Slide 0

Flick: A Flexible, Optimizing IDL Compiler

Gary Crum
Kevin Frei
Nathan Dykman
Bryan Ford
Eric Eide
Jay Lepreau

October 11, 1996

Slide 1

The Problem

Existing IDL compilers are not suitable for use in high-performance or specialized systems.

- Inflexible presentations.
- Assume "heavy" transport.
- Non-optimizing.
- Output not tweakable.

IDL (interface definition language) compilers generate "stubs" which provide separate communicating processes with the abstraction of local object invocation or procedure call.

Traditional IDL compilers have two problems when applied to emerging high-speed network technologies and same-machine component-based systems.

- Typical IDL compilers do little optimization of their generated code. They assume that interprocess communication (IPC) is the bottleneck, and that therefore it is not important to optimize the handling of data within the communicating processes.

- They are "rigid" in that they support only one IDL, implement a single, fixed mapping onto each supported target language, and produce code for a narrow range of transport mechanisms.

Further, it isn't possible to "tweak" traditional IDL compilers to produce specialized code such as security access vector checks (e.g., for access control to non-kernel objects implemented by a server).

Slide 2

The Solution

Flick is an IDL compiler for performance-critical and specialized applications.

- *Optimizing.*

  Buffer management, inline code, smart data copying.

- *Flexible.*

  Multiple IDL's, multiple transports, multiple presentations.

- *Flexibility supports Optimization.*

  Application-specific presentations.

Flick is our solution, and will be the IDL compiler used in Flux. Flick implements many optimizations including:

- Careful marshal/unmarshal buffer management (*globs* and *chunks*).
- Hashed message discrimination (with encoded discriminators).
- Inline marshal/unmarshal code (to avoid overhead from function calls).
- Smart data copying (using '`memcpy`' when possible and prudent).
- Speculative code generation.
- '`auto`' allocation for "in" parameters on server side.
- Other techniques (many not yet complete).

Moreover, Flick is extremely flexible. It supports multiple IDL's, multiple transports, and multiple presentations. Flick is designed to support multiple target languages, but currently, only C is implemented.

Flexibility means that the same interface can be presented differently to different applications. This allows for application-specific optimization. Example: An application might want to allocate the buffer for an "out" parameter.

Slide 3

Flick: IDL Compiler Kit

Flick is an IDL compiler kit:

- Separable compilation phases implemented by separate components (programs).
- Each component is specific to some aspects of compilation and independent of others.
- Extensive code sharing.
- Mix-and-match flexibility.

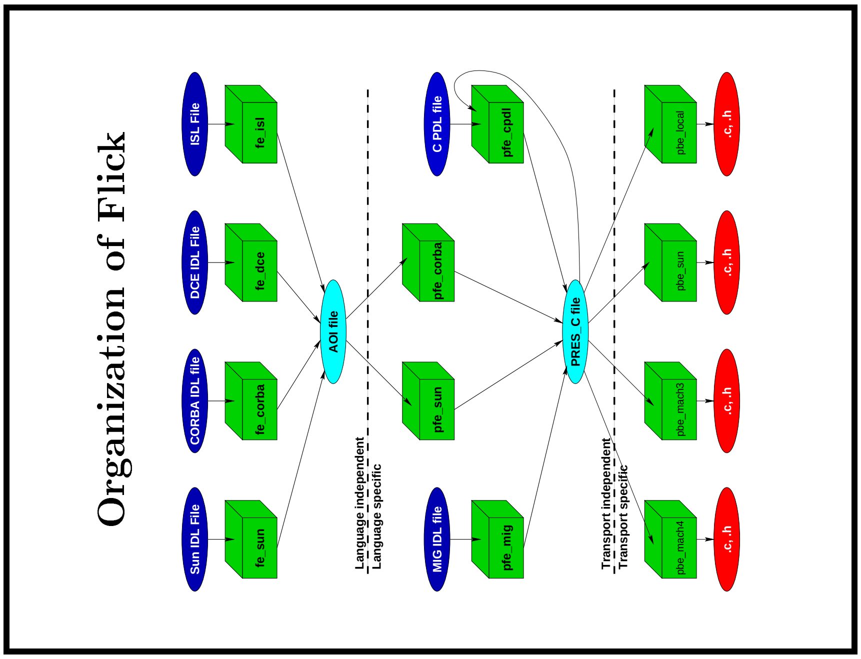
Flick is really a kit for writing IDL compilers. Flick is a collection of programs and libraries that allow us and other researchers to investigate efficient, novel, and/or specialized communication techniques.

