Stub-Code Performance is Becoming Important

Andreas Haeberlen Jochen Liedtke Yoonho Park Lars Reuther Volkmar Uhlig

University of Karlsruhe
System Architecture Group
76128 Karlsruhe, Germany
{haeberlen,liedtke,uhlig}@ira.uka.de

IBM T. J. Watson
Hawthorne, NY 10532
yoonho@us.ibm.com

Dresden University of Technology
Department of Computer Science
01062 Dresden, Germany
reuther@inf.tu-dresden.de

Abstract

As IPC mechanisms become faster, stub-code efficiency becomes a performance issue for local client/server RPCs and inter-component communication. Inefficient and unnecessary complex marshalling code can almost double communication costs. We have developed an experimental new IDL compiler called IDL^4 that produces near-optimal stub-code for gcc *and the L4 microkernel. IDL^4 is neither portable nor adaptable but generates in most cases stub-code that is 3 times shorter (and faster) than the code generated by a commonly used portable IDL compiler.*

1 Motivation

Multi-server and component-based systems are promising architectural approaches to handle the ever increasing complexity of operating and application systems. Components or servers (and clients) communicate with each other through cross-domain method invocations. Such interface method invocations – if crossing protection boundaries – are typically implemented through interprocess communication (IPC) mechanisms offered by a microkernel.

Component interaction in such systems has to be both convenient in use and highly efficient in execution. For over a decade, performance-oriented research focused on microkernel construction, in particular IPC performance, finally resulting in acceptable IPC overheads (100–200 cycles) [4, 1]. The second property, convenience in use, leads to the development of interface definition languages (IDLs) such as Corba IDL [5] and according IDL compilers. From interface procedure/method definitions, such compilers generate stub-code that marshals parameters on the client side, communicates through IPC or RPC kernel primitives with the server, unmarshals the parameters on the server side, invokes the corresponding server procedure/method, etc. As a result, the programmer can specify and use remote interfaces as easily as internal ones.

So far, IDL compiler research focused more on generating code in a portable and adaptable way than on producing efficient stubs. In fact, stub-code performance was negligible for early microkernels that required multiple thousands of cycles per IPC. However, with high performance IPC, stub-code efficiency becomes an issue.

For example, when using the Flick IDL compiler [2] for the SawMill Linux file system [3], we found that the generated user-level stub-code consumed about 340 instructions (approximately 170 cycles) per read request. When reading a 4K block from the file system, the stub-code adds an overhead of about 7%. For an industrial system, such overheads can no longer be ignored.

A hand coding of the according stub resulted in 80 instructions, i.e. estimated costs of 40 cycles. We think that a potential reduction from 7% to 2% stub overhead justifies an experiment to find out whether near-optimal stub-code can be generated at reasonable costs.

This paper describes the resulting IDL^4 compiler that generates code for *gcc* on x86 and the L4 microkernel. The current IDL^4 is a prototype that purely focuses on generating efficient code. Portability and adaptability are ignored and remain a topic for future work.

Structure of the paper

This paper reports on progress that has been made with IDL^4, an experimental IDL compiler for the L4 microkernel. Section 2 sketches prerequisites useful for understanding the subsequent discussion, e.g. IDL syntax, the L4 IPC mechanism, and our experiences using the Flick IDL compiler in the SawMill project. Section 3 describes the stub-code model that was designed for the IDL^4 compiler, and Section 4 illustrates how the compiler works. Finally, Section 5 reports on the achieved stub-code quality.

2 Prerequisites

2.1 L4/x86 IPC

L4's basic communication paradigm [4] is synchronous IPC. Typical operations are *send, receive, call* (atomic send&receive), and atomic *reply&wait.* Rich message types help to improve end-to-end IPC performance.

Register messages consist of a small number of 32-bit words that are sent and received in general purpose registers. On the x86 platform, up to 3 words (plus sender id and message descriptor) can be transferred as a register message. As there is no need for copy operations to cross address space boundaries, register messages have the lowest IPC costs, e.g. 180 cycles on a Pentium III 450 MHz.

Memory messages can be used to copy longer messages from the sender's address space to the receiver. Message size can be up to 2MB; however, this mechanism is slower than the first one because it involves copying to/from memory, and the kernel might have to establish a temporary mapping to make both address spaces available at the same time.

