Using Platform-Specific Optimizations in Stub-Code Generation

Andreas Haeberlen
University of Karlsruhe
haeberlen@ira.uka.de

July 2002

Abstract

The use of a stub code generator can greatly reduce the effort required to implement a multi-server system on top of a microkernel. However, stub code has traditionally been highly generic and therefore rather slow, which has prevented it from being used in performance-critical applications.

In this work, we show that this restriction can be eliminated by specialization for the underlying platform, i.e. by exploiting specific properties of the kernel, compiler, or hardware platform that are being used. We demonstrate our approach for an example platform, the L4 microkernel, and present a variety of optimization techniques that can be used to improve stub code performance on this platform.

Our techniques are implemented and validated in IDL^4, our optimizing stub code generator. A comparison with traditional stub code generators shows that IDL^4 stubs can perform up to an order of magnitude better, and that they can increase application performance by more than 10 percent.

Contents

1 Introduction 4
1.1 Remote procedure call . . . 5
1.2 Interface definition languages . . . 5

2 Related Work 6

3 Target Platform 8
3.1 The L4 microkernel family . . . 8
3.2 The Intel Pentium II Processor . . . 10
3.3 The GNU C compiler . . . 11

4 Optimization 11
4.1 A simple RPC model . . . 12
4.2 Kernel-specific optimizations . . . 13
4.2.1 IPC characteristics . . . 13
4.2.2 Special system calls . . . 13
4.2.3 Lazy process switching . . . 14
4.2.4 Cacheability hints . . . 15
4.3 Compiler-specific optimizations . . . 15
4.3.1 Direct stack transfer . . . 15
4.3.2 Inline functions . . . 17
4.3.3 Compile-time evaluation . . . 18
4.3.4 Annotation . . . 18
4.3.5 Record partitioning . . . 19
4.3.6 Extending the calling convention . . . 19
4.4 Architecture-specific optimizations . . . 20
4.4.1 Reordering message elements . . . 20
4.4.2 Function tables . . . 21
4.4.3 Server-side stack switch . . . 21
4.4.4 Special instructions . . . 22
4.4.5 Cache-aware memory allocation . . . 22
4.5 Usage-specific optimizations . . . 23
4.5.1 Pass-by-reference . . . 23
4.5.2 Domain-local servers . . . 23
4.5.3 Annotated data types . . . 23
4.5.4 Custom presentation . . . 24

5 IDL^4 25
5.1 Requirements . . . 25
5.2 Design decisions . . . 25
5.2.1 CORBA and DCE . . . 25
5.2.2 IDL extensions . . . 26
5.2.3 Language mapping . . . 27
5.2.4 Buffer allocation . . . 27
5.3 Architecture . . . 27
5.3.1 Front end . . . 28
5.3.2 Marshaling stage . . . 28
5.3.3 Kernel-specific part . . . 29

5.3.4 Back end 29
5.3.5 Regression test 29

6 Evaluation **30**
6.1 Flick 30
6.2 DICE 30
6.3 Methodology 30
6.4 Microbenchmarks 31
6.5 Marshaling overhead 34
6.6 Application performance 35
6.7 Discussion 35

7 Conclusion **36**

1 Introduction

Operating systems have traditionally been designed with a monolithic structure. Most of the functionality, such as task management, scheduling, or device drivers, is implemented by a single entity, the kernel, which runs at the highest privilege level. This approach has been criticized for its lack of protection, because every kernel component has complete control; a faulty component can break other, unrelated components, or even crash the entire system.

Multi-server operating systems solve this problem by breaking the kernel into multiple components, which are executed in separate protection domains. Only one small component, the microkernel, remains at the highest privilege level; it implements essential primitives such as address space management or inter-process communication (IPC), which cannot be performed at a lower level. Apart from the robustness gained by increased protection, this approach also offers additional flexibility because components can easily be exchanged, and multiple versions can coexist in a single system.

However, in order to perform a specific function, the components may need to interact frequently. As they execute in different protection domains, this requires the use of IPC, which is not directly supported by most programming languages. Thus, additional code is needed to handle the communication, i.e. for packing data into messages, for sending those messages, and for receiving and unpacking them again. Writing and debugging large amounts of this code can be a tedious and error-prone task.

Fortunately, tools are available for generating communication code (stub code) automatically from a high-level specification. Many of those tools support the remote procedure call (RPC) which, from the programmer's perspective, works exactly like an ordinary procedure call, except that it can also be used to call procedures in another protection domain. Because of this similarity, RPC integrates well with many programming languages and is widely used today.

Apart from convenience, code generators offer many other benefits. They provide an abstraction from the underlying communication mechanism, hiding many unnecessary details from the programmer. Also, they are available for many platforms, so code that uses them is easily portable. Finally, due to a lack of type safety, hand-written communication code is hard to debug, whereas generated code is usually reliable and does not need any debugging.

However, stub code performance is usually not a primary goal. The reason is that RPC has traditionally been used in distributed systems, which are connected over a network; there, the overhead of the stub code is negligible because communication costs are much higher, sometimes by several orders of magnitude. This is different for modern microkernels, where high-speed IPC primitives are available; in this case, the relative overhead is considerable and may even exceed the actual transfer costs.

The result is that communication becomes prohibitively expensive, which is totally unacceptable for a multi-server system. Moreover, programmers may be tempted to base their design decisions on communication cost; as Birrell and Nelson have pointed out, "[RPC performance] seems important, lest communication become so expensive that application designers strenuously avoid it" [4].

In this paper, we present an approach that improves stub code performance by specialization for the underlying platform. We exploit specific knowledge of the microkernel, the hardware architecture, and the compiler that are being used, as well as certain characteristics of the application code.

To demonstrate our approach, we have built IDL^4, an optimizing stub code gener-

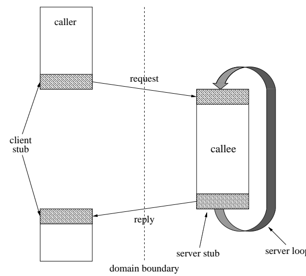

Figure 1: RPC scheme

ator for the L4 microkernel. Experiments show that IDL^4-generated code can perform up to an order of magnitude better than that of other optimizing code generators, and that the overall performance of applications can be improved by up to 13%.

The rest of this paper is structured as follows: Section 2 describes related work in the distributed and operating systems communities. Section 3 gives an overview of the target platform that was used in this work, and Section 4 describes the optimizations that we applied for this platform. In Section 5, we present the design and architecture of IDL^4, our optimizing stub code generator. Section 6 shows the measured performance of IDL^4, and Section 7 presents our conclusions.

1.1 Remote procedure call

It was already mentioned that RPCs mimic the semantics of local procedure calls because this is a natural extension to many programming languages. However, the remote procedure is executed in a different domain, so a communication primitive must be used instead of the usual stack- or register-based call. The caller uses the communication primitive to send a *request message* to the callee, then it suspends itself and waits for the callee to respond with a *reply message*.

When the procedure has arguments or return values, these must be included in the corresponding message. However, unlike the local case, references cannot be transferred because they have no meaning in the remote domain. Thus, all values must be copied during the call, i.e. the sender must encode (*marshal*) them into the message and the receiver must decode (*unmarshal*) them again.

Because most higher-level languages do not support RPC natively, additional code is needed for handling the communication. This code is often termed *stub code* because it serves as a placeholder for the actual function (see Figure 1). The *client stub* marshals the arguments and sends a request message, then it waits for the reply and unmarshals the return values. Its counterpart, the *server stub*, performs the inverse operations.

When an individual server exports multiple procedures to its clients, an additional *server loop* is required which receives all incoming requests and dispatches them to the appropriate server stub.

1.2 Interface definition languages

In order to generate stub code automatically, a formal specification of the remote procedures is required. This specification must contain at least the names of the procedures and their parameters, complete with size and direction (client to server, server to client, or both). However, many details, e.g. properties of the underlying communication

mechanism, can be left out because they can be determined by the code generator. This permits a high level of abstraction.

A variety of *interface definition languages (IDL)* have been specified for this purpose [13, 17, 24]. Here is an example interface definition in CORBA IDL:

```
interface foo {
  typedef sequence<char> buffer_t;
  short alpha(in string a);
  void beta(out buffer_t b);
};
```

This specification defines an interface `foo` which contains two procedures, `alpha` and `beta`. The first procedure takes a string as an argument and returns a short integer. The second procedure does not have any arguments, but returns a variable number of characters.

Because CORBA is designed for interoperability, the type system of its IDL is independent from any particular platform or target language; mappings have been specified for Ada, C, C++, COBOL, Java, Lisp, PL/1, Python and Smalltalk, and have been implemented for many platforms. For example, a stub code generator for the C language might produce the following code from this specification:

```
typedef struct {
  long _length, _maximum;
  void *data;
} buffer_t;

short foo_alpha(foo obj, char *a) {
  ...
};

void foo_beta(foo obj, buffer_t **b) {
  ...
};
```

Some IDLs are more platform-specific; for example, Matchmaker [24] has support for ports, a special data type of the Mach kernel, so it cannot be used for other platforms.

2 Related Work

A considerable amount of work has been dedicated to optimize RPC, and many different techniques have been demonstrated. In this section, we discuss some of the research directions, and how they relate to this work.

Virtual memory

Some approaches use virtual memory primitives to optimize the transfer of large messages. Peregrine [23], which is based on the V system [6], uses page remapping to efficiently move the call arguments and results between the client's and server's address spaces. The DEC Firefly RPC facility [32] uses a global buffer pool, which resides in memory shared among all user address spaces, whereas in the fbuf scheme presented by Druschel and Peterson [9], buffers are shared only across the members of

a particular data path. In the LRPC scheme presented by Bershad et al. [3], client and server use privately shared memory for argument passing.

The remapping approach uses move rather than copy semantics and is thus limited to situations where the client does not need further access to the data after the call. Also, revoking memory from an address space can be a costly operation on modern CPUs because it may require modification of the translation lookaside buffer (TLB). Shared memory can compromise the security model of the system.

Automatic program transformation

In the Tempo system [29], Muller et al. use source-level transformations to optimize the output of an existing IDL compiler (in this case, Sun RPC). Tempo applies partial evaluation to specialize the code at compile time, removing genericity where it is not needed. In micro-benchmarks, the authors measured speedups of up to 3.75.

Because Tempo uses existing stub code, it cannot apply optimizations that would involve changing the transfer algorithm itself. Nevertheless, this work clearly shows the high overhead that can be caused by genericity.

Compiled vs. interpretive stubs

A time vs. space tradeoff exists between these two approaches: Interpretive stubs are smaller, but compiled stubs yield better performance.

