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@os.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 that produces near-optimal stub code for gcc *and the L4 microkernel. The current experimental* IDL^4 *compiler cooperates with the* gcc *compiler and its x86 code generator. Other compilers or target machines would require different optimizations. In most cases, the generated stub code is approximately 3 times faster (and shorter) than the code generated by a commonly used portable IDL compiler. Benchmarks have shown that efficient stubs can increase application performance by more than 10 percent. The results are applied within IBM's* SawMill *project that aims at technology for constructing multi-server operating systems.*

1 Motivation

Multi-server and component-based systems are promising architectural approaches for handling 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 the inter-process communication (IPC) mechanisms offered by a microkernel.

Firstly, component interaction in such systems has to be highly efficient. Therefore, for over a decade, performance-oriented research focused on microkernel construction, in particular IPC performance, finally resulting in acceptable IPC overheads (100…200 cycles)[6, 2].

Secondly, component interaction has to be conveniently usable from an application programmer's perspective. This requirement led to the development of interface-definition languages (IDLs), e.g. Corba IDL [7], DCOM [3] and their corresponding 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, a programmer can specify and use remote interfaces as easily as internal ones.

So far, IDL-compiler research has focused more on generating code in a portable and adaptable way than on producing efficient stubs. In fact, stub-code performance was insignificant 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 [4] for the *SawMill* Linux file system [5], we found that the generated user-level stub code consumed about 260 instructions per read request. When reading a 4K block from the file system, the stub code adds an overhead of about 17% due to stub instructions. (The stub code may also generates further indirect costs through side effects such as cache pollution.) For an industrial system, such overheads can no longer be ignored.

Hand coding of the aforementioned stub resulted in 80 instead of 260 instructions. Although this was a singular experiment, it gave us some evidence that improving stub-code generation might be worthwhile. The potentially achievable reductions justified a compiler-construction experiment to explore 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 for understanding the subsequent discussion such as IDL syntax, L4-IPC mechanisms, 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 the code-generation principles. Finally, Section 5 reports on the achieved stub-code quality, Section 6 discusses the costs of adapting the system to other processor architectures and compilers, and Section 7 concludes.

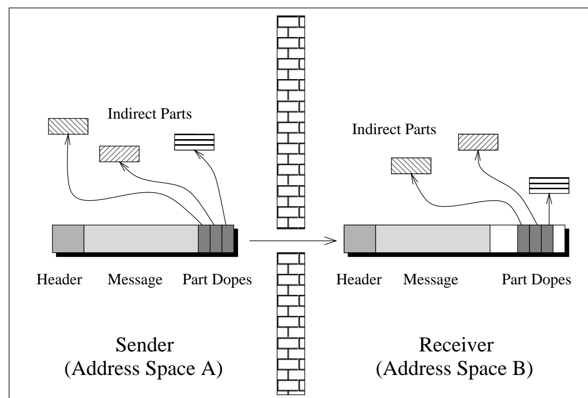

Figure 1. Complex memory message including indirect strings.

2 Prerequisites

2.1 L4/x86 IPC

L4's [6] basic communication paradigm 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 across 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 a register message because it involves copying from/to memory, and the kernel might have to establish a temporary mapping to make both address spaces available at the same time.

Indirect strings avoid 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 such strings can be specified so that the IPC can copy directly from server object to 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.

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 SawMill

IBM's *SawMill* project aims 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 systems customized to device and application requirements has increased significantly. The development and maintenance of these operating systems is quite unwieldy. As a first step, 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 which consists of Linux-based components running on top of the L4 microkernel. It provides typical system services through multiple user-level servers, such as file systems, device drivers and network systems. Further general components such as memory servers, task servers, and access control managers enable the composition of a coherent Linux system.

2.3 Flick

IDL Compilers such as Flick [4] are relatively easy to port to a new OS or middleware kernel, and they are extensible through new data types. The output of an IDL compiler is typically used as input for a general-purpose compiler, e.g. *gcc*, that a 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 with *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; consequently, inefficient code is generated.