Indirect strings save unnecessary copy operations to/from the message buffer. Up to 31 strings can be included in a memory message. On the receiver side, buffers for each strings can be specified so that the IPC can copy directly from the server object to the client object or vice versa. Scatter/gather permits strings to be gathered on the sender side and/or scattered on the receiver side. Thus, multiple blocks can be directly transferred to a single receive buffer; a single send buffer can be split into multiple blocks. Figure 1 illustrates how a complex memory message is transferred.

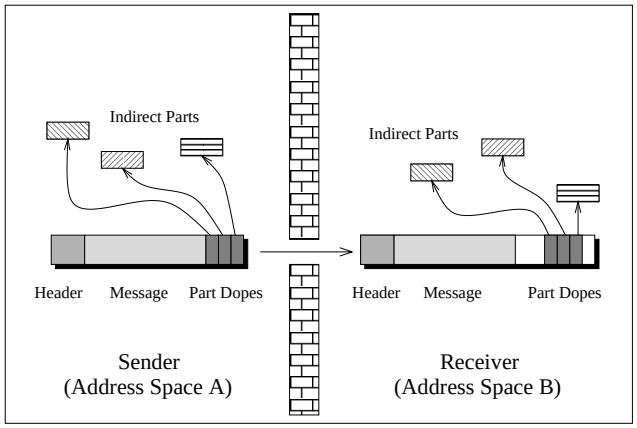

Figure 1. Complex memory message including indirect strings.

Map messages map pages or larger parts of the sender's address space into the receiver's space. This feature enables user-level pagers and main memory management on top of the microkernel. Special communication mechanisms based on shared regions can also be constructed.

2.2 The SawMill Project

The SawMill project is aimed at addressing the complexity of developing and maintaining a variety of custom operating systems. With the emergence of embedded and personal systems, the need to create operating system customized to device and application requirements has increased significantly. The development and maintenance of these operating systems is quite unwieldy. First, the SawMill project is developing an approach and tools to decompose existing operating systems into flexibly reusable components. The next step is to define an architecture upon which efficient and robust operating systems can be composed. This framework is being applied to Linux to create SawMill Linux. SawMill Linux consists of Linux-based components that provide typical system services – such as file systems and network systems – and general components – such as memory, task, device, and access control managers – which enable the composition of a coherent Linux system.

2.3 Flick

IDL compilers such as Flick [2] are relatively easy to port to a new OS or middleware kernel and permit extensions through new data types. The output of an IDL compiler is typically used as input for the general-purpose compiler such as *gcc* that the programmer uses for code development. Easy adaptation of the IDL compiler to new general purpose compilers is a further relevant property.

Flick tries to generate efficient stub-code by using inline functions and macros for the generated stubs whenever possible. Nevertheless, at least when combined within *gcc*, this results in huge amounts of data transfer operations that are logically superfluous. In theory, a compiler should be able to remove all of them. In practice, the required data flow analysis is too complicated so that inefficient code is generated.

3 Designing a Stub Model

3.1 A Simple Stub Model

We first describe a simple stub model to illustrate which tasks have to be done by stub-code on the client and on the server side. For this simple model, we assume that a client invokes a procedure or method M that is supplied by the server. Synchronization and concurrency are ignored in the simple model. M has *in* parameters (values passed from the client to the server), *out* parameters (result values passed from the server back to the client), and *inout* parameters that

are first passed to the server and then overwritten by results coming back from the server.

The IDL compiler generates a client stub procedure M_{client} for each function M in the interface definition. The client stub is called locally by the client application. It hides the fact that the service function does not run locally, but rather in another address space or even on another computer thousands of miles away. The client stub assembles a message with all the information the server requires to complete the task, including all the parameters (*marshalling*).

The message is then sent to the server, and the client waits for the server to reply. The reply message contains all out and inout result values. The client stub unpacks those values from the message and stores them in the appropriate client parameters (*unmarshalling*). In detail, the client stub works as follows:

C1) M_{client} constructs a *request* message that contains all input and inout parameters and a *key* that identifies the procedure/method M (*marshalling*);

C2) sends the request message to the server that implements M and waits for a reply message from the server;

C3) fills the inout and out parameters with data received through the reply message (*unmarshalling*); and

C4) returns to the invoker.

The server programmer implements a procedure M_{server} on the server side for each method M of the interface definition.

The IDL compiler generates a central code pattern that handles communication, decoding, marshalling/unmarshalling of parameters, etc. This central server code typically includes a main loop that receives requests from clients and distributes them to the corresponding server procedures M_{server}. For each M_{server}, the IDL compiler generates a server stub that examines the request packet and retrieves the input data (*unmarshalling*). The stub then invokes the routine itself and creates a reply for the client.