Gokhale and Schmidt [16] have pointed out the importance of a small memory footprint for embedded systems. The interpretive stubs generated by their TAO IDL compiler achieve comparable performance to hand-crafted compiled stubs (75-100%) at a relative code size of only 26-45%. On the other hand, speed-oriented IDL compilers like Flick [10] have demonstrated impressive performance with compiled stubs and aggressive inlining. Hoschka and Huitema have suggested a hybrid approach [21], where the decision between compiled and interpretive is based on estimated cost and subsequent solving of a Knapsack-like combination problem.

Presentation

In the OSI network reference model, a "presentation layer" handles the conversion between the local data representation and the network format. Clark and Tennenhouse [7] found that this conversion can be responsible for up to 97% of the total protocol overhead. In [31], O'Malley et al. demonstrate how conversion code can be improved; the stubs generated by their code generator, USC, were up to 20 times faster than conventional stubs.

Others have explored the possibility of customizing the presentation, i.e. adapting it to the application's needs. In DCE IDL [13], every interface definition may be accompanied by a separate file, the application configuration file (ACF), which may contain additional attributes, e.g. for controlling binding protocol or error condition treatment. The Concert system [2] partitions interface specifications into a network contract, which is relevant for interoperability, and an endpoint modifier, which represents the interface to the stub, i.e. the "programmer's contract". Ford et al. [12] introduce several new presentation attributes, e.g. for changing allocation semantics or for choosing between alternate transfer mechanisms.

Combining multiple optimizations

To build a general, high-performance RPC system, a single optimization technique is not sufficient; thus, many newer IDL compilers (e.g. [10, 15]) combine a variety of those techniques. Ford et al. [11] pointed out that the task of making many optimizations work together has become more challenging than simply finding them.

3 Target Platform

In this work, our approach is to maximize stub code performance by specializing it to the underlying platform, i.e. to remove genericity where it is not needed. Of course, this technique is not restricted to one particular platform, but it cannot be discussed merely at an abstract level. Therefore, a specific environment was chosen for this paper.

Our environment consists of a hardware platform (the Intel Pentium II processor), a microkernel (L4/Hazelnut), and the compiler used to generate application binaries (`gcc`). This section explains each of those elements in more detail.

3.1 The L4 microkernel family

The L4 microkernel was developed by Jochen Liedtke at the German National Research Center for Information Technology (GMD) and the IBM T. J. Watson Research Center [22]. As a true microkernel, L4 implements only a minimal set of concepts:

- Address spaces and virtual memory, which is managed by external pagers
- Threads, which are activities executing inside an address space
- Synchronous, message-based inter-process communication (IPC)
- Priority-based scheduling with hard priorities

Initially implemented for the IA-32, L4 has been ported to a variety of other architectures, including Alpha, MIPS, StrongARM, IA-64 and PowerPC; also, the kernel API has been revised twice (V2, X0, V4).

For this work, we chose the IA-32 version of the Hazelnut kernel, which implements the X0 version of the API [27]. Additionally, in order to cover the upcoming V4 API [33], we sometimes include references to the Pistachio kernel.

Because this work focuses on communication, we mainly discuss the IPC kernel primitive and omit other functions, such as scheduling or thread control. Please refer to the kernel manuals [26, 27, 33] for more information.

Inter-process communication

In L4, IPC is synchronous. This means that both sender and receiver have to be ready at the same time, and that one partner may be blocked while waiting for the other. Also, communication is unbuffered, i.e. no intermediate kernel object is used to store messages; instead, data is copied directly from the sender to the receiver.

Four different message types are supported. The simplest is the *register message*; it consists of a small number of values that are loaded into processor registers by the sender before it traps into the kernel. This message type has a very low latency because the kernel only needs to perform a context switch, while the values can remain in their

respective registers and are delivered to the receiver untouched. However, on many platforms, the number of registers that can be used for register messages is quite small.

When the message does not fit into registers, it must be partly stored in memory. The sender collects the data in a contiguous buffer, the *memory message*, and supplies a reference to the kernel. During IPC, the kernel then copies the contents of the message to a buffer in the receiver's address space.

It is also possible to construct multi-part messages by extending a memory message with *indirect parts*. An indirect part points to additional data at a different memory location, which is also copied during IPC. This saves one copy operation on the sender side because the data does not have to be copied to the primary buffer; however, it also causes some overhead for the indirection. Thus, indirect parts should only be used for large, contiguous memory objects.

Finally, a *map message* can be used to map virtual memory from the sender's address space to the receiver. Map messages are similar to register messages or memory messages, except that they contain one or more *send flexpages*, which describe regions of virtual memory.

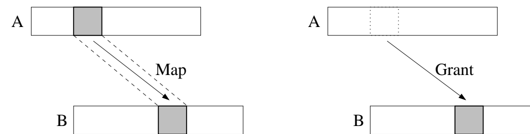

Figure 2: Virtual memory primitives

A flexpage may either be mapped or granted (see Figure 2). When a page is mapped, it is shared between sender and receiver, i.e. it is accessible from both address spaces; however, the sender can revoke the mapping later. When a page is granted, the receiver is given complete control, i.e. the page disappears from the sender's address space and cannot be revoked.

Memory messages and indirect parts are generally transferred with a temporary mapping [25]. Instead of buffering messages in kernel memory, the target region is mapped temporarily to a transfer window in the source address space, then the message is copied directly into the transfer window. This causes additional misses in the translation lookaside buffer (TLB), but saves one copy operation and decreases cache pollution.

The Hazelnut kernel

The Hazelnut kernel implements the X0 kernel API [27]. On IA-32, this means that IPC always transfers a register message of three 32-bit machine words; optionally, a memory message of up to 2MB can be added, which may contain an arbitrary number of send flexpages and up to 31 indirect parts.

The simplest (and fastest) message type is an intra-domain register message, i.e. a register message that is sent between two threads in the same address space. It has a best-case overhead of 99 cycles, which is the cost for entering and leaving kernel mode, for a context switch from sender to receiver, and for a few simple checks.

Inter-domain messages are more expensive because the mapping between virtual and physical addresses must be invalidated; on IA-32, this can only be done by flushing (and subsequently repopulating) the TLB. Also, memory messages and indirect parts cause additional overhead for creating and removing the temporary mapping.

Recent work [28] has introduced an optimization for intra-domain IPC which does not require a switch to kernel mode in certain cases. With this optimization, the best-case overhead can be reduced to approximately 23 cycles [35].

The Pistachio kernel

The Pistachio kernel implements the V4 kernel API [33]. The most important difference to the X0 API is that this version does not support memory messages; instead, every thread has 64 message registers (MRs) which can be transferred to the receiver's MRs partly or entirely and may also contain indirect parts (StringItems) or send flexpages (MapItems and GrantItems). On architectures that have less than 64 CPU registers, some MRs may be virtual, i.e. backed by memory.

Because Pistachio is still an experimental kernel, no performance numbers are currently available. However, it is expected that medium-sized messages can be transferred considerably faster, because it is no longer necessary to establish a shared mapping.

Stub code for L4 microkernels

The IPC primitive offered by current L4 microkernels is very flexible. A stub code generator can usually choose between multiple different ways to implement a particular message transfer. This choice is important because it has a significant impact on performance; for example, it can be considerably more efficient to transfer a large data structure with a multi-part message than with a simple memory message.

Also, a stub code generator must provide support for different kernel and API versions. Like any research project, L4 is highly dynamic, so there are often changes to the API; three major revisions have already been issued. The stub code generator should provide an abstraction layer to isolate application code from smaller changes, while at the same time it should use new features to improve performance.

3.2 The Intel Pentium II Processor

In this work, we focus on the Pentium II and its successor, the Pentium III, because the Pentium 4 was not yet available when the project was started. Both processors are still very common and are currently used in most desktop PCs.

The Pentium II is an implementation of Intel's 32-bit architecture, the IA-32, which has a CISC-style instruction set. Internally, the CISC instructions are broken down into simple operations called micro-ops, or μops, which are then issued to a RISC-style execution core; thus, the Pentium II can combine an excellent code density with high execution speed. However, some CISC instructions are more thoroughly optimized than others, so a complex instruction can actually be slower than an equivalent sequence of simple instructions. See [8] for more details.

When optimizing application code for the Pentium II, it is important to consider the following points:

- *High memory latency*. At a high clock speed, accessing main memory is extremely expensive, so cache efficiency can be a limiting factor. The chip has on-die instruction and data caches of 16kB each and an additional second-level cache of 512kB on the module (all 4-way set-associative, 32 byte cache lines).

- *Small register set*. The IA-32 has only 8 general-purpose registers, including the stack pointer and a base pointer. Thus, compilers are often forced to generate spill code, i.e. free some registers by moving their values to memory, and to reload them later.
- *Branch prediction*. In order to increase the efficiency of its pipelined core, the Pentium II attempts to predict conditional branches and tentatively fills the pipeline with the most likely path. However, when a misprediction occurs, the entire pipeline must be flushed and refilled.

These points have become even more important with the release of the Pentium 4, whose memory latency is comparably even higher (due to clock speeds of more than 2GHz), and which has a much longer pipeline (20 stages, which is twice the length of the Pentium III pipeline). Therefore, we believe that our optimization techniques are also applicable to this chip.

Other properties of then Pentium II, such as the architecture of the translation lookaside buffer (TLB), are not directly relevant here. The stub code generator produces code in a higher-level language, e.g. C or C++, which is then compiled separately; thus, it cannot control many low-level features of the CPU.

3.3 The GNU C compiler

The GNU C compiler (`gcc`) was chosen for this work because it is widely used and has support for a variety of platforms. Also, it is well suited for low-level system code because it has a powerful interface for inline assembly, which is important e.g. for system calls.

However, `gcc` has a poor optimizer and cannot match the performance of proprietary compilers such as Intel's `vtune`. Also, we have encountered problems with some releases, ranging from occasional crashes to incorrect code.

4 Optimization

In this section, we present various methods for optimizing generated stub code; most of these methods have been implemented and validated in IDL^4, our optimizing IDL compiler, which is presented in the following section.

Our basic claim is that the best performance can be obtained by specialization for the target platform. To make this possible, the code generator is given detailed information about:

- the particular *kernel* that the system will run on
- the particular *compiler* that will be used
- the particular *architecture* of the target machine
- the particular *usage pattern* of the application code

Because this work is targeted at multi-server operating systems (and not distributed systems), we assume that interoperability is not required, i.e. that client and server run on the same machine, use the same kernel, and that both stubs are compiled with the same IDL compiler. For the same reason, we assume that no messages are damaged

or lost by the underlying transport mechanism (in this case, IPC). Finally, we restrict ourselves to static invocation, i.e. all interfaces must be known at compile time. This permits us to use a fixed message layout and eliminates the need for tagging message elements.