Flick is very much like Stanford's SUIF toolkit. Whereas SUIF is targeted toward general compiler research, Flick is designed specifically for IDL compiler research.

Like SUIF, Flick is implemented as a collection of communicating programs. These programs:

- implement different and separable stages of IDL compilation;
- are "specializations" of shared library codes; and
- allow for mix-and-match arrangement of IDL, presentation, encoding, and transport. At each stage, you can pick the program that implements the particular IDL, presentation, or transport that you want your application to use.

Slide 4



Flick divides IDL compilation into three stages:

- A *front end* translates an IDL file into a semantic representation in AOI (Abstract Object Interface).
- A *presentation generator* translates the AOI description into a PRES_C file, which describes the presentation of the interface in a particular target programming language (e.g., C).
- A *back end* translates the PRES_C representation into actual code (e.g., a "`.c`" file and a "`.h`" file).

Each stage of the compiler is specific to some aspects of the compilation process and independent of other aspects:

Front end:	Specific to: Independent of:	IDL target lang., pres., encoding, transp.
Presentation generator:	Specific to: Independent of:	target language, presentation. IDL (ideally), encoding, transport.
Back end:	Specific to: Independent of:	target language, encoding, transport. IDL, presentation.

Slide 5

Code Reuse in Flick

Component	Lines	
Presentation Generator		
Libraries (`c/pfe/lib` + ...)	4535	
CORBA (`c/pfe/corba`)	459	10.1%
ONC RPC (`c/pfe/sun`)	241	5.3%
Back End		
Libraries (`c/pbe/lib` + ...)	6270	
XDR on TCP/UDP (`c/pbe/sun`)	373	5.9%
MIG on Mach (`c/pbe/mach3mig`)	464	7.4%

Flick is committed to maximal code sharing. At every stage there is a library of "general" code. Every program that implements a particular stage simply augments and/or overrides functions within the library as necessary in order to support the peculiarities of a specific IDL, presentation, encoding, or transport.

Code sharing in Flick has been highly successful, as shown in the table above. The individual presentation generators and back ends contain only small fractions of the amount of code that the shared libraries contain.

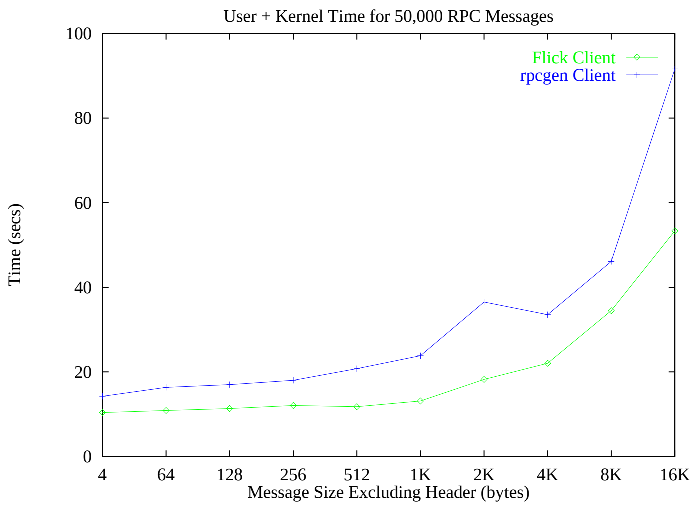
The '...' indicators in the table above include:

Library	Lines
AOI (`libaoi`)	567
MINT (`libmint`)	389
CAST (`c/libcast`)	1282
PRES_C (`c/libpres_c`)	558
Utilities (`libcompiler`)	611

The approximate line counts exclude comment lines, blank lines, and lines containing only braces.

