/ilu.html`.

[16] MCVOY, L., AND STAELIN, C. lmbench: Portable tools for performance analysis. In *Proc. of 1996 USENIX Conf.* (Jan. 1996).

[17] NETBULA, LLC. PowerRPC, Version 1.0, 1996. `http://www.netbula.com/products/powerrpc/`.

[18] OBJECT MANAGEMENT GROUP. *The Common Object Request Broker: Architecture and Specification*, 2.0 ed., July 1995.

[19] O'MALLEY, S., PROEBSTING, T. A., AND MONTZ, A. B. USC: A universal stub compiler. In *Proceedings of the Conference on Communications Architectures, Protocols and Applications (SIGCOMM)* (London, UK, Aug. 1994), pp. 295–306.

[20] OPEN SOFTWARE FOUNDATION AND CARNEGIE MELLON UNIVERSITY. *Mach 3 Server Writer's Guide*. Cambridge, MA, Jan. 1992.

[21] SCHMIDT, D. C., HARRISON, T., AND AL-SHAER, E. 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.

[22] SHAO, Z., AND APPEL, A. A type-based compiler for standard ML. In *Proc. ACM SIGPLAN Symp. on Programming Language Design and Implementation* (June 1995), pp. 116–129.

[23] SRINIVASAN, R. RPC: Remote procedure call protocol specification version 2. Tech. Rep. RFC 1831, Sun Microsystems, Inc., Aug. 1995.

[24] SRINIVASAN, R. XDR: External data representation standard. Tech. Rep. RFC 1832, Sun Microsystems, Inc., Aug. 1995.

[25] SUN MICROSYSTEMS, INC. *ONC+ Developer's Guide*, Nov. 1995.

[26] SUNSOFT, INC. SunSoft Inter-ORB Engine, Release 1.1, June 1995. `ftp://ftp.omg.org/pub/interop/iiop.tar.Z`.