A Case Study on the Cost and Benefit of Dynamic RPC Marshalling for Low-Level System Components

Norman Feske
Technische Universität Dresden
feske@os.inf.tu-dresden.de

Published in SIGOPS OSR Special Issue on Secure Small-Kernel Systems, July 2007,
Minor correction of the machine type used for the benchmarks in April 2008

ABSTRACT

Interface definition languages are omnipresent in microkernel-based operating systems for providing a time-tested solution for realizing communication between user-level components. Driven by advancing kernels and application demands, IDL compilers and the generated communication-stub code have become significant contributors to the tool-chain complexity and the size of the trusted-computing base of such systems. This paper examines the performance and the engineering costs of an alternative technique for RPC communication between microkernel servers. Initially intended as an interim solution, the presented approach turned out to be low complex, yet very flexible and fast. These overly positive results turned our interim solution into a proposal for realizing inter-component communication in future microkernel-based operating systems.

1. INTRODUCTION

A typical software stack running on a today's machine is composed of a large number of processes that communicate with each other by using operating-system (OS) mechanisms such as files, shared memory, or sockets. Among these mechanisms, message-based inter-process communication allows for fine-grained interaction between processes, whereas the most popular granularity of a message is a procedure call. There exist numerous tools for translating high-level definitions of a procedural interface to stub code that performs the actual communication using appropriate OS primitives. The different approaches for expressing high-level procedural interfaces (e. g., ONC RPC [10]) converged to de facto standard interface description languages (IDL), which unify the specification of components in distributed environments. An IDL compiler translates a generic IDL interface specification to stub code that implements the actual communication code for the client and the server side of the interface. Whereas the IDL code is platform independent, the generated stub code may be created for any desired programming language and optimized for a particular operating system or machine type.

Given the success of IDL in large distributed systems, the choice for IDL as abstraction for inter-process communication in a microkernel-based operating system seems natural. Hence, in current multi-server OSes, IDL is used as communication abstraction by applications and by system components. These components include even the lowest-level parts of the operating system such as the memory manager, process manager, and low-level device drivers. Such low-level components, however, live under very specific conditions and constraints. All components are implemented in a small set of implementation languages (e. g., C and C++) and are executed on one and the same machine. Language independence and machine-type (e. g., byte order) independence of interface descriptions, two major features of IDL, are not required. In general, IDL supports the use of complex data structures with nested indirections as RPC arguments. For the low-level components of our system however, we observed that such complex messages are not used at all because such data structures require dynamic memory allocation in the stub code. These allocations are unsuitable for low-level system components that require complete control over their memory usage and need to avoid memory leaks under any circumstances.

Given these conditions, the advantage of using an IDL compiler over hand-written communication code comes down to two arguments: convenience and performance. By using IDL, the system developer spares himself the boring, yet bug-prone, development of communication code and custom communication protocols. The performance argument is most notably valid for microkernel-based systems for which low latency of RPC communication is considered as crucial [6]. The IDL compiler can exploit the a-priori knowledge of the message layouts and the specific machine type to optimize the stub code, for example by using native assembly instructions or by considering data alignment in memory.

On the other hand, the first-grade design criteria for a microkernel as well as for low-level components is minimizing complexity and thereby maximizing the robustness of the system as a whole. Based on examples, Section 2 illustrates that an IDL compiler and the generated stub code significantly contribute to the overall tool-chain and system complexity.

In this paper, we strive for eliminating the need for an IDL compiler for low-level components in microkernel-based OSes. We introduce the usage of C++ streams as abstraction for inter-process communication in Sections 3 and 4. This technique leads to extremely simple stub codes that are easily maintainable by hand and that can be statically type checked by the C++ compiler. In Section 5, we discuss the feasibility of the solution by comparing our custom RPC framework to the classical IDL approach with regard to per-

formance, flexibility, utility value, manageability, stub-code complexity, and tool complexity. We base our argumentation on an experiment for which we created a basic multi-server OS [5] including a graphical user interface [11] and interactive applications that we implemented without employing an IDL compiler. As a case study, we compare our gathered experiences against those with our prior existing multi-server OS called DROPS [8], which is based on classic IDL-based RPC communication.