Slide 6

Flick versus rpcgen

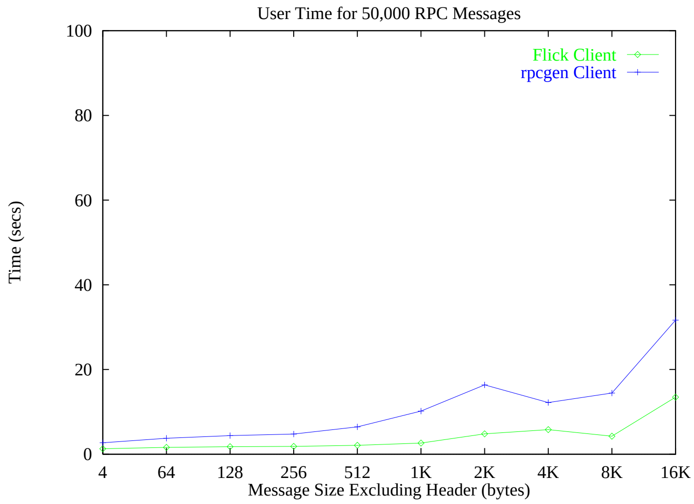


Each test was run 50,000 times and timed on `drip`, a PPC604-132MHz system using the `timex` program

Message Size	Flick		rpcgen	
	user	kernel	user	kernel
4 bytes	1.30	9.10	2.70	11.55
64 bytes	1.62	9.27	3.77	12.59
128 bytes	1.79	9.56	4.40	12.61
256 bytes	1.85	10.22	4.78	13.23
512 bytes	2.10	9.69	6.47	14.31
1KB	2.63	10.49	10.20	13.62
2KB	4.82	13.40	16.38	20.13
4KB	5.81	16.23	12.21	21.33
8KB	4.27	30.21	14.44	31.68
16KB	13.49	39.85	31.66	59.94

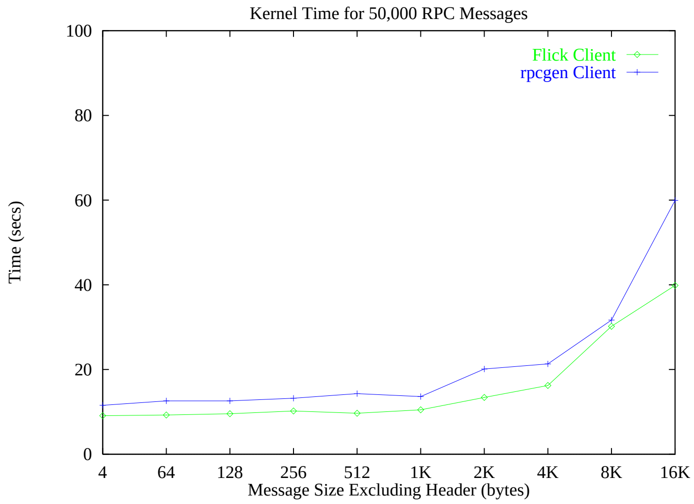
Slide 7

Flick versus rpcgen



Slide 8

Flick versus rpcgen



Slide 9

USC

USC is the Universal Stub Compiler from the University of Arizona.

- Generates fast C data marshaling functions.
- Based on byte-level descriptions of data.

```
typedef (4,4,<0..3>) long;
```

- Not a complete IDL compiler.
  - Doesn't determine on-the-wire format for you.
  - Doesn't generate communication code.

USC from the University of Arizona generates very fast C functions for marshaling and unmarshaling data structures. USC marshaling routines are often 10 to 20 times as fast as those generated by `rpcgen`.

USC's stub language is essentially C with annotations that describe the physical layouts of data in memory. Given a set of annotated structures, USC can produce C functions that efficiently convert data from one format to another.

USC is not a complete IDL compiler, however:

- There is no IDL.
- It doesn't produce communication stubs, only marshal and unmarshal stubs.
- USC can't determine the on-the-wire encoding for a particular structure.