3 Designing a Stub Model

3.1 A Simple Stub Model

We first describe a simple stub model to illustrate the tasks performed by stub code on the client side 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 this 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. The fact remains hidden 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's reply. The reply message contains all out and inout result values. The client stub unpacks these values from the message and stores them in the appropriate client parameters (*unmarshalling*). In detail, the client stub M_{client} —

- C1 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 (*unmarhalling*); and
- C4 returns to the invoking client.

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, and unmarshalling of parameters. 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 finally 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, e.g. 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 determined by the general-purpose compiler used, 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 can 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. 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

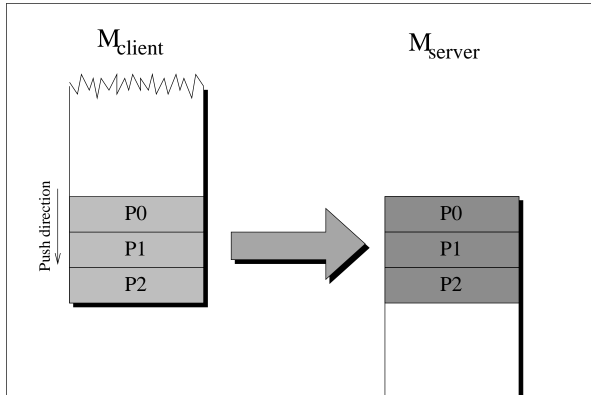

Figure 2. Procedure with 3 input parameters.

M_{server} has exactly the same parameters as M_{client}. Basically, the stub code had 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:

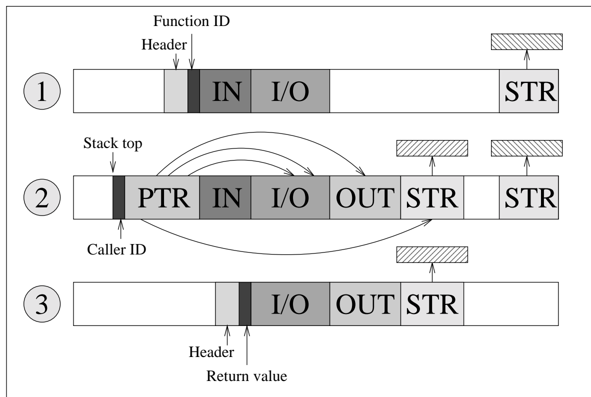

Figure 3. Message layouts. *(1): sent by the client to the server; (2): received message, extended to server stack; (3): message sent back by the server to the client.*

1. The client constructs 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, IN and the caller ID).
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 (up to 2 MB). It will be extended by pages to also handle mapping through IDL functions. Any other data type can be implemented through those basic types. 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.

4 Generated Code — An Example

To further illustrate details, we analyze the output that the compiler generates for the function *pfs_write* of the physical file system (pfs):

```
int pfs_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 an application programmer. It is common to all functions and decodes incoming requests, i.e. selects the appropriate server function and invokes it through the server stub:

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

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

Client stub

Table 1 shows the output IDL^4 creates for the pfs_write() call on the client side. Assuming it 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. *Generate message header*. The header specifies the number of dwords 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 2) 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 added.
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*.

```
__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));    // (1) push in string
    asm volatile (pushl   %0 ::"g" (data_size));    //     descriptor

    asm volatile (pushl   %0 ::"g" (*pos));         // (2) push 2 in
    asm volatile (pushl   %0 ::"g" (data_size));    //     parameters

    asm volatile (
        sub     $12, %%esp

        pushl   $0xA100             // (3) msg header,
        pushl   $0x8000             //     bits describe msg struct
        pushl   $0

        mov     %%esp, %%eax        // (4) ipc register setup
        pushl   %%ebp               //     save frame prt reg
        xor     %%ebp, %%ebp        //     reply msg type = short
        mov     func_id, %%edx      //     function key
        xor     %%ecx, %%ecx        //     timeouts = infinity

        int     $0x30 // (5)

        popl    %%ebp               // (6) (restore frame ptr reg)
        add     $48, %%esp          //     release stack space