An IDL compiler should generate both the main loop and the stubs automatically. Users should be able to easily modify the loop code, because they might want to implement additional features such as load balancing. In detail, a thread that waits for client requests

S0) receives the request message and uses the received key to determine which procedure/method M should be invoked and which parameters are expected and will be returned by M;

S1) extracts in and inout parameters from the received request message (*unmarshalling*);

S2) calls the server procedure M_{server} with the extracted parameter values;

S3) constructs a *reply* message and stores the result values of all inout and output parameters of procedure M_{server} in that message (*marshalling*);

S4) sends the reply message back to the client.

Steps C2, S0, and S4 are basically determined by the underlying IPC system, in our case by the L4 microkernel. Steps C4 and S2 are given by the used general-purpose compiler, in our case *gcc*. Marshalling and unmarshalling, steps C1+S1 and S3+C3, are less restricted and more crucial. As our experience with Flick shows, a less optimal model could easily result in significant copy overhead for marshalling and unmarshalling.

3.2 Marshalling Through Direct Stack Transfer

To get an idea of how parameters can be communicated most efficiently between M_{client} and M_{server}, we first look at a local procedure call. *Gcc* and many other C compilers push input-parameter values on the stack prior to procedure invocation. Figure 2 shows the stack layout for a procedure called with 3 input parameters.

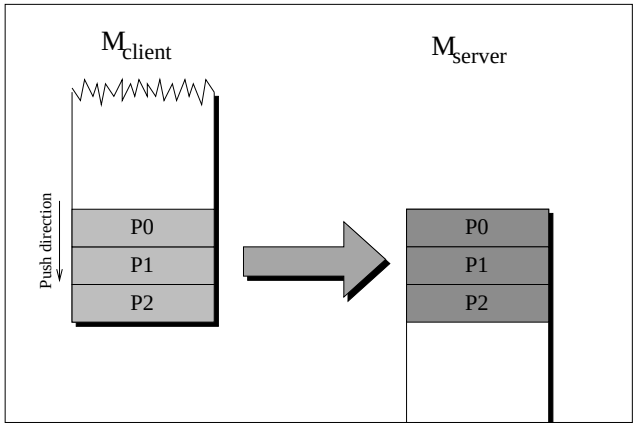

Figure 2. Procedure with 3 input parameters.

Now look at the remote case. Three parameter values have just been pushed on to the client stack (left, M_{client}). On the server side (right), M_{server} would ideally expect a stack of exactly the same content since M_{server} has exactly the same parameters as M_{client}. Then, the stub-code had basically to copy the stack frame one-to-one from the client to the server stack. No additional operations would be required for parameter marshalling/unmarshalling.

Since out parameters in C are typically implemented through pointers (which are passed as in parameters), we have to extend the parameter set by pointers that point to those variables that are later sent back to the client as out parameters. Figure 3 illustrates the three basic layouts:

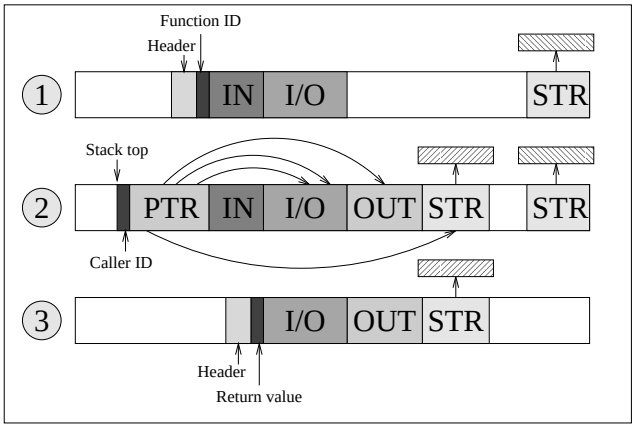

Figure 3. Message layouts. *(1): sent by the client to the server; (2): received message, extended to server stack; (3): message sent back by the server to the client.*

1. The client construct a message that contains all in and inout values (plus optional strings). The message buffer has enough space to receive the reply message from the server.
2. The server extends the received message by pointers that make the inout and out parameters (and optional strings) accessible for the server procedure M_{server}. Then it invokes M_{server}. As a normal C function, it works on its input parameters (PTR and IN).
3. After returning from M_{server}, the stub removes pointer and in parameters from the stack, pushes the return value and an appropriate message header, and sends the resulting reply message to the client.