Slide 10

Flick and USC

- Flick has the right IDL compiler infrastructure.
  - MINT connects target language types and marshaled types.
- USC has excellent byte-level reasoning.
- Put USC smarts in Flick back end library.
- Still need "external" knowledge from the target language compiler.

Flick and USC are highly complementary. It would be useful to incorporate USC's algorithm into Flick's type system. Flick currently matches target language types with on-the-wire encoding in the following way:

- A MINT tree represents the abstract format of a message.
- A presentation generator associates typed data objects in the target language presentation with nodes of the MINT tree.
- A back end associates on-the-wire encodings with nodes of the MINT tree.

If the chain of associations were extended at both ends with USC byte-level type information, then Flick could implement additional optimizations. For example, when the byte-level signatures are compatible, Flick could produce code to:

- copy the data as words rather than halfwords or bytes;
- copy the data with '`memcpy`' (currently implemented ad hoc);
- simply pass a pointer to the data in the marshal buffer; or
- byte-swap within the marshal buffer and then pass a pointer.

Putting USC-style login in Flick's C back end library would make that logic available to all C presentations.

Slide 11

Flick and SUIF

Integrating Flick with SUIF has long-term promise:

- Flick back end producing SUIF.
  - Knowledge of compiler's data formats.
  - Leverage SUIF code optimizers.
- Redesign all Flick stages around SUIF?
- Optimization based on application data flow, etc.
- "Microkernel language" implemented via SUIF annotations.
- Could still escape from SUIF via C back end.

Flick and SUIF also appear to be complementary — however, investigations in this area are very preliminary. Integrating Flick with an optimizing traditional language compiler like SUIF would open many promising avenues:

- By producing SUIF at the back end, Flick could leverage SUIF's other optimizing modules (i.e., in order to further optimize Flick-produced stubs).
- New optimizations become possible. For example, Flick could use SUIF's data flow analysis tools in order to "premarshal" objects, decide to cache objects at runtime, presize buffers, choose between compiled or interpreted stubs, etc.
- We would investigate whether SUIF is suitable as an intermediate representation for earlier Flick stages.

SUIF has a C back end, so it would be possible to use and then "escape" from SUIF if necessary (at least for C presentations).

Slide 12

Status

- Alpha version released August 12, 1996.
- Front ends for CORBA, ONC RPC, and MIG IDL's.
- Presentation generators for CORBA and ONC RPC, C language.
- C PDL presentation modifier.
- Back ends for XDR over TCP/UDP, MIG over Mach 3 ports, and Mach 4.

We made an alpha version of Flick available to five external testers on August 12, 1996. The list of testers included people at TIS, U.C. Davis, the Helsinki University of Technology, and Secure Computing Corporation. The testers helped us fix a number of bugs and solve portability issues.

Slide 13

Immediate Future

- Greatly improve optimization.
  - Incorporate USC code and/or techniques.
- Produce accurate performance measurements.
- Implement a CORBA IIOP back end.
- Implement a Fluke IPC back end.
- Coordinate with Flask efforts.
- Make a new release.

The greatest immediate need is to improve the optimization of the code output by the back ends. All or most of this work is occurring is the back end library, so it is applicable to all IDL's and transports. The greatest gain will come from adding USC-style logic to the back end.

We also need new back ends: IIOP so that we may compare ourselves to commercial CORBA compilers, and Fluke so that we can stop writing Fluke stubs by hand!

Coordination with Flask requirements is now getting started. Some preliminary ideas include:

- Adding security-related annotations to Flick's PDL:
  - to produce stubs that control access to high-level objects;
  - to describe high-level security requirements that may be implemented differently in differing communication systems;
- Demonstrating "secure CORBA" quickly by porting Flick to DTOS.

Slide 14

Post-Immediate Future

- Provide other target languages (C++, Java, "microkernel language").
- Implement security features required by Flask.
- Pursue integration with SUIF:
  - Implement a SUIF back end, *or*
  - Retool Flick programs as SUIF compiler passes.
- Implement more high-level optimizations (e.g., driven by SUIF analysis of application code).

Slide 15

Conclusion

Everything ultimately comes down to communication.