2. COMPLEXITY ON ACCOUNT OF IDL

In Section 1, we claimed that the IDL compiler and the generated stub code make up considerable parts of the tool-chain and security-critical system complexity. To substantiate this claim, let us revisit the source-code complexity of our custom DROPS multi-server OS and put that in relation to the complexity caused on account of IDL. In the following, we use the number of source lines of code (SLOC) [3] as indicator for source-code complexity. Note that using SLOC as complexity measure must be taken with a grain of salt and there exist more expressive metrics [12]. However, SLOC is a very intuitive measure and at least illustrates the magnitude of source code complexity. Furthermore, SLOC expresses well the *amount* of code (the size of the hiding place for potential bugs) a human being has to face on a manual audit.

We base our inspection on a simple OS setup approximating a typical trusted computing base when running security-sensitive applications on top of DROPS. Without taking the microkernel into account, the setup consists of 8 components: a process for starting the other initial components, a log output service, a naming service, the memory manager, the process manager, the program loader, an event service, and a simple GUI server. The human-written source code (mainly written in C) including the ported `uclibc` C library with 9,000 SLOC and ported input drivers with 7,000 SLOC stack up to circa 50,000 SLOC.

The communication between the components involves 14 IDL interfaces with a total of 139 different procedures. From these specifications, our custom IDL compiler generates 32,000 SLOC of communication stub code. Therefore, the amount of generated code makes up to 40% of the overall source code to be ultimately trusted by security-sensitive applications running on top. This relation places emphasis on the crucial role of correct stub code for system security. The correctness of this code, however, is hard and costly to validate.

One could argue that IDL-compiler-generated stub code is just an intermediate format likewise to the intermediate formats that are generated by `gcc` when translating C-source code to binary code. When talking about SLOC, we usually do not refer to code expressed in such an intermediate format but to human-written code only. This argumentation effectively disregards the generated stub code from our complexity analysis. Then however, the complexity of the used code-generation tool comes into question.

DICE [1] is our feature-rich and mature implementation of an IDL compiler that supports multiple back-ends (e. g., L4 version 2, L4 version 4, and socket communication via TCP), multiple platforms (IA32, ARM, AMD64), CORBA types, the inclusion of C headers, generating dependency information, and a rich set of additional configuration options. Its development was driven by the evolving needs of DROPS. As features generally had been added on user requests, almost all of DICE's functionality is actually used. In its current stage, the C++ implementation of DICE comprises circa 48,000 SLOC. For our example scenario, the complexity of the stub code generator is in the same order of magnitude as the actual human-written trusted operating-system code.

As illustrated in [13], faulty compilers and tools may introduce critical security vulnerabilities into the generated binary code. Thus, tool-chain complexity is critical. If comparing DICE to the overall tool-chain including the GNU C++ compiler and binutils, DICE's complexity appears negligible at the first glance but there are two characteristic differences between DICE and the standard parts of the tool chain. Firstly, when using standard tools such as `gcc`, `ld`, and `as` for compiling human-written code, these tools cannot be eliminated and are a mandatory part of the tool chain anyway. Secondly, those tools are developed and maintained by a huge community and are heavily used by a considerable part of the earth population each day. By our personal experience, this extremely high exposure leads to an overall low bug rate and high confidence. On the other hand, our custom IDL compiler is developed and maintained by only one person and is exposed to only a low (less than 50) number of regular users.

Regardless of how we consider the complexity overhead imposed by the use of IDL, the stub complexity or the effect on the tool-chain complexity, the overhead is significant. In the next section, we present our approach to minimize this overhead.