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

return __return;
}
```

Table 1. *Client stub for* pfs_write.

5 Performance

5.1 Measurement Environment — SawMill Linux

IDL^4 is used in the *SawMill* project for component communications. *SawMill* Linux is a Linux-derived multi-server OS where physical file systems (PFS), file and buffer cache, device drivers, network stack, VM subsystems such as anonymous memory, etc. are all implemented as user-level servers that communicate through L4 IPC and IDL^4 stubs.

For *SawMill*, we analyze the stubs that are required to let a normal Linux process execute file-system operations such as *open, read,* and *write*. The physical file system we used in the experiments is compatible to

```
__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 2. *Server stub for* pfs_write.

Linux' *ext2*. In fact, the *ext2* code was extracted from Linux and then combined with IDL^4-generated server templates. The resulting ext2-compliant PFS runs as a user-level server in its own address space. Libraries have been modified such that now IDL^4 stubs and L4 IPC communicate with *SawMill* servers. An *open* request is always sent first to the virtual file system (VFS) which propagates it to the corresponding PFS server. Subsequent *read/write* requests, however, are handled through direct communication between the user application and that PFS server, i.e., need only one RPC (two IPCs).

The normal *SawMill* Linux has all stubs generated by the IDL^4 compiler. In addition, we generated a second version of *SawMill* Linux whose stubs were all generated by the Flick compiler. For both versions, we measured stub instructions and application performance.

For our measurements, we used a Pentium III running at 500 MHz with 64 MB of main memory and a 540 MB IDE disk drive (IBM DALA-3540).

5.2 Effects On IOzone Throughput

The IOzone benchmark [1] begins by writing a file of 64kB, then it reads the contents twice. In the second read phase, all requests can be backed by the page cache. The performance of the second phase is completely determined by processor operations, basically for communication and for copying data into the user program's buffer, and not by disk accesses.

We measured reread throughput where IOzone read 4 KB[2] of file data per read request. Table 3 presents the overall performance results reported by IOzone (ten consecutive iterations). IDL^4 improves the IOzone throughput by approximately 13%. The time for a 4-KB read request decreases from 8.0 μs to 7.0 μs. Since reread costs are dominated by the data copy costs this result can only be explained by significant improvements in stub code.

IOzone reread throughput on *SawMill* Linux using	
Flick stubs	**IDL^4 stubs**
503 kB/s ± *17 kB/s*	**569 kB/s** ± *18 kB/s* *(+13%)*

Table 3. *Overall throughput (± standard deviation) in the IOzone benchmark.*

5.3 Stub-Code Instructions

To analyze the stub-code performance, we counted the executed instructions for the Flick-generated and the IDL^4-generated stubs. Table 4 compares the results 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 corresponding IPCs is also included. (Note that complex operations such as block transfer operations are counted per iteration.) The effective communication costs are then given by adding the stub costs —either Flick or IDL^4— to the IPC costs.

[2]Longer read requests effectively decrease application performance, independently of whether pure monolithic Linux or *SawMill*/Flick or *SawMill*/IDL^4 is used: The Pentium L1 cache has a size of 16 KB. If, e.g., 8 KB of data are copied from the page cache to the user buffer, this operation already floods the entire cache. So every other application or file system data access leads at first to a cache miss. Furthermore, since some further cache lines are also used for the data copy, the first part of the user buffer will be flushed from L1 at the end of the copy operation. Effectively, most application accesses to the data read will thus also lead to L1 cache miss except if a clever application would read its data or if the OS would copy its data in reverse direction.

int **pfs_open** ([in] int client, fobj, flags, mode, [out] int *handle)			
	IPC (kernel)	**Flick stub**	**IDL^4 stub**
client → server	*163*	116	65 *(–44%)*
client ← server	*95*	105	37 *(–61%)*
total	*258*	221	102 *(–54%)*
eff. comm. instructions, *IPCs+stubs*		**479**	**360** *(–25%)*