An immediate consequence of the stack and message layouts is that the IDL compiler must sort parameters to enforce the sequence in, inout, out.[1]

3.3 Complex Data Types

At this time, the only data types IDL[4] handles are 32-bit words and strings. It will soon be extended by flexpages to also handle mapping through IDL functions. Any other data type can be implemented with those. Large objects like arrays or structs can be transferred as strings, while small objects (characters, short integers) may safely be extended to 32-bit words.

Extending smaller objects to words has no additional costs since *gcc* maps such objects to words anyhow when generating local function calls. Implementing large data types as indirect strings is beneficial since it avoids copying them into the message buffer.

[1] A similar sorting mechanism is used to collect string parameters and flexpages to be mapped.

4 Generated Code — An Example

To illustrate further details, we analyze the output that the compiler generates for the file system function *pfs_write*:

```
int write([in] int handle, [in, out] int *pos,
          [in] int len, [in] int data_size,
          [in, size_is(data_size)] int *data);
```

IDL[4] generates three files, which contain the client stubs, the server stubs and the main server loop. Client and server stubs are generated as `asm` functions for *gcc*. The server loop is in C so that it can easily be modified by the application programmer. It is common to all functions and decodes incoming request, i.e. selects the appropriate server function and invokes it through the server stub:

```
setupNewBuffer();
ipcReceive();
do {
     unpackQuery();
     callStub();
     packResponse();
     setupNewBuffer();
     ipcReplyWait();
   } while (1);
```

Client stub

Table 2 shows the output IDL[4] creates for the pfs_write() call on the client side. Assuming that hands over two parameters in registers, this stub consists of 17 instructions. In detail, the code sections (referring to the numbers in the code) work as follows:

1. *Create descriptors for indirect strings*. pfs_write() has one input string, `*data`, so a descriptor has to be created.
2. *Marshal parameters*. The input and inout parameters are pushed on the stack; inout parameters go first. Note that the last two parameters (`len` and `handl`) are not pushed, but loaded into the `EBX` and `EDI` registers
3. *Append message header*. The header specifies the number of dwords (32-bit words) to be transferred for both directions, as well as one dword for the mapping function, which is not used here.
4. *Load registers for IPC and supply function key*. IDL[4] needs to specify the send and receive buffer addresses and a timeout. The function key is transferred via registers and loaded here as well.
5. *Invoke IPC call.*
6. *Unmarshal server output*. In the case of pfs_write(), a return value and the `*pos` parameter must be handled. These can be transferred

via registers, so the memory buffer is entirely discarded.

Server stub

The stub (see Table 3) is called from the server loop. It converts the request message from the client into a stack frame for the server function:

1. *Move the stack pointer* to the message buffer. The message header and the function ID (which is the first dword in the payload) can be overwritten, so the new ESP points to the fifth dword in the buffer
2. *Add pointers to strings and output values*. First, a pointer to *pos is pushed, then one to the input string buffer. Finally, the ID of the source thread is supplied.
3. *Perform function call.*
4. *Create reply message*. The input values and pointers are discarded, then the return value and a new message header are appended.
5. *Restore the stack pointer*. Its original value was saved in EBP during the function call, as it is the only register that is automatically saved by GCC.

5 A Preliminary Performance Comparison

Table 1 shows the number of stub instructions generated by Flick and IDL^4 for three SawMill file system functions — *pfs_open, pfs_write,* and *pfs_get_direntries.* The numbers include all instructions that are executed in stubs and in the central server loop. For comparison, the number of instructions the L4 microkernel executes for the according IPCs is also included. The effective communication costs is the sum of the stub costs — either Flick or IDL^4 — and the IPC costs.

Flick stubs take typically as many instructions as the microkernel needs for the IPC system call (including message copy). IDL^4 uses only 1/3 as many instructions.

We are still in the process of analyzing whether and how much better code could be generated.

6. Conclusion and Future Work

IDL^4 shows that efficient stub-code can be generated with reasonable effort. However, it is still an open question how specialized (with respect to target compiler and target OS) an optimizing IDL compiler must be.