3. DYNAMIC RPC MARSHALLING

We use C++ as implementation language, which facilitates the use of *streams* as concept for handling input and output rather than relying on C library functions such as `scanf` and `printf`. Streams are objects that provide *insertion* (<<) and *extraction* (>>) operators. The basic mode of operation of our RPC framework is based on C++ streams and best illustrated by examples: Sending a one-way message with two arguments to a server involves the two steps of transforming the arguments to an array of bytes (*message*) and invoking the OS-communication mechanism with the message as argument. The step of creating the message from a set of arguments is called *marshalling*.

```
Ipc_ostream sender(dst, &snd_buf);
sender << a << b << IPC_SEND;
```

The object `sender` is an output stream that is initialized with a communication endpoint (`dst`) and a message buffer (`snd_buf`). Depending on the underlying OS mechanism, `dst` may be a thread ID, a port number, the name of a pipe, a capability, or any other communication address. For sending the message, we sequentially insert both arguments into the stream to transform the arguments to a message (first step) and finally invoke the actual OS-communication

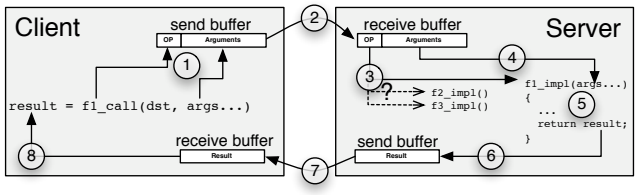

Figure 1: Illustration of an RPC call. (1) Client marshals arguments to form a message. The first part of the message is a function opcode. (2) Client sends message via the kernel to the server and blocks for the result. (3) Server dispatches request according to the opcode stored in the message. (4) Server unmarshalls function arguments, (5) executes function, and (6) marshals the produced results. (7) Server sends result message to the client. (8) Client unmarshals results.

mechanism by inserting the special object `IPC_SEND` (second step).

The counterpart on the receiver side looks similar:

```
int a, b;
Ipc_istream receiver(&rcv_buf);
receiver >> IPC_WAIT >> a >> b;
```

For creating the `receiver` input stream object, we specify a receive message buffer as argument that can hold one incoming message. By extracting the special object `IPC_WAIT` from the receiver, we block for a new message to be stored into `rcv_buf`. After returning from the blocking receive operation, we use the extraction operator to *unmarshal* the message argument by argument.

3.1 Remote Procedure Calls

We expanded this simple mechanism to support full RPC semantics. A complete RPC includes the steps as displayed in Figure 1. For performing RPC, we introduce the stream classes `Ipc_client` and `Ipc_server`, which act as input streams as well as output streams. In the following example, a client performs a call with two arguments and receives one result value:

```
Ipc_client client(dst, &snd_buf, &rcv_buf);
int result;
client << OPCODE_FUNC1 << 1 << 2
       << IPC_CALL >> result;
```

The first argument is a constant that references one among many server functions. It is followed by the actual server-function arguments. All arguments are marshalled into the `snd_buf`. When inserting the special object `IPC_CALL` into the `client` stream, the client blocks for the result of the RPC. After receiving the result message in `rcv_buf`, the RPC results can be sequentially unmarshalled via the extraction operator. Note that `rcv_buf` and `snd_buf` may use the same backing store as both buffers are used interleaved.

The corresponding server-dispatch function looks as follows:

```
Ipc_server server(&snd_buf, &rcv_buf);
while (1) {
  int opcode;
  server >> IPC_REPLY_WAIT >> opcode;
  switch (opcode) {
    case OPCODE_FUNC1:
    {
      int a, b;
      server >> a >> b;
      server << func1(a, b);
      break;
    }
    ..
  }
}
```

The special object `IPC_REPLY_WAIT` replies to the request of the previous server-loop iteration with the message stored in `snd_buf` (ignored for the first iteration) and then waits for an incoming RPC request to be received in `rcv_buf`. By convention, the first message argument contains the opcode to identify the server function to handle the request. After extracting the opcode from the `server` stream, we branch into a server-function-specific wrapper that reads the function arguments, calls the actual server function, and inserts the function result into the `server` stream. The result message is to be delivered at the beginning of the next server-loop iteration. The two-stage argument-message parsing (opcode to select server function, reading server-function arguments) is simply done by subsequent extraction operations.