Our goal here is to maximize application performance, i.e. to increase execution speed; however, a variety of other criteria could equally be applied. In an embedded system, for example, a small memory footprint and low power consumption may be preferable, and a security system might require maximum confidentiality. While different goals would certainly result in different optimizations, the general approach is the same.

4.1 A simple RPC model

We first introduce a simple model for remote procedure calls. This model will be used later in this section to explain where a particular optimization is applied.

Assume that a client invokes a remote method M that is supplied by a server. The parameters of M can then be characterized as either *in* parameters, which are passed from the client to the server, *out* parameters, which are sent back to the client, or *inout* parameters, a combination of both.

In order to perform a call to M, the client marshals the in and inout parameters, sends the resulting message to the server and then waits for a reply. When the reply arrives, it unmarshals the inout and out parameters and then resumes normal execution. In detail, the client stub must:

- C1) Pass all input values to the stub code and transfer control to it
- C2) Allocate a sufficiently large message buffer
- C3) Marshal the in and inout parameters, and add an identifier for M
- C4) Send the resulting message to the server, and wait for a reply
- C5) Check for any error conditions that may have occurred
- C6) Unmarshal the out and inout parameters
- C7) Release the storage occupied by the message buffer
- C8) Return all output values and resume execution

On the server side, the request is received by the server loop and dispatched to the appropriate stub. In detail, the server loop must:

- L1) Wait for a request message to arrive
- L2) Extract the method identifier and determine the appropriate stub
- L3) Pass the request message to the stub and transfer control to it
- L4) If a reply message is returned, send it back to the client

When it receives a request, the server stub unmarshals the in and inout parameters and passes them to M. When M completes, it marshals the inout and out parameters and prepares a reply message which is to be sent back to the client. In detail, the server stub must:

- S1) Unmarshal the in and inout parameters
- S2) Pass the resulting values to M, and transfer control to it
- S3) Check for any error conditions that may have occurred
- S4) Marshal the inout and out parameters, and create a reply

4.2 Kernel-specific optimizations

It was already mentioned that the L4 microkernel platform consists of many different implementations:

- The kernel interface exists in three major revisions: V2, X0 and V4
- Many different hardware platforms are supported, including Alpha, IA-32, IA-64, StrongARM, MIPS, and PowerPC
- Some platforms have special support for different CPU versions. For example, the IA-32 platform distinguishes the 486, Pentium, Pentium II/III and Pentium 4.
- There are special kernels with support for new features, e.g. small address spaces or local IPC

This diversity is relevant for stub code generators in two ways. First, not all features are supported by all kernels, so the code generator must be very conservative when the target kernel is not known. Second, a feature may have different characteristics on different implementations, even when the API is the same.

For example, consider the L4 IPC mechanism. Due to its flexibility, there are usually multiple different ways to transfer a particular message; however, the tradeoff is different for every implementation. Thus, if the target kernel is known, performance can be improved by choosing the most efficient option for every call.

4.2.1 IPC characteristics

All L4 kernel interfaces provide at least three basic methods for copying data during IPC. A small, fixed number of machine words can be transferred directly in *registers*; additional data must be put into a *memory message* which is then copied to the receiver's address space. Finally, the buffer may contain references to *indirect parts*, which are logically part of the message but need not be copied into the buffer.

Obviously, a tradeoff exists between memory messages and indirect parts. The latter can save up to two copies (to and from the message buffer), but require address space manipulations for the temporary mapping, which cause an increased overhead. Figure 3 illustrates this fact for the Hazelnut kernel[1]; for small messages of up to 30 machine words, a simple memory message is more efficient, even though

Another tradeoff exists for very small messages. When the data fits entirely into a register message (in Hazelnut, up to three machine words), the memory message can be omitted; however, it may be necessary for multiple smaller values to share a register. The overhead of packing and unpacking those values must be considered, but is usually very small.

4.2.2 Special system calls

Some kernels support more than one way to invoke the IPC system call. For example, generic calls might coexist with special calls that are optimized for a certain scenario, and portable call methods might be supported along with version- or hardware-specific methods. However, not all calls are supported by all kernels, and there is often not a single call method that performs best for all messages. Thus, the IDL compiler can use

[1]Round-trip IPC with hot caches on a dual Pentium II/400; the second IPC is always short

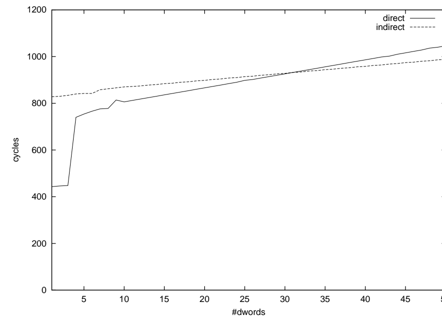

Figure 3: IPC copybreak

its kernel-specific knowledge to choose the call variant that is likely to deliver the best performance.

Hazelnut, for example, supports both the `sysenter` and the `int $0x30` instructions for trapping into the kernel. The former is faster, but works only on the Pentium II and newer processors, while the latter can also be used on a Pentium. This is why `sysenter` is not part of the official kernel ABI [27], and some implementations (Pip, Fiasco) do not support it. While a conservative code generator would have to use the slow call at all times, an optimizing code generator can use its kernel-specific knowledge to choose `sysenter` where it is available.

Note that this optimization trades portability for performance. While code optimized with the previous method still runs on other kernels (although slightly slower), a stub that uses `sysenter` does not work *at all* on a Pentium machine, or with the Pip kernel. However, this is not a serious restriction because only the portability of the *binary* code is affected; binaries for other platforms can easily be generated by recompiling the stub code.

A similar optimization is possible for the version 4 ABI [33], which is implemented in Pistachio. It specifies that system calls are not invoked with a special machine instruction, but rather with an indirect call to the kernel interface page (KIP). The code generator could be allowed to inline this call, which would reduce the overhead and result in improved stub code performance.

4.2.3 Lazy process switching

Lazy process switching [28] is an optimization for frequent intra-domain communication; it allows the callee to run in the context of the caller for some time, thereby saving the cost for a context switch. The principle is part of the V4 interface and has also been implemented in an experimental variant of the Hazelnut kernel [35].

Special system calls (local IPC, lipc) are available in Hazelnut and Pistachio for switching lazily from the sender to the receiver. My experiments show that using this primitive can boost RPC performance by up to 900%; however, it can also add overhead when used improperly, e.g. for nonlocal calls. This means that the code generator can drastically improve performance by using lipc, but only if it a) knows that the target kernel supports it, and b) can assume that most of the calls will indeed be local. See Section 4.5.2 for further discussion.

4.2.4 Cacheability hints

The new V4 interface introduces a mechanism to improve cache efficiency. During IPC, an application can supply cacheability hints to the kernel, e.g. prevent it from allocating cache lines in lower levels, or disable caching entirely for certain message elements. Thus, it is possible to transfer large message elements without flushing the entire cache.

The fact that this feature is highly platform dependent makes it an ideal candidate for stub code optimization; however, it is unclear yet how the code generator can guess a suitable cache policy. The most promising method seems to be a user-specified attribute, e.g. as part of the ACF, but this seems problematic because the cacheability hint must be specified on the sender side, whereas it is the receiver that processes the data and thus would benefit most from a custom caching strategy.

4.3 Compiler-specific optimizations

Almost all stub code generators produce code in some higher-level language. Compared to object code or binary code, this approach has the advantage of keeping stub code generation separate from the main compiler; also, the stubs can be more easily integrated with application code. However, the stub code generator has little influence on the final machine code, so code quality largely depends on optimizations done by the main compiler.

Nevertheless, the stub code generator has a lot of freedom because a particular effect can usually be achieved in many different ways; for example, in a C program, unrelated statements can be reordered or different language constructs (`switch` instead of multiple `if`s) can be chosen. Thus, if detailed knowledge about the compiler is available, the code generator can take advantage of it by choosing the implementation that results in the most efficient code.

Because the `gcc` compiler was used in this work, the following analysis is largely `gcc`-specific. Also, some assumptions may not hold for architectures other than the IA-32; for example, the IA-64 does not use a stack-based calling convention. Nevertheless, a similar approach could be taken for any other compiler.

4.3.1 Direct stack transfer

When a remote procedure call to a method M is invoked, the input parameters need to be passed to the client stub (step C1 in the stub code model). One possible implementation is a conventional function call, which creates an activation record on the stack. A similar operation is performed later on the server side, when the procedure M is called; again, the parameters are pushed on the stack, and the activation record from the client side is recreated. Thus, the marshaling steps C2-C3 and S1 are apparently equivalent to a no-op.

Therefore, it is sometimes possible to omit the marshaling steps entirely. Instead, the client can simply send its activation record to the server, where the procedure M can be called directly (see figure 4).

As shown in [19], this approach leads to extremely efficient code for simple procedures; however, there are certain implementation issues:

- *Output parameters* are problematic because unlike input parameters, they are usually passed by reference. Thus, the activation record will contain pointers, which are meaningless in the callee's address space.

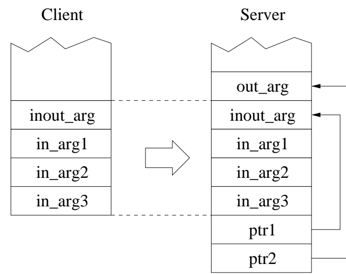

Figure 4: Direct Stack Transfer

This problem can be solved in two ways. One is to send the activation record as-is, i.e. including the meaningless pointers; they can be then overwritten with the correct values on the server side. The other is to remove the pointers on the client side and have the server add them again. This can be implemented efficiently by reordering the arguments so that the pointers are pushed first; thus, only the lower part of the stack frame needs to be sent.

- *Indirect strings* cannot easily be used because the corresponding descriptors would have to appear *after* the memory message. On the stack, the necessary space is likely to be occupied by other activation records.

- *Short messages* cannot be fully optimized because the calling convention requires that *all* arguments must be passed via the stack. For short messages, it is desirable to keep the values in registers and pass them directly to the IPC system call.

All of these issues can be solved by using inline assembler code on the client side, instead of the compiler-generated function call. This enables the code generator to alter the layout of the activation frame, so it can easily omit pointers, reserve space for indirect parts, or keep some of the values in registers. On the server side, some minor changes may still be required (e.g. pointers to the output values must be added), but the client can minimize those changes by choosing an appropriate message layout.