An obvious next step is therefore to find out whether and how the current results can be generalized. An ideal solution would permit the extension of the portable Flick compiler with the IDL^4 code-generation techniques.

int ***pfs_open*** ([in] int client, fobj, flags, mode, [out] int *handle)

	IPC (kernel)	**Flick stub**	**IDL^4 stub**
client → server	*165*	151	55
client ← server	*90*	130	49
total	*255*	281	104

int ***pfs_write*** ([in] int handle, [in,out] *pos, [in] int len, data_size, [in, size_is data_size] int *data)

	IPC (kernel)	**Flick stub**	**IDL^4 stub**
client → server	*250*	200	57
client ← server	*90*	130	48
total	*340*	330	105

int ***pfs_get_direntries*** ([in] int handle, [in,out] *pos, [in] int count, [out] int data_size, [out, size_is data_size] int **data)

	IPC (kernel)	**Flick stub**	**IDL^4 stub**
client → server	*159*	189	66
client ← server	*229*	175	51
total	*388*	364	117

Table 1. *Instructions executed for Flick and IDL^4 stubs (client+server). The* IPC *column shows the instructions executed by the microkernel per IPC (depends on message type and size). The entire effective communication cost is either the sum of* IPC *and Flick stub or the sum of* IPC *and IDL^4 stub.*

References

[1] J. Bruno, J. Brustoloni, E. Gabber, A. Silberschatz, and C. Small. Pebble: A component-based operating system for embedded applications. *Proc. USENIX Workshop on Embedded Systems*, pages 55–65, 1999.

[2] E. Eide, K. Frei, B. Ford, J. Lepreau, and G. Lindstorm. Flick: A flexible, optimizing idl compiler. *Proceedings of the ACM SIGPLAN '97 Conference on Programming Language Design and Implementation (PLDI)*, pages 44–56, June 1997.

[3] A. Gefflaut, T. Jaeger, Y. Park, K. Elphinstone, V. Uhlig, J. Tidswell, L. Deller, and L. Reuther. Building efficient and robust multiserver systems: The SawMill approach. Submitted to the 4th Symposium on Operating Systems Design and Implementation (OSDI), 2000.

[4] H. Härtig, M. Hohmuth, J. Liedtke, S. Schönberg, and J. Wolter. The performance of μ-kernel-based systems. In *16th ACM Symposium on Operating System Principles (SOSP)*, pages 66–77, St. Malo, oct 1997.

[5] The Object Management Group (OMG). *The Complete CORBAServices Book*. http://www.omg.org/library/csindx.html.

```
__inline__ extern sdword pfs_write(
  sm_service_t __service, sdword handl,
  sdword *pos, sdword len,
  sdword data_size, sdword *data)

{
  dword __return;

  int dummy0,dummy1,dummy2,dummy3;

  asm volatile (sub   $8, %esp);
  asm volatile (pushl %0 ::"g" ((int)data));
  asm volatile (pushl %0 ::"g" (data_size));

  asm volatile (pushl %0 ::"g" (*pos));
  asm volatile (pushl %0 ::"g" (data_size));

  asm volatile (
                sub   $12, %%esp

                pushl $0xA100      // (3)
                pushl $0x8000
                pushl $0

                mov   %%esp, %%eax // (4)
                pushl %%ebp
                xor   %%ebp, %%ebp
                mov   func_id, %%edx
                xor   %%ecx, %%ecx

                int   $0x30        // (5)

                popl  %%ebp        // (6)
                add   $48, %%esp

       :  "=S" (dummy0), "=d" (__return),
          "=b" (*pos), "=D" (dummy3)
       :  "S" (__service), "D" (len),
          "b" (handl)
       :  "%eax", "%ecx"
  );

  return __return;
}
```

Table 2. *Client stub for* pfs_write.

```
__inline__ extern void *call_pfs_write(void *buf,
        int com_source, int *strlist)

{
  int __return,dummy0,dummy1;

  asm volatile (
                pushl %%ebp          // (1)
                mov   %%esp, %%ebp
                mov   %%eax, %%esp

                mov   %%eax, %%edi   // (2)
                add   $12, %%edi
                pushl %%edi
                pushl 4(%%esi)
                pushl %%ebx

                call  _pfs_write     // (3)

                add   $24, %%esp     // (4)
                pushl %%eax
                pushl $0x2000
                pushl $0x2000
                pushl $0

                mov   %%esp, %%eax   // (5)
                mov   %%ebp, %%esp
                popl  %%ebp

       :  "=a" (__return), "=b" (dummy0),
          "=S" (dummy1)
       :  "a" (buf), "b" (com_source),
          "S" ((int)strlist)
       :  "%ecx", "%edx", "%edi"
  );

  return (void*)__return;
}
```

Table 3. *Server stub for* pfs_write.