3.2 Marshalling

The previous examples used only integer arguments to compose messages. Of course, the marshalling is not limited to that particular type but allows for the insertion of any self-contained (not having pointers to other objects), fixed-sized object type. The following template does the trick:

```
template <typename T>
Ipc_client &operator << (T value) {

  /* check for buffer overrun */
  assert(write_offset + sizeof(T)
         < sndbuf_size);

  /* write value of type T into buffer */
  *(T *)(&sndbuf[write_offset]) = value;

  /* increment write offset */
  write_offset += sizeof(T);

  return *this;
}
```

The C++ compiler automatically instantiates the template for each type that is used for an argument to the insertion operator. Furthermore, template specializations allow even for the marshalling of non-self-contained object types such as lists by implementing type-specific stream-operator semantics. The unmarshalling via the extraction operator is done in an analogous way.

Apart from fixed-size objects, typical RPCs contain variable-sized arguments, most prominently character strings. We support such arguments by providing a specialized insertion

operator for the type `Buffer`. Such a buffer consists of an address and a size and thus, describes an arbitrary memory region. The insertion operator for the `Buffer` object simply copies the corresponding memory into the message buffer. For convenience, a `Buffer` object can be constructed with the address of a null-terminated string as argument where the buffer size gets determined via `strlen`:

```
client << OPCODE_WRITE
       << Buffer("very convenient")
       << IPC_CALL;
```

When extracting a `Buffer` object from a received message, the `Buffer` object references the corresponding range within the received message.

The same methodology can be applied for configuring RPCs. If the underlying communication mechanism provides support for communication timeouts, such a timeout can be configured by inserting a `Timeout` object that gets handled by a specialized insertion operator performing the required configuration. On microkernel-based systems, IPC messages are further used to delegate resources (e. g., memory pages) and rights between processes. The handling of such non-plain-data messages is another use case that can be handled by specialized stream operators.

4. IMPLEMENTATION

Our custom experimental OS is targeted at running on top of a microkernel. Therefore, we conducted our first experiments regarding our RPC framework on the L4/Fiasco kernel that provides the IPC interface of L4 version 2. This interface features two variants of synchronous IPC: *Short IPC* transfers two 32bit words directly via CPU registers whereas *long IPC* copies the (potentially large) message payload from a send buffer stored in memory to the receiver of the message. Because short IPC is much faster than long IPC, our framework distinguishes at runtime both cases for IPC calls and IPC reply messages.

Apart from transmitting plain data messages, the L4 IPC interface is used for delegating memory mappings to let processes establish shared memory. Unfortunately, the layout of L4-version-2 IPC messages that include flexpage mappings unreasonably complicated the implementation into our framework. Concluding from the observation that memory-mapping IPCs are only used in the fashion of sending exactly one mapping via a short IPC, we decided to handle this special case separately and keep the general IPC marshalling code clean.

In the course of the work on the RPC interfaces for our experimental multi-server OS, apart from the mentioned flexpage support, we have not encountered RPC semantics that are difficult to realize with dynamic RPC marshalling. We take this as an indicator for the functional feasibility of our approach regarding low-level system components.

5. EVALUATION

To evaluate our dynamic-marshalling approach, we compare our implementation against the DICE IDL compiler as mentioned in Section 2. In the following, we analyze performance, utility value, and overall manageability as the distinctive properties of both solutions.

5.1 Performance

Since the rise of second-generation microkernels with L4 [9] as the prime example, the crucial role of high-performance IPC for achieving good overall system performance is regarded as a fundamental realization. Hence, kernels of the L4 microkernel family are extremely optimized to minimize the costs of IPC operations. To be in the line of the optimized IPC performance of the kernel, IDL compilers became increasingly more powerful in generating speed-optimized stub code. As a particular case, the transition from the Flick [4] IDL compiler to the more sophisticated IDL4 IDL compiler is described in [7]. The DICE IDL compiler was created as a flexible and speed-optimizing counterpart of IDL4 for the L4/Fiasco microkernel.