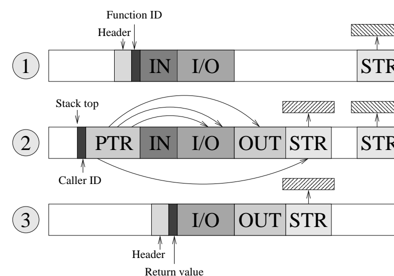

Figure 5: Message Layout

Figure 5 shows the recommended message layout which also uses the overlapping buffers optimization (see Section 4.4.1). The client pushes the inout values first, followed by the in values, the function ID, and a message header (step 1), while ensuring that enough free space remains for receiving the output message. The server allocates

space for the output values directly behind and then adds pointers to imitate pass-by-reference (step 2). After the call, it discards both the pointers and the input values, and sends the remaining message back to the client (step 3).

4.3.2 Inline functions

In a higher-level language, function calls are among the most costly operations. The caller pays high overhead costs, because it has to push the arguments on the stack, save its context, create a new context for the callee and transfer control to it. Upon return, it must dispose the callee context, restore its own one, and acquire the return values.

Additionally, there are indirect costs due to the fact that most compilers cannot apply optimizations across function calls, especially when the function is globally visible and it is not possible to determine all contexts from which it is called[2]. Also, many common optimizations, such as partial redundancy elimination (PRE) or common subexpression elimination (CSE) are usually limited to individual basic blocks.

The call-related overhead can be eliminated by inlining the code, i.e. by substituting it for every individual function call. One possible way to do this is to use a macro, another is to declare the function as `inline` and let the compiler perform the substitution.

For large functions, a tradeoff exists between execution speed and code size; also, more code may cause the instruction cache to be used less efficiently. However, stub code is usually small.

```
without_inline_stub:
   ...
   mov   var, %eax      ; caller invokes stub(var+8)
   add   $8, %eax
   push  %eax           ; argument is passed via the stack
   call  stub
   ...

stub:
   push  %ebp           ; establish new context
   mov   %esp, %ebp
   mov   8(%ebp), %ebx  ; load argument into EBX
   ipc                  ; invoke IPC (simplified)
   mov   %ebp, %esp
   pop   %ebp
   ret

with_inline_stub:
   ...
   mov   var, %ebx      ; load argument directly into EBX
   add   $8, %ebx       ; again, stub(var+8) is called
   ipc                  ; invoke IPC (simplified)
   ...
```

Figure 6: Inline stub code example

Figure 6 shows an example. The lower function inlines the stub code, so the compiler can eliminate both the call instruction and the entry/exit code.

Inlining is especially important for system calls with many parameters, such as IPC. The application has to load these parameters, e.g. the address of the callee and pointers

[2] When global knowledge is available, the problem could be alleviated with cross-procedural optimization. However, this is not supported by `gcc`.

to the message buffers, into registers before it traps into the kernel; also, the results are returned in registers after the call.

When the system call code is in a separate function, the register allocator in the compiler cannot know these constraints, so the argument values may reside in the activation record or in entirely different registers; they may also have been partly spilled to memory. This causes additional overhead, which can be avoided by inlining the system call code in the calling function. Thus, the compiler can avoid spilling the argument values and use the correct registers in the first place.

In Figure 6, this effect can be observed for the EBX register. Without inlining, the argument value is loaded in a different register (EAX) and then passed over the stack, whereas in the inlining case, the correct register is used from the beginning.

4.3.3 Compile-time evaluation

Generated code is usually more general than actually required. For example, it may contain functionality that is never used, or it may check for error conditions (e.g. null pointers) that can never occur. Theoretically, this could be prevented by adding more details to the specification (in this case, the interface definition), but it would add a lot of complexity and thus is not practicable.

Fortunately, the compiler can sometimes detect this situation and either specialize the corresponding code or remove parts of it entirely. However, the resulting machine code must still be correct, so this can only be done when it is possible to *prove* that a certain condition can never occur. Thus, the generated code should avoid any constructs that would make this proof impossible.

For example, `gcc 2.95.2` seems to be unable to do data flow analysis on records that are larger than 64 bits. Thus, a record that is used for condition checking should not exceed this size.

4.3.4 Annotation

Optimizing compilers perform an extensive code analysis to gather as much information as possible, which is then used to choose the most promising optimization. However, this approach is limited. For example, the code may contain 'fast paths', i.e. paths that are more frequently taken than others because certain conditions occur more often. The compiler could improve performance by optimizing for the common case, but it is very hard to statically determine it from a given C program.

Yet, this information is often known to the programmer because she knows the details of the underlying algorithm, or the intended use of the code. Thus, newer versions of `gcc` provide language extensions which can be used for annotation. For example, the `__builtin_expect` function can be used to indicate the most likely result for an expression (which determines the 'fast path'), and hints can be given with `__builtin_prefetch` as to what data will probably be referenced soon. Finally, the `noreturn` attribute can be applied to a function which is known not to return, either because it contains an infinite loop or because the code is left in another way, e.g. by means of an inline jump instruction.

A stub code generator can use those laguage extensions because it knows the high-level algorithm behind the code it generates. For example, it can assume that exceptions occur rarely and thus are not on the fast path, and it can exploit prefetching because it knows exactly the order in which the data will be referenced.

4.3.5 Record partitioning

Record data types, or `structs`, require a large marshaling effort because, in order to preserve their internal structure, each element must be marshaled individually. However, when client and server use the same representation (in particular, the same memory layout), the record can be treated as an opaque array of bytes and copied *en bloc*, e.g. with `memcpy` or special machine instructions (see Section 4.4.4).

This technique is most efficient when the record only contains data types that do not need any special marshaling (e.g. scalars). However, it can also be applied when this is not the case:

- When there are few special elements, they can be marshaled separately. This requires additional buffer space and increases transfer costs, but this may be outweighed by the more efficient copy operation.[3]
- When the special elements are sparse, the record can be divided into multiple regions containing only 'flat' data types; these can then be copied separately.
- When the exact representation of the record in the endpoints is not important, its members can be reordered so that the special elements are grouped together; the remaining elements can then be copied efficiently.

Some records contain 'memory holes' that are inserted by the compiler for alignment. When the record is marshaled *en bloc*, these holes cannot be removed, and the resulting message is larger. However, our experiments show that element-level marshaling is extremely expensive, so the optimization still works in most cases. Also, when the endpoint representation can be changed, some of the holes can be eliminated by sorting the elements by alignment size.

4.3.6 Extending the calling convention

As described in Section 4.3.2, inlining can be used to eliminate the overhead of a function call. However, `gcc 2.95.2` can only apply this optimization when the function to be inlined appears *before* the call[4] and *in the same translation unit*, i.e. in the same code file or in the associated header files.

This is not an issue for client stubs because they can be put in a header file and included wherever necessary. On the server side, however, there are two calls per method (steps L3 and S2 in the stub model), so full inlining would require both stubs and method implementations to appear in a single file and in a certain order, which is impractical.

The solution is to inline only one of these calls and implement the other as a traditional function call. However, a standard function call would have to follow the calling convention, so the caller context would have to be saved, and parameters would have to be passed via the stack and could not be kept in registers.

For this situation, `gcc` offers the `regparm` attribute, which can be applied to a function and, on IA-32, causes up to three parameters to be passed in registers. Where this is not sufficient, e.g. because more or different registers are required, it is possible to use inline assembler code to implement a custom call.

[3] In some cases, the region of the record that contains the special element can be reused for marshaling; for example, the space for a 32-bit pointer could be used to transfer the 16-bit value it points to

[4] This restriction has been removed in newer versions of `gcc`

4.4 Architecture-specific optimizations

In Section 4.2, we already mentioned that L4-style kernels are available for many hardware architectures, such as IA-32, StrongARM or PowerPC. These architectures differ in many ways, both qualitative (by supporting different features such as predication, data speculation, or SMT) and quantitative (with differently sized caches or instruction and register sets). By exploiting architecture-specific features, code quality can be drastically improved.

This type of optimization is mostly the domain of the main compiler and thus is well beyond the scope of a stub code generator. However, the compiler can only apply optimizations that preserve the low-level semantics of the code, whereas the stub code generator must follow a specification of a much higher level (the interface definition). In doing so, it is free to choose any algorithm it likes. This freedom can be used for optimization, if enough details on the hardware architecture are available.

In this work, the IA-32 architecture is used (see Section 3.2). Its most important features with respect to optimization are: A particularly small register set of only eight general-purpose registers, a high penalty for mispredicted branches, and a high memory latency. See [8] for more details.

4.4.1 Reordering message elements

Due to the high memory latency, it is advisable to avoid copies during server-side unmarshaling, i.e. to use the input values directly from the message buffer (see Sections 4.3.1 and 4.4.3). However, this means that in step S4, when a reply message is created from inout and out values, the inout values may have to be copied to the new buffer.

This additional overhead can be avoided by sorting the message elements by direction, i.e. collecting the inout values on one side of the buffer. Fortunately, the V2 and X0 interfaces allow for separate send and receive buffers, so it is possible to let them overlap and marshal the inout values *in situ*.

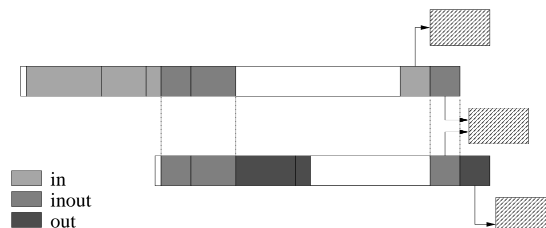

Figure 7: Message layout

Figure 7 shows the resulting message layout. In the request message, the inout values are collected at the end of the message buffer. Thus, the lower part of the buffer, which contains only input values, can simply be dropped during the reply; part of it is overwritten with the header of the reply message. The indirect parts are reordered in a similar fashion; however, it may be necessary to adjust the size of the message buffer in this case because it is used by some kernels to locate the first indirect part.

A similar technique, the in-out heap, has been used by Bourassa and Zahorjan in their LRPC implementation on Mach [5].

4.4.2 Function tables

One important task of the server loop is to distribute incoming requests to the appropriate server stub (step L2). In this process, the method identifier must be checked and then resolved into the stub address. If this is done with a sequence of conditional branches (e.g. a decision tree), a large number of mispredictions is likely to occur, because the method identifiers of successive calls are usually not related.

Another possibility is to use the method identifier as an index into a function table; a single memory access is then sufficient to perform the entire operation. Of course, bounds checking is required; this can be done by enlarging the table to a power of two and filling unused entries with the address of an error handler. Then a simple *and* operation can be used to strip off the irrelevant bits.