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	*248*	150	73 *(–51%)*
client ← server	*95*	105	38 *(–64%)*
total	*343*	255	111 *(–56%)*
eff. comm. instructions, *IPCs+stubs*		**598**	**454** *(–24%)*

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	*157*	145	79 *(–46%)*
client ← server	*248*	140	42 *(–70%)*
total	*405*	285	121 *(–58%)*
eff. comm. instructions, *IPCs+stubs*		**690**	**526** *(–24%)*

Table 4. *Instructions executed for Flick and* IDL^4 *stubs (client+server). The* IPC *column shows the instructions executed by the microkernel per IPC (this depends on message type and size). The* **effective communication instructions** *are the sum of the required* IPC *(kernel) instructions* plus *the (user) instructions of the stubs.*

Flick stubs take almost as many instructions as the microkernel needs for the IPC system call (including the message copy). Current IDL^4 stubs use only half as many instructions.

6 Portability Versus Specialization

The IDL^4 experiment gave us some evidence that specialization in stub-code generation pays and is perhaps even necessary for industrial acceptance of component-based system construction. However, the obvious questions are (1) how portable can an optimizing IDL code generator be made, and (2) what efforts are required to port a specific code generator to a different compiler or machine architecture?

Currently, the IDL^4 code generation is specialized for the *gcc* compiler and x86 processors. From our current experience, we can give some raw estimates about the costs to adapt IDL^4 to other architectures:

New register link conditions: Low adaptation costs, comparable to those that are required to modify the C bindings for all 7 microkernel system calls.

Different Processor: Low adaptation costs as long as the stack layout is similar. Basically, the stub templates used by the IDL^4 code generator have to be translated into the new machine/assembler language.

Different stack layout: Depending on how different the stack layout is, adaptation costs might be lower or higher. Different orderings or distances on the stack are easy; a runtime model without a stack might require designing a new data model for cross-address space parameter transfer.

Different C compiler: Easy if the C compiler offers *inline asm* procedures exactly like *gcc*. Medium-high costs if the compiler offers basically the same features but uses different syntax. Impossible or ineffective if the compiler offers no such features.

The last point is probably the most critical one. Optimization is hard or even impossible if the C compiler does not offer access to its code generation process. However, this seems to be an inherent problem of separating the IDL and the programming language. In all other cases, the adaptation costs are similar if not lower than porting a normal compiler.

7 Conclusions

IDL^4 shows that efficient stub code can be generated with reasonable effort. With the availability of fast IPC, the gains achievable through optimized stub code are becoming relevant for component-based systems. Multi-server operating systems can probably not be built efficiently without such optimized stubs.

We have shown that significant performance improvements are possible. Nevertheless, it is still open, how far the current IDL^4-generated stubs are from the optimum.

The optimized stub-code generation requires specialization of the IDL compiler's code generator in two dimensions, firstly, toward the target programming language and compiler, secondly, toward the target machine. In this area, two questions are still open: (1) How specialized (with respect to acceptable efficiency) must an optimizing IDL compiler be? (2) Can we find a small set of templates and/or methods that permit easy and low-cost specialization of an optimizing IDL compiler

for most existing programming-language compilers and hardware architectures?

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

References

[1] *The IOZone filesystem benchmark*, April 2000. Available from http://www.iozone.org/.

[2] 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.

[3] G. Eddon and H. Eddon. *Inside Distributed COM*. Microsoft Press, 1998.

[4] Eric Eide, Kevin Frei, Bryan Ford, Jay Lepreau, and Gary 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.

[5] Alain Gefflaut, Trent Jaeger, Yoonho Park, Jochen Liedtke, Kevin Elphinstone, Volkmar Uhlig, Jonathon E. Tidswell, Luke Deller, and Lars Reuther. The SawMill multi-server approach. In 9th *ACM SIGOPS European Workshop*, Koldingfjord, Denmark, September 2000.

[6] 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.

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