Our presumption in comparing the dynamic RPC marshalling approach to DICE-generated stub code was not optimistic because, in contrast to DICE, dynamic marshalling cannot take a-priori knowledge about the message buffer layout into account and thus, cannot perform optimizations to the same degree. Furthermore, we were uncertain about the actual costs of the C++ stream operators. When using dynamic marshalling for our research OS experiment, we were willing to trade IPC-performance for the prospect of having a practical and low-complex RPC solution immediately available. Still, the magnitude of performance degradation is of interest to judge if the advantage of the simplicity of our approach outweighs its performance penalty.

To estimate the performance in a realistic setting, we picked five characteristic RPC function signatures from our existing interfaces. Each function provides one integer return value, which is typically used to indicate the success or failure of the function call.

`f1()` has no arguments. Such a signature is typically used for sending notification messages. Because of the use of such messages for interrupt notifications at high rates, L4 provides an optimization for such *short messages* and transfers the message directly via general-purpose CPU registers.

`f2(int)` has a similar usage pattern as `f1` but is used when more contextual information must be transmitted. It also profits from L4's short-message optimization. Other use cases are `close` calls for sessions or files where the argument is a session or file ID.

`f3(3 x [out] int *)` returns three integer values and is used to request properties of a server. Such requests usually happen at a low rate.

`f4(5 x int)` is a typical function signature provided by a service. For example, a GUI service provides a `refresh` call taking a window ID, a (x,y) position,

and a (width, height) size as arguments. Another example is a copy function of a memory management service, which takes references to memory ranges (addresses, offsets) and flags as arguments.

`f5(8 x int)` specifies eight integer arguments to a function of a stateless server. For such servers, all contextual information must be provided with each call. For example, for a graphics operation, all drawing attributes and coordinates must be provided.

`f6(char *)` transmits a variable-sized, null-terminated string. At compile time, only a maximum string length is known. The stub code must determine the actual string length at runtime. Such an argument is usually combined with a couple of integer arguments but we test it separately to get a clearer profile. For our test case, we transfer a 36 character string representing a typical pathname or a script command.

For this set of functions, we created both IDL-based communication stubs and hand-written stub-code using dynamic marshalling. We performed our first tests on a 1.70GHz Intel P4 Celeron Willamette CPU with a cache of 128KB and 256MB of memory.

Function	Mechanism	Stub	RPC
f1()	IDL stub	678	3797
	Dyn RPC	1130	4270
f2(int)	IDL stub	676	3796
	Dyn RPC	1154	4293
f3(3 x [out] int *)	IDL stub	784	5897
	Dyn RPC	1622	6407
f4(5 x int)	IDL stub	766	5947
	Dyn RPC	1785	6425
f5(8 x int)	IDL stub	809	5808
	Dyn RPC	2108	6763
f6(char *string)	IDL stub	1232	6101
	Dyn RPC	2072	6756

Table 1: Number of clock cycles required to perform RPC calls of the test functions via DICE-generated stub codes on the one hand and dynamic marshalling on the other hand. The *Stub* values correspond to the clock cycles spent in the stub code path. The *RPC* values are the overall costs including the kernel IPC code path.

Table 1 compares the performance of dynamic marshalling against DICE-generated stub code. Depending on the signature of the function, the stub code path of dynamic marshalling is 20%-100% slower than the corresponding code as generated by DICE. The relative differences of the overall costs including the required IPC operation of the kernel however, are much smaller because the overall RPC performance is dominated by the IPC operation of the L4/Fiasco microkernel. Because client and server reside in different address spaces, each RPC involves two costly context switches. Still, for this particular kernel, dynamic marshalling leads to a performance degradation of more than 10%. For other kernel implementations that are even more optimized for IPC performance, the negative impact of dynamic marshalling on the overall performance may be even higher. The inferior performance of dynamic marshalling is primarily caused by the stream operations that are treated by `gcc` as function calls. For example, for the stub function

```
int f4(int a, int b, int c, int d, int e)
{
  return ipc_client << 4 << a << b << c << d
                    << e << IPC_CALL;
}
```