`gcc` usually generates function tables for dense `switch` statements[5]. When the method identifiers are sparse, a custom implementation may be preferable; this also has the advantage of giving the user control over the table, so implementations can be changed at runtime.

A tradeoff exists for large function tables; while the call itself performs better, the data cache working set also grows, which may lead to hidden costs.

4.4.3 Server-side stack switch

The limitations of a small register set are most apparent in the case of server stubs. In order to reference input values, a server stub needs to know the address of the message buffer. While it would be possible to use a buffer at a fixed memory location, this is not desirable because, for example, the resulting stub could not be used in a multithreaded server. The alternative is to pass in the buffer address as an argument to the stub, where it will, due to frequent references, occupy a valuable register.

It is possible to save this register by using the stack pointer instead. In a fashion similar to direct stack transfer (see 4.3.1), the stack pointer is moved to the beginning of the message buffer and can subsequently be used to reference message elements *even if* the compiler changes it, e.g. to allocate spill buffers or local variables.

For this approach to work, the compiler must treat the message buffer like an argument, i.e. update its relative position whenever it moves the stack pointer. This can be achieved by defining a record data type which covers the message buffer, and then declaring the stub to have one argument of this type. Consider the following example:

```
struct buffer {
   unsigned fid;
   short a;
   char b;
};

void stub(struct buffer b) {
   ...
}
```

When the `stub` function is entered, the compiler assumes that the caller has pushed a `buffer` record on the stack, so it will reference the element `b.fid` at position `ESP+4`. The generated code can now move the stack pointer to the bottom of the

[5]In `gcc` 3.1, a sequence of conditional branches is generated instead if there are less than `case_values_threshold()` case labels (usually 4) or when the value range is much (i.e. more than 10 times) bigger than the number of values

message buffer and call the `stub` function *without arguments*, which will have the desired effect.

This approach obviously trades performance for code readability because it alters the calling convention. Also, the data cache working set grows if the original stack is used as well.

4.4.4 Special instructions

The extensive instruction set of the IA-32 contains some complex instructions that are rarely or never generated by `gcc`. When the code generator knows that the use of those instructions will improve performance, it can insert them 'by hand' using inline assembly. For example, this approach can be applied to:

- `cmov`, a predicated move. Introduced with the P6, this instruction can be used to avoid conditional branches. `gcc` generates it when the `-march=i686` flag is specified, but not reliably.
- `rep movs`, a fast memory copy. When used on suitably aligned data, it can dramatically improve performance because it operates directly on cache lines.
- `rep scas`, a memory scan which can be used to search for a certain bit pattern. This instruction can be used to determine the length of a string.[6]

The `rep movs` instruction is particularly interesting because it can be used to efficiently move large data structures, e.g. during marshaling.

4.4.5 Cache-aware memory allocation

In recent years, a widening gap between processor speed and memory speed can be observed. This has dramatically increased the importance of caching; on a 400 MHz Pentium II with 100 MHz single-bank SDRAM, the memory latency can exceed 150ns, which is the equivalent of 60 CPU cycles and could be used to execute up to 120 instructions. Therefore, efficient cache usage is crucial for application performance.

Like many modern microprocessors, the Pentium II has separate data and instruction caches; the data cache consists of 256 cache lines of 32 bytes each (8k total capacity) and is 4-way set-associative. Apart from the initial costs of filling the empty cache, two types of faults can occur: *Capacity misses*, which are due to the limited size of the cache, and *conflict misses* due to the limited size of a set.

The code generator can do little about conflict misses in the instruction cache, because it cannot affect the placement of the machine code; however, capacity misses can be reduced by generating smaller stub code (a tradeoff exists; see 4.3.2 and 4.3.3).

Data cache misses can generally be reduced by using cache lines more efficiently, e.g. by bundling local variables to increase the chance that they fit into a single line, or by aligning the message buffers to cache line boundaries. Furthermore, when memory is allocated dynamically, conflict misses can be avoided by ensuring that the greatest common divisor of cache size and allocation size is small.

[6]On newer chips, `rep movs` has been optimized while `rep scas` has not; thus, a tight loop performs better

4.5 Usage-specific optimizations

By default, a stub code generator cannot make any assumptions on how the stubs will be used; function calls might go to any server at any time, use any arguments that the specification allows, and have any result. In practice, however, stub code is seldom used arbitrarily; for example, a call for looking up a person's name in a phone directory might usually be called with a name string of 15-20 characters, but seldom with 3 or 50 characters, although this could definitely happen and should be supported. Or an application might contain a thread implementing a semaphore; the corresponding methods would never be called from outside the address space, the stub code would nevertheless have to support it.

With additional knowledge like this, stub code can be optimized in two ways:

- It can be *specialized* by removing functionality that is never used, or
- It can be *optimized for the common case*; other cases are still supported, but possibly less efficient.

While it is not advisable to include this kind of information with the interface definition (this would spoil the abstraction), it can be provided in a separate file, e.g. in the ACF.

4.5.1 Pass-by-reference

By default, the request message of an RPC must contain *all* the information that will be required to execute the procedure in the server. In particular, this means that all pointers must be dereferenced and the resulting values copied as well; for complex data structures like lists or trees, this may even involve recursion. The resulting request message may be very large.

However, when caller and callee are on the same machine, they may choose to use shared memory for common data; in this case, the pointer itself is meaningful to the callee and can be copied directly. This information can be passed to the stub code generator by marking the method, or the entire interface, as 'local'.

Pass-by-reference is particularly useful for Single-Address-Space Operating Systems like Mungi [20], where one address space is shared by all applications and thus *all* pointers are meaningful. However, for some parameters, copy semantics may still be required.

4.5.2 Domain-local servers

Further optimizations are possible if both caller and callee run in the same address space. This is not unusual; for example, some applications have a separate thread for memory management, a control thread coordinating multiple worker threads, or even a dedicated thread for every critical section.

In addition to using pass-by-reference, the code generator can now exploit the fact that both parties implicitly trust each other; for example, it can omit any security checks. Also, it can use special system calls where available (see Section 4.2.3).

4.5.3 Annotated data types

It was already stated that some parameters do not fully exploit the value range that would be permitted by their data type; some values might appear more frequently than

others, and some might not appear at all. This information can be supplied to the code generator and used for optimization, for example:

- *Maximum size:* In principle, some IDL data types like sequences or strings can be of an arbitrary size. When no additional information is available, the stub code must provide for the worst case, e.g. allocate very large buffers; also, it cannot use IPC variants that only work for limited data sizes, e.g. the virtual registers in V4.

- *Typical size:* Some arguments, e.g. strings or arrays, are typically much shorter than their maximum size. If a typical size is known, the code generator can optimize for the expected case, e.g. choose direct copy instead of indirect parts (see also Section 4.2.1)

- *Partial redundancy:* For some arguments, the type scheme in the IDL may be too coarse. Consider an integer that is always divisible by 1024; the lower 10 bits are implicit and need not be transferred. Also, only a limited region of a large array may be of interest, and the rest can be omitted.

4.5.4 Custom presentation

Although the in-transit representation of a particular data type can be freely chosen (e.g. to eliminate partial redundancy), it must be consistent across all client and server stubs, because it must be ensured that every server can understand request messages from every client. This is different for the data representation in a particular communication *endpoint*, e.g. a specific client task. Because this representation is not visible externally, it may be different for every instance, and can be chosen to suit the application's needs.

For example, there are two common ways to represent a string: Either as a sequence of characters with a trailing zero, or as a variable-sized array with a separate length. The C language (and also the CORBA C language mapping) uses the first representation; for the stub code, however, the second representation is more convenient because it can copy the string more efficiently (e.g. with `memcpy`) when the length is already known.

When both application code and stub code need to know the length of the string, the absurd situation occurs that the length is discarded by one and then immediately retrieved by the other with an expensive call to `strlen`. This can be avoided by choosing the second representation, e.g. by using the `length_is` attribute in DCE.

Another example is the buffer allocation policy, especially for output values. Buffer space can be allocated by the application code or dynamically by the stub code. Dynamic allocation is more efficient for variable-length arguments because the buffer can be allocated after the call, when the actual length is known; however, compared to a static buffer, it has additional overhead for allocation and deallocation, and an additional copy may be required when the data must be at a specific memory location. Again, the problem can be solved by introducing a special attribute and choosing the allocation policy on a case-by-case basis (see also [12]).

Note that unlike the properties in the previous section, the attributes discussed here should not be specified in the interface definition because different instances can make different choices. Instead, they should be kept separately, e.g. in the ACF.

5 IDL^4

To substantiate our claims, we present IDL^4, an IDL compiler for the L4 platform. IDL^4 produces stub code for all three current kernel interfaces (see Section 3.1) and implements most of the optimizations discussed in the previous chapter. This section discusses our requirements and the resulting design decisions; also, it gives a short overview of the IDL^4 architecture.

5.1 Requirements

Apart from maximizing stub code performance, IDL^4 has to meet the following additional requirements:

- **Portability**. By design, part of the compiler will be platform-specific and has to be rewritten for every new target platform. To keep this effort down to a minimum, it must be possible to reuse as much code as possible.
- **Ease of use**. The primary justification for automated stub code generators is convenience. If the tool is complicated, error-prone or requires too much introduction, it will not be used.
- **Maintainability**. Designed for a research environment, the tool is likely to be a 'moving target', i.e. additions or changes will occur frequently. The architecture must support these changes, and it must be possible to verify that they do not break existing code.

5.2 Design decisions

5.2.1 CORBA and DCE

Some stub code generators, e.g. MIG [24], provide a custom interface definition language. This gives them the opportunity to tailor the language to their needs; for example, special data types can be introduced, or the programmer can be given control over certain implementation details. However, there is a danger of introducing limitations or design flaws that cannot be detected until much later; also, users are required to learn an entirely new language.

Therefore, the decision was made to use an existing IDL standard such as CORBA IDL [17] or DCE IDL. However, both languages have their own unique advantages:

- CORBA IDL offers interface inheritance and proper exception handling. Also, it has a high level of abstraction, which makes it independent of a particular programming language such as C.
- DCE IDL allows the specification of attributes, which can be used to control various implementation details; the attributes can be kept in an application configuration file, separate from the interface definition.

Because the syntax of the two IDLs is quite similar, and neither fulfils all of our requirements, we decided to support both. Users can choose either of them as their primary language, and use features from the other where appropriate.

5.2.2 IDL extensions

In L4, the VM primitives *map* and *grant* (see Section 3.1) are handled via IPC, so it is desirable to have support for them in the stub code generator. Therefore, a new data type, `fpage`, was added to the IDL. `fpage` maps to a kernel-specific data type which contains:

- A flexpage, which describes a region in memory
- The send base, which is necessary for the IPC
- Permissions such as read, write or execute
- The transfer mode (map or grant)

These parameters are encapsulated to allow for API-independent applications. An `fpage` can be manipulated by means of helper functions, which are provided for every kernel.

Unlike other IDL compilers for the L4 platform [1, 34], IDL^4 does not support a special data type for indirect parts. This is because we believe that the decision whether or not to use indirect parts should be made by the IDL compiler, as it knows the details of the target platform (see 4.2.1). Also, in contrast to flexpage mapping, indirect parts are not semantically different from memory messages[7], so they should not be distinguished in the interface definition.

A number of DCE-style attributes were added to the IDL:

- `typ_size` and `max_size` can be used to specify the typical and maximum sizes of a variable-length argument, e.g. a string
- `local` can be applied to interfaces that will only be invoked locally, i.e. from within the same domain
- `relevant_bits` may be used on scalars that do not use the full value range; they can then be packed more efficiently
- `nocache`, `l1_only` and `all_caches` designate the caching policy to be used for a flexpage

On user request, the IDL specification was relaxed to allow for bitfields and 'flat' unions. A 'flat' union does not have a discriminator value and may not contain members with special semantics, such as pointers or flexpages. It is always copied with its maximum size.

Similar to Microsoft's MIDL, the `include` and `import` directives were added. The former causes a special header file to be included *in the target code*, while the latter can be used to import data type definitions from ordinary C-style headers. In contrast to MIDL, IDL^4 can also import headers that contain complex definitions or even code; these are simply ignored.

The `any` data type from CORBA IDL was not implemented because it only makes sense with a tagged message format.

[7] Both memory messages and indirect parts have copy semantics; the only difference is the placement of the buffer

5.2.3 Language mapping

Apart from the IDL, a language mapping is required which specifies the relationship between IDL definitions and the stub code that should be generated for them. Again, the choice was whether to create a custom language mapping or to adapt an existing one. We decided to use the CORBA C language mapping [18] because it is a proven concept that has been used in a variety of implementations.

IDL^4 tries to adhere to the standard whenever possible, especially on the client side, but not to the point where performance would suffer. Fortunately, the specification gives implementers some freedom, e.g. by defining certain data types as partially opaque.

5.2.4 Buffer allocation

Variable-length output parameters need special handling because unlike input parameters, their size is often unknown at call time. This situation can be handled in three different ways:

1. The caller can statically allocate buffers in advance, using the maximum argument size it expects, and supply those buffers to the stub.
2. The stub can allocate the buffers dynamically after the call, using the actual argument size.
3. Either caller or stub can provide the buffers on a case-by-case basis.

The first method usually results in a waste of memory due to overprovisioning; however, the second method involves an additional copy because it is not possible to receive the message directly into the buffer. Also, the caller may need the data at a certain virtual address, which causes yet another copy.

The CORBA C language mapping uses dynamic allocation. However, for the reasons stated above, we decided to additionally support static allocation. The user can change the global policy with a compiler flag and use attributes to override it for individual arguments. This is purely a matter of presentation and does not affect the actual interface.

5.3 Architecture

Because flexibility is an important design goal, we have chosen a modular architecture. The compiler consists of four major parts (see Figure 8):

- A *front-end* which reads the interface definition and performs the syntactical and semantical analysis. The result is an abstract, language-independent representation of the interface.
- A *marshaling stage*, which transforms the interface into a series of marshal operations. Then it creates the stub code by mapping the marshal operations to the target language.
- A *kernel-specific part*, which encapsulates the details of the IPC mechanism. It creates the message layout and produces message-related code, e.g. for copying elements to and from the message, and for invoking IPC.
- A *back-end*, which writes the stub code to the output file.

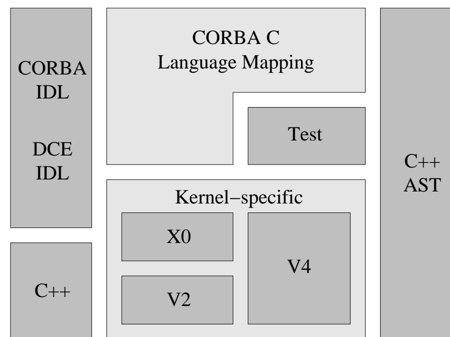

Figure 8: IDL⁴ architecture

These parts interact over well-defined interfaces, so it is possible to replace them independently. Also, multiple instaces of any part can coexist and can be chosen at runtime using compiler flags.

5.3.1 Front end

Currently, two front ends are available: A combined CORBA/DCE IDL front end and a front end for C/C++ code; however, the latter is only used for type import and will not be discussed here.

The IDL front-end consists of scanner, parser and analyzer. The scanner transforms the input file into a stream of tokens, the parser then applies the grammar, and the analyzer resolves any references and checks for consistency. The result is an abstract syntax tree (AST) which contains all the information from the input file, but at a higher level of abstraction; it is designed to be sufficiently general that other IDLs can also be mapped to it. This tree is then passed to the marshaling stage.

5.3.2 Marshaling stage

The marshaling stage is the central part of IDL⁴ because it determines the transfer algorithm that is implemented by the actual stub code. It is specific to a particular language mapping, but independent from the kernel API and the IDL.

Its first step is to build an intermediate representation from the input. For every procedure defined in an interface, it determines how the arguments should be marshaled, and creates a sequence of marshal operations, or *ops*. The ops are no longer specific to a particular data type; for example, they could copy a certain chunk of memory, scan a memory area for the value zero, or map a flexpage. Multiple ops can be created for a single argument (e.g. a record containing a pointer), and they can have dependencies (an array must be unmarshaled *after* its size).

The second step is to create a message layout, which requires interaction with the kernel-specific part. Every op can allocate any number of message elements (FCCs, VCCs and FMCs, see below). However, the placement of the elements or their mapping to specific kernel primitives cannot be influenced at this stage.

In the third step, the actual stub code is produced. Every op is implemented with four pieces of code (marshaling and unmarshaling on both client and server side), which are then combined with message-handling code from the kernel-specific part. The result is another AST in the target language, in this case C or C++. While the

stub code could also be written directly to the output file, using an AST is considerably more flexible.

5.3.3 Kernel-specific part

The kernel-specific part provides an abstraction for the IPC mechanisms in different kernel versions, and on different architectures. To the marshaling stage, it exports a generic interface which supports:

- *Fixed-size copy chunks (FCC)*, whose length is known at compile time
- *Variable-size copy chunks (VCC)*, whose length can only be determined at run-time
- *Flexpage map chunks (FMC)*, which cause a region of virtual memory to be mapped to the receiver

These elements are grouped in *connections*, which usually correspond to individual procedures. A connection describes a set of messages that can be exchanged between client and server, e.g. a request, a reply, and exception reports. For every message, a kernel-specific layout is computed:

- For X0- an V2-style kernels, FCCs are placed in registers if possible, otherwise in a memory message. VCCs are mapped to memory messages or indirect parts, depending on their typical size. The placement of FMCs is fixed by the API; the first one must be in registers, and the others must be at the beginning of the memory message, followed by a delimiter.
- For V4-style kernels, message registers are normally used. If the capacity of the MRs is exceeded, a memory message must be emulated with a StringItem. FCCs are placed in MRs or in the memory message; VCCs may use MRs, the memory message, or a StringItem. For FMCs, a MapItem or GrantItem must be created dynamically.

The marshaling stage needs kernel-specific code at various points, e.g. to send and receive messages, or to access individual message elements. This code is provided by the kernel-specific part. On request, it delivers AST subtrees to the marshaling stage, where they are transformed and integrated with the remaining stub code.

5.3.4 Back end

The back end is responsible for writing the actual output of the compiler, which consists of header files and code files. It accepts an AST from the marshaling stage, traverses it, and simply prints the corresponding code for every tree node. Additionally, it supports modification and transformation of AST subtrees, which is occasionally required.

5.3.5 Regression test

In order to verify and maintain the correctness of the compiler, a regression test is included. When invoked with a special command-line flag, the compiler produces, in addition to the stubs, an entire application which can be run stand-alone on top of the microkernel. This application sets up a server task and invokes all the methods in the interface definition, performing the following checks for each invocation:

- For every direction (in and out), the arguments are initialized with random values and checked after the transfer
- The alignment and size of the message elements is checked
- Every user-defined exception is tested
- Side effects are monitored, e.g. stack overruns, memory leaks and invalid pointer references

The compiler includes an interface definition with some relevant test cases.

6 Evaluation

To evaluate the impact of our optimizations, we compared IDL^4-generated stubs to those from two other stub code generators that also support the L4 platform, DICE [1] and an adapted version of Flick [34].

6.1 Flick

The Flexible IDL Compiler Kit (Flick) was developed by Eide et al. [10] at the University of Utah. Flick is designed as a toolkit of reusable components and can easily be adapted to different IDLs, language mappings, or transport mechanisms. Also, Flick implements a number of powerful optimizations, ranging from efficient memory management to code inlining and fast data copying. The generated stubs are between 2 and 17 times faster than stubs produced by other IDL compilers, e.g. PowerRPC and rpcgen.

Uhlig [34] adapted Flick 1.1 for the L4 platform. The L4 version has a front-end for CORBA IDL and uses the CORBA C language mapping with some L4-specific extensions; for example, special data types for flexpages and indirect parts were introduced. It supports the V2 and X0 versions of the kernel API.

6.2 DICE

The DROPS IDL Compiler (DICE) has been developed by Ronald Aigner [1] as part of the Dresden Real-Time Operating System (DROPS) project. In contrast to Flick, DICE is not an adapted version of a generic IDL compiler, but was designed specifically for the L4 platform; thus, it can take full advantage of the L4 IPC transport mechanism. Additionally, it applies various generic optimizations; for example, message elements may be reordered or packed in order to generate more efficient copy operations.

DICE understands both CORBA IDL and DCE IDL and can generate code for the V2 and X0 versions of the API. In addition to RPCs, DICE supports message passing; however, this feature is not evaluated here.

6.3 Methodology

To initially evaluate stub-code performance, we first determine the round-trip cost for six typical remote procedure calls. To account for indirect costs (e.g. due to cache and TLB pollution) and different call types (domain-local or intra-domain), we measure both the best case (a domain-local call with hot caches) and the worst case, which is a

cross-domain call after a complete cache flush. Additionally, we count the number of instructions executed during each call; this gives an estimate for the complexity of the generated code.

In a second experiment, we determine the marshaling cost *per element* by measuring calls with a variable number of arguments. Again, both the best and worst cases are evaluated; the total cost is then compared with the pure IPC overhead, i.e. the cost for an IPC transferring the same amount of data, but without marshaling or dispatching.

Finally, we evaluate the impact of stub code quality on overall application performance. We cite results from [19], where an earlier version of IDL^4 was tested with the SawMill multi-server operating system. The stub code generator for one component, the physical file system (PFS), was switched from Flick to IDL^4, then performance was compared with an I/O-intensive benchmark.

6.4 Microbenchmarks

In this experiment, we determine the round-trip cost on the Hazelnut kernel for the following typical remote procedure calls:

- `long tiny(in long a)`

  A simple function like this is often used when only a reference or a handle needs to be passed, e.g. to a worker thread in the local domain. The single argument fits easily into a register message and is very simple to marshal; thus, it is possible to generate very efficient stub code for this call.

  For IDL^4, this function was also tested with the `local` keyword, i.e. with a domain-local server (see Section 4.5.2).

- `long small(in short a, in long b, in short c)`

  This function is taken from a device driver interface, where it is used to commit I/O requests. The small number of arguments can still be transferred in registers, but only if they are reordered and packed.

- `long large(in long a, in long b, in long c, in long d, in long e, in long f)`

  The next function, which is equivalent to the `start` function in the SawMill task server interface, requires a few simple marshaling operations and cannot be transferred in registers.

- `void strxfer(in string<119> a, out long b, out long c)`

  Designed to resolve a name into a handle, this function takes a complex argument, a zero-terminated string. The stub code must determine the length of the string by scanning it for the trailing zero, then it must marshal it and copy it into the message buffer.

- `long structxfer(in large_t a, out long b)`

  Stub code for this function must marshal an argument of type `large_t`, which is a large record:

```
struct large_t {
  long a[20];
  short b, c;
```

```
    char d, e[200];
    short f, g[80];
    long h, i;
    char j, k, l[20];
    struct m_tag {
      short n;
      char o;
      long p;
      short q, r;
    } m[2];
  };
```

However, the record does not contain any pointers or special data types such as `fpages`, which would require marshaling; all elements can be copied directly.

- `void arrayxfer([in, length_is(l1)] char *str1, [in, length_is(l2)] char *str2, [in] long l1, [in] long l2);`

  This function copies two large, variable-length arrays from the client to the server. The length of each element is given, so the stub code can easily use indirect parts. We also supplied hints to each compiler, e.g. by using the `refstring` type in Flick or the `ref` attribute in DICE, to indicate that the arrays would be large.

We compiled this IDL specification with the most current version of each IDL compiler, i.e. IDL^4 0.9.0, DICE 1.3, and the L4 version of Flick 1.1; optimizations were enabled where available. The resulting stub code was then added to our test application, which was compiled with `gcc` 2.95.3 in C++ mode (the flags `-O9`, `-march=i686` were given and the frame pointer was not used) and then run on a dual Pentium II/400 system with a current Hazelnut kernel (uniprocessor version, no debugging or tracing enabled). The test application sets up a server and then executes RPCs in a tight loop, while the time stamp counter (TSC) and the performance counters (PMCs) are used to make measurements.

Each call was invoked 200 times; for each invocation, we determined the total number of cycles spent, the number of cycles in kernel mode, and the number of user-mode instructions. To check the quality of our measurements, we compared the 5% and 95% quantiles and found that they typically differed by less than 2%, but never more than 8%. Thus, only average values are given here.

The domain-local variant of `tiny` was evaluated with a user-level implementation of local IPC. This is equivalent to the implementation described in [35], except that it does not contain any kernel fix-up code.

Unfortunately, DICE did not produce correct code for three of the six RPCs. The generated stub for `small` did not marshal one of the parameters at all, and the stub for `arrayxfer` unnecessarily called `strlen()` to determine the length of the string, which was already given. The `structxfer` stub could not be compiled with `gcc` because it seemed to treat arrays as scalars. For this reason, only the remaining three functions were measured with DICE.

Instruction count

Figure 9 shows the results of the instruction count. It can be seen that Flick stubs need two to seven times more instructions than IDL^4 stubs; for DICE, the factor is

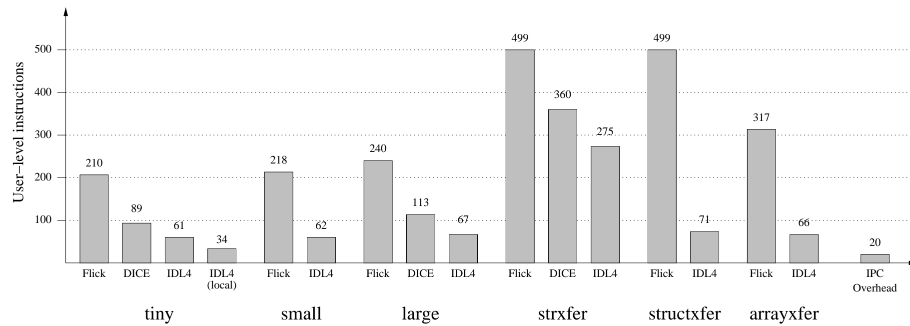

Figure 9: User-level instruction count

approximately 1.5-2. The `strxfer` function is an exception because the length of the string argument is determined with a tight loop; every iteration of this loop is counted separately. The domain-local variant of `tiny` has a lower instruction count than the generic call because it implements fewer security checks and uses local IPC, which results in simpler stub code.

Best case

The cost for an RPC is lowest when a) the call is domain-local, and b) code and data are already in the cache. We measured the best case by repeatedly invoking calls on a domain-local server; as the kernel does neither flush the caches nor the TLB on an intra-domain IPC, all calls except the first one should get maximum benefit from the caches.

The results are shown in Figure 10; the cycles spent in the kernel are shown in a darker color. For the `tiny` call, we see a slightly higher kernel overhead for Flick; the reason is that Flick uses the `int $0x30` call method instead of `sysenter`, which is faster. When local IPC is used, the kernel overhead is much lower because the IPC code is actually executed at user level, so the extra cost of entering and leaving kernel mode is removed.

The results for `small` clearly show the effect of register packing (see Section 4.2.1). Because the IDL^4 stub uses a single 32-bit machine word to transfer the two 16-bit values, it can use a register message, which is considerably less expensive than the memory message used by Flick.

The huge differences for `structxfer` are due to the way records are marshaled by Flick. Whereas IDL^4 uses a simple memory copy to move the entire record to the message buffer, Flick marshals every member individually.

Worst case

For the worst-case measurements, we changed our benchmark in two ways: First, client and server were executed in separate domains, and second, all caches and TLBs were flushed before every call[8].

[8] If the server-side implementation of the RPC has a large cache footprint, the overhead may be even higher because part of the kernel IPC code may be evicted from the caches between request and reply. We did not test this; our RPC implementations always return immediately and do not perform any work.

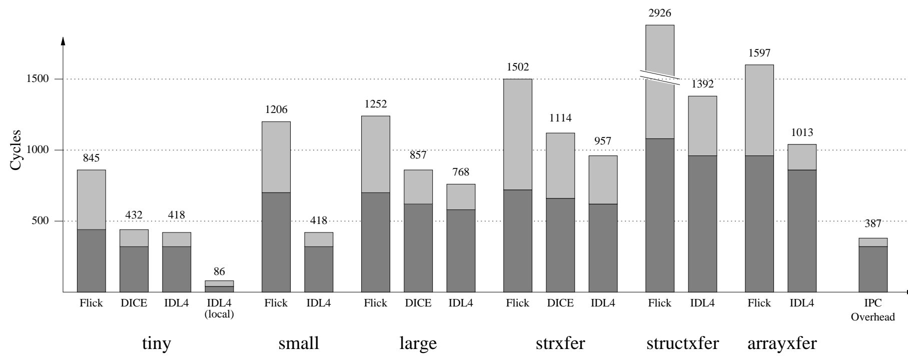

Figure 10: Best-case performance

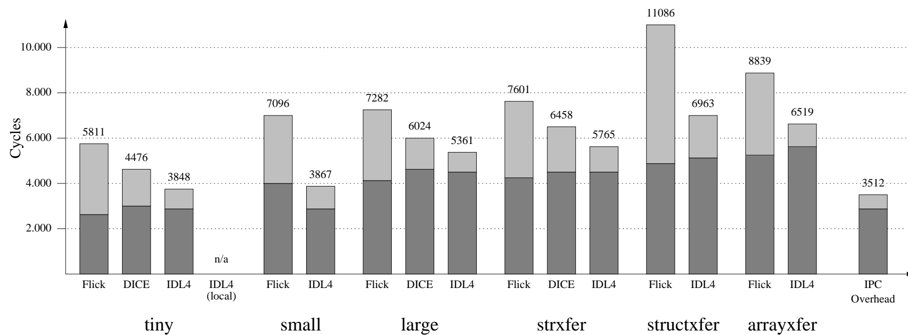

Figure 11: Worst-case performance

Figure 11 shows the results. In contrast to the best-case measurements, `sysenter` has a higher kernel overhead than `int $0x30`. One possible reason is that the cache footprint of the `sysenter` code path is larger.

6.5 Marshaling overhead

In our second experiment, we varied the number of arguments to an RPC in order to determine the marshaling overhead per element. Specifically, we measured round-trip costs for RPCs with 1-100 simple arguments (`long`s, i.e. 32-bit words). Again, best and worst case were tested.

Our results are shown in Figure 12; for comparison, we also included the cost of an IPC of the same size, i.e. the pure setup and transfer cost without marshaling. The jitters, especially in the case of Flick, are caused by cache effects, e.g. conflicts between kernel and user code.

It can be seen that both Flick and DICE have a higher fixed overhead than IDL^4; possible reasons include additional complexity in the server loop and buffer management, e.g. allocation and deallocation of memory messages. However, the per-element marshaling and unmarshaling cost is also higher; this can be determined from the gradient.

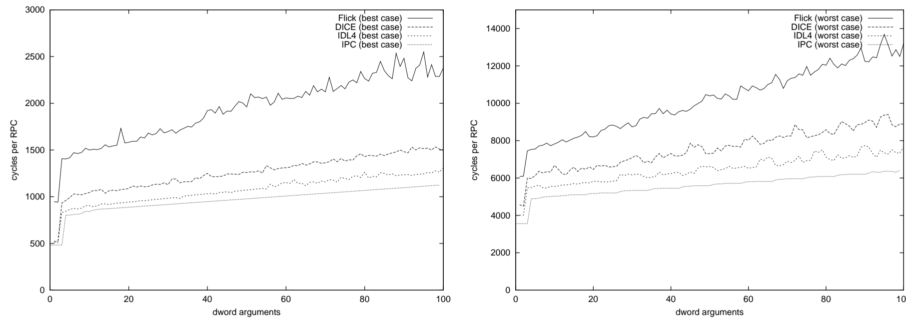

Figure 12: Variable number of arguments

6.6 Application performance

To illustrate the effect of improved stub code on application-level performance, we cite results from earlier work [19]. In this experiment, IDL^4 was used to generate stub code for the SawMill project [14]. SawMill is a Linux-derived multi-server OS where physical file systems (PFS), file and buffer cache, device drivers, network stack, and VM subsystems such as anonymous memory, are all implemented as user-level servers that communicate through L4 IPC and stub code generated by an IDL compiler.

For evaluation, we used both Flick and IDL^4 to generate stubs for the *ext2* PFS, which implements a physical file system that is derived from the *ext2* file system in Linux. The resulting system was run on a Pentium III/500 with 64 MB of main memory and a 540 MB IDE disk drive, and file system performance was measured with the IOzone benchmark [30].

The benchmark begins by writing a file of 64kB, then it reads the contents twice. In the second read phase, all requests can be fulfilled from the page cache; thus, this phase is completely processing bound and not disk bound.

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

Table 1: Overall throughput (± standard deviation) in the IOzone benchmark

We measured re-read throughput where IOzone read 4kB of file data per read request. Table 1 presents the overall performance results as reported by IOzone (ten consecutive iterations). IDL^4 improves the IOzone throughput by approximately 13%; the time for a 4kB read request decreases from 8.0 μs to 7.0 μs. Since re-read costs are dominated by the data copy costs, this result can only be explained by significant improvements in the stub code.

6.7 Discussion

CORBA IDL provides a rich type set, which can be combined in a variety of ways to form a virtually infinite number of RPCs. Also, the performance of generated stub code

depends, among other factors, on the surrounding code, the calling context and the values of the arguments. Therefore, a systematic investigation of stub code performance is hard, and six RPCs may be too small a basis for making statements on performance in general.

Nevertheless, our results show that the IDL^4 stubs were consistently less complex than the stubs generated by Flick and DICE; also, both the fixed overhead and the per-element cost were lower. The IDL^4 stubs we tested were 1.5-3 times faster than the corresponding Flick stubs; for domain-local servers, we even observed speedups of an order of magnitude. As both Flick and DICE are optimizing IDL compilers, this improvement can only be explained by the special platform-specific optimizations we applied in IDL^4.

Also, our experiments with SawMill show that stub-code performance can have a considerable impact on application-level performance. The comparison with IDL^4 stubs shows that in the original system, at least 10% of the achievable end-to-end performance was lost in the generated code, which is unacceptable. With the platform-specific optimizations in IDL^4, this overhead can be reduced significantly.

7 Conclusion

The use of a stub code generator can greatly reduce the effort required to implement a multi-server system. However, stub code has traditionally been highly generic and therefore rather slow, which has prevented it from being used in performance-critical applications.

In this work, we have shown that this limitation can be eliminated by optimizing the stub code for the target platform. We have presented a variety of optimization techniques that can be used to improve performance on Intel's IA-32 platform and the L4 microkernel, and we believe that similar techniques can be applied to other platforms as well.

We have implemented and validated our techniques in IDL^4, our optimizing IDL compiler. IDL^4 applies many of our platform-specific optimizations, yet, due to its modular structure, it can easily be adapted to other platforms. Compared with traditional optimizing IDL compilers, IDL^4 stubs can improve performance by up to an order of magnitude.

Availability

Complete IDL^4 source code and documentation are available from the IDL^4 home page at `http://www.l4ka.org/projects/idl4`.

Acknowledgements

I would like to thank my supervisor, Volkmar Uhlig, for his support and helpful feedback during this work, and also Kevin Elphinstone, Espen Skoglund, Marcus Völp, Horst Wenske, and Stefan Götz for many fruitful discussions. Special thanks go to Uwe Dannowski, who tirelessly answered about a million questions and then did not get impatient when I asked the million-and-first.

References

[1] Ronald Aigner. Development of an IDL compiler for micro-kernel based components. Diploma thesis, Dresden University of Technology, Sep 2001.

[2] Joshua S. Auerbach and James R. Russell. The Concert signature representation: IDL as intermediate language. *ACM SIGPLAN Notices*, 29(8):1–12, Aug 1994.

[3] Brian N. Bershad, Thomas E. Anderson, Edward D. Lazowska, and Henry M. Levy. Lightweight remote procedure call. In *Proceedings of the 12th ACM Symposium on Operating Systems Principles*, pages 102–113. ACM Press, Dec 1989.

[4] Andrew D. Birrell and Bruce Jay Nelson. Implementing remote procedure calls. *ACM Transactions on Computer Systems (TOCS)*, 2(1):39–59, 1984.

[5] Virgil Bourassa and John Zahorjan. Implementing lightweight remote procedure calls in the Mach 3 operating system. Technical Report TR-95-02-01, University of Washington, Department of Computer Science and Engineering, Feb 1995.

[6] David Cheriton. The V distributed system. *Communications of the ACM*, 31(3):314–333, 1988.

[7] David D. Clark and David L. Tennenhouse. Architectural considerations for a new generation of protocols. In *Proceedings of the ACM Symposium on Communications Architectures and Protocols*, pages 200–208. ACM Press, Sep 1990.

[8] Intel Corporation. Intel architecture optimization reference manual. Order No. 245127-001.

[9] Peter Druschel and Larry L. Peterson. Fbufs: a high-bandwidth cross-domain transfer facility. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles*, pages 189–202. ACM Press, Dec 1993.

[10] Eric Eide, Kevin Frei, Bryan Ford, Jay Lepreau, and Gary Lindstrom. Flick: a flexible, optimizing IDL compiler. In *Proceedings of the 1997 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 44–56. ACM Press, Jun 1997.

[11] Bryan Ford, Mike Hibler, and Jay Lepreau. Separating presentation from interface in RPC and IDLs. Technical Report UUCS-95-018, University of Utah, Dec 1994.

[12] Bryan Ford, Mike Hibler, and Jay Lepreau. Using annotated interface definitions to optimize RPC. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, page 232, Dec 1995.

[13] Open Software Foundation. *OSF DCE Application Development Guide*. Prentice Hall, 1993.

[14] Alain Gefflaut, Trent Jaeger, Yoonho Park, Jochen Liedtke, Kevin Elphinstone, Volkmar Uhlig, Jonathon E. Tidswell, Luke Deller, and Lars Reuther. The SawMill multiserver approach. In *Proceedings of the 9th ACM SIGOPS European Workshop*, Sep 2000.

[15] Aniruddha Gokhale and Douglas C. Schmidt. Principles for optimizing CORBA internet inter-ORB protocol performance. In *Proceedings of the 31st Hawaiian International Conference on System Sciences*, Jan 1998.

[16] Aniruddha Gokhale and Douglas C. Schmidt. Techniques for optimizing CORBA middleware for distributed embedded systems. In *Proceedings of INFOCOM '99*, pages 513–521. IEEE, Mar 1999.

[17] Object Management Group. CORBA 2.6 specification. http://www.omg.org/cgi-bin/doc?formal/01-12-35.

[18] Object Management Group. CORBA C language mapping. http://www.omg.org/cgi-bin/doc?formal/99-07-35.

[19] Andreas Haeberlen, Jochen Liedtke, Yoonho Park, Lars Reuther, and Volkmar Uhlig. Stub-code performance is becoming important. In *Proceedings of the First Workshop on Industrial Experiences with Systems Software (WIESS)*, pages 31–38, Oct 2000.

[20] Gernot Heiser, Kevin Elphinstone, Stephen Russell, and Jerry Vochteloo. Mungi: A distributed single address-space operating system. In *Proceedings of the 17th Australasian Computer Science Conference*, pages 271–280, Jan 1994.

[21] Philipp Hoschka and Christian Huitema. Automatic generation of optimized code for marshalling routines. In Manuel Medina and Nathaniel S. Borenstein, editors, *International Working Conference on Upper Layer Protocols, Architectures and Applications*, pages 135–150. Elsevier, Jun 1994.

[22] Hermann Härtig, Michael Hohmuth, Jochen Liedtke, and Sebastian Schönberg. The performance of μ-kernel-based systems. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, pages 66–77. ACM Press, Oct 1997.

[23] David B. Johnson and Willy Zwaenepoel. The Peregrine high-performance RPC system. *Software - Practice and Experience*, 23(2):201–221, 1993.

[24] Michael B. Jones and Richard F. Rashid. Mach and Matchmaker: Kernel and language support for object-oriented distributed systems. In *Proceedings of the 1st Conference on Object-Oriented Programming Systems, Languages and Applications*, pages 67–77. ACM Press, Nov 1986.

[25] Jochen Liedtke. Improving IPC by kernel design. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles*, pages 175–188. ACM Press, Dec 1993.

[26] Jochen Liedtke. Lava nucleus (LN) reference manual. http://www.l4ka.org, Mar 1998.

[27] Jochen Liedtke. L4 nucleus version X reference manual. http://www.l4ka.org, Sep 1999.

[28] Jochen Liedtke and Horst Wenske. Lazy process switching. In *Proceedings of the 8th Workshop on Hot Topics in Operating Systems (HotOS-VIII)*, pages 15–18, May 2001.

[29] Gilles Muller, Eugen-Nicolae Volanschi, and Renaud Marlet. Scaling up partial evaluation for optimizing the Sun commercial RPC protocol. In *Proceedings of the 1997 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 116–125. ACM Press, Jun 1997.

[30] William D. Norcott. The IOzone file system benchmark. Available from http://www.iozone.org/.

[31] Sean O'Malley, Todd Proebsting, and Allen Brady Montz. USC: a universal stub compiler. In *Proceedings of the Conference on Communications Architectures, Protocols and Applications*, pages 295–306. ACM Press, Sep 1994.

[32] Michael D. Schroeder and Michael Burrows. Performance of Firefly RPC. In *Proceedings of the 12th ACM Symposium on Operating Systems Principles*, pages 83–90. ACM Press, 1989.

[33] The L4Ka team. L4 experimental kernel reference manual, version X.2. http://www.l4ka.org, Feb 2002.

[34] Volkmar Uhlig. A micro-kernel-based multiserver file system and development environment. Technical Report TR 21582, IBM T.J. Watson Research Center, Oct 1999.

[35] Horst Wenske. Design and implementation of fast local IPC for the L4 microkernel. Study thesis, University of Karlsruhe, Jul 2002.