the `gcc` compiler generates the following assembly code:

```
push   %ebp
mov    %esp,%ebp
sub    $0x8,%esp
movl   $0x4,0x4(%esp)
movl   $0xbb6020,(%esp)
call   insert_int
mov    %eax,%edx
mov    0x8(%ebp),%eax
mov    %eax,0x4(%esp)
mov    %edx,(%esp)
call   insert_int
 ...
mov    %eax,%edx
mov    0x18(%ebp),%eax
mov    %eax,0x4(%esp)
mov    %edx,(%esp)
call   insert_IPC_CALL
movl   $0x0,0x4(%esp)
mov    %eax,(%esp)
call   int_operator
mov    %eax,(%esp)
call   b99084
leave
ret
```

Each argument involves a costly call operation. Furthermore, stack operations are needed to retrieve the `f4` arguments `a` to `e` and push their values to the stack for calling the insertion operator.

As hinted by the assembly code, the performance of the stub code drastically changes when enabling the inlining optimizations of the compiler. With inlining enabled, the complete marshalling code for a call of `f4` with the arguments `1`, `2`, `3`, `4`, `5` gets translated to the following code:

```
mov    write_offset,%edx
mov    message_buffer,%edi
movl   $0x4,(%edx,%edi,1)
mov    write_offset,%ebx
add    $0x4,%ebx
mov    %ebx,write_offset
movl   $0x1,(%ebx,%edi,1)
mov    write_offset,%ecx
add    $0x4,%ecx
mov    %ecx,write_offset
movl   $0x2,(%ecx,%edi,1)
mov    write_offset,%edx
add    $0x4,%edx
mov    %edx,write_offset
movl   $0x3,(%edx,%edi,1)
mov    write_offset,%ebx
mov    $0x1,%edx
```

```
add     $0x4,%ebx
mov     %ebx,write_offset
movl    $0x4,(%ebx,%edi,1)
mov     write_offset,%ecx
add     $0x4,%ecx
mov     %ecx,write_offset
movl    $0x5,(%ecx,%edi,1)
addl    $0x4,write_offset
mov     %edx,0x4(%esp)
movl    $0xbb5020,(%esp)
call    insert_IPC_CALL
```

The RPC opcode (4) and all arguments get directly written into the message buffer and for each step, the write offset gets incremented by 4 (`sizeof(int)`). Thus, for regular RPCs with fixed arguments specified in the source code, the inlining of stream operations as performed by the C++ compiler translates dynamic marshalling C++ code effectively to static marshalling at the assembly level.

Function	Mechanism	Stub	RPC
f1()	IDL stub	542	3486
	Dyn RPC	585	3387
f2(int)	IDL stub	599	3514
	Dyn RPC	627	3428
f3(3 x [out] int *)	IDL stub	584	5454
	Dyn RPC	559	4898
f4(5 x int)	IDL stub	636	5491
	Dyn RPC	771	5086
f5(8 x int)	IDL stub	624	5236
	Dyn RPC	738	5095
f6(char *string)	IDL stub	967	5929
	Dyn RPC	982	5301

Table 2: Performance comparison of dynamic marshalling against DICE-generated stub code when compiled with the inlining optimizations (`-O3`) of `gcc-3.3`.

Table 2 shows the performance of the test functions with enabled inlining optimizations. Both the dynamic marshalling as well as the generated stub code significantly profit from these optimizations. A surprising observation is that the dynamic marshalling code gains so much from function inlining that it outperforms DICE-generated code in some cases. The relatively rigid code as generated by DICE leaves less potential for automated compiler optimization than the high-level C++ stream code. Another interesting observation is that the performance of the IPC code path in the kernel seems to depend on the user-level stub code. Both the dynamic marshalling code and the DICE-generated stub code use the same IPC kernel operations with the same arguments. For example, for function `f3`, the execution time of the dynamic marshalling code is only 25 cycles faster than the corresponding DICE-generated code. The overall performance difference including the IPC kernel operations, however, are 556 cycles.

Table 3 examines the costs for the individual phases of the RPC call of `f3`. The performance gain of the dynamic marshalling code is mainly attributed to the performance difference of the IPC reply kernel operation. Since the kernel operations are issued with the same arguments, these differences must originate from indirect effects such as different cache usage patterns of both code paths [1]. These effects make the significance of such microbenchmarks for predicting system performance questionable. This becomes even more evident when conducting the same benchmarks on platforms with different microarchitectures.

f3(3 x [out] int *)	IDL stubs	Dyn RPC
Client marshal arguments	92	117
IPC request (client → server)	1840	1781
Server unmarshal arguments	213	186
Server marshal results	96	120
IPC reply (server → client)	3030	2558
Client unmarshal results	183	136
(Un-)Marshalling costs	584	559
Overall costs	5454	4898

Table 3: Decomposition of the RPC costs for function `f3`.

Table 4 compares the performance of the simplest of all test functions `f1` on four different microarchitectures. Taking the *client-marshal-args* code path as a prime example, the dynamic marshalling code path performs worse (P4 Willamette), equal (P4 Prescott), or better (AMD Opteron) compared to corresponding DICE-generated stub code. Another interesting observation is that the simple *client-unmarshal-args* code path of the DICE-generated stub executed on the P4 requires more than 300 clock cycles more than the dynamic marshalling equivalent. By closely examining both corresponding code paths and successively moving our measurement sensor code (based on `rdtsc` [2]), it seems that, under certain circumstances, a single `ret` instruction after leaving the kernel can eat 300 cycles. In fact, we observed such a 300-cycles penalty for other functions of both dynamic marshalling and DICE-generated stub code on the P4.

With regard to comparing both RPC approaches on the examined test platforms, the differences of DICE-generated stubs and dynamic marshalling have only a marginal effect on the overall RPC performance. For some test functions, we even observed a performance gain in using dynamic marshalling compared with DICE-generated code.

5.2 Utility value

Apart from achieving fast communication code, the usage of IDL compilers is driven by a convenience benefit for its users, who are relieved from writing communication code by hand. In contrast, our dynamic marshalling approach suggests to involve more manual work. For example, the programmer

[1] We prepended one additional execution of the RPC code path to the front of the benchmark loop to perform the actual measurement with warm caches. We suspected that the cache usage patterns of both stub-code variants conflict differently with the cache lines used by the kernel. However, by counting cache misses using performance counters, we observed that in neither case, cache misses happened.

[2] To estimate to systematic error caused by the `rdtsc`-based benchmarking code itself, we repeated the benchmark on the P4 with only the two outer measurement points enclosing the complete RPC call and excluded the five inner measurement points. The difference to the original measurement lies in the range of a dozen clock cycles indicating that the `rdtsc` instructions do not distort the measurements too much.

	CPU	IDL stubs	Dyn RPC
Client marshal args	P4(W)	89	117
	P4(P)	99	99
	AMD	63	40
	Core	153	148
IPC request	P4(W)	1368	1284
	P4(P)	1603	1507
	AMD	568	539
	Core	1083	1023
Server unmarshal args	P4(W)	171	208
	P4(P)	111	104
	AMD	57	143
	Core	217	226
Server marshal results	P4(W)	96	128
	P4(P)	167	170
	AMD	52	43
	Core	120	152
IPC reply	P4(W)	1576	1518
	P4(P)	2436	2245
	AMD	617	591
	Core	1208	1166
Client unmarshal results	P4(W)	186	132
	P4(P)	463	100
	AMD	140	73
	Core	258	196
(Un-)Marshalling costs	P4(W)	542	585
	P4(P)	840	473
	AMD	312	299
	Core	748	722
Overall costs	P4(W)	3486	3387
	P4(P)	4879	4225
	AMD	1497	1429
	Core	3039	2911

Table 4: RPC performance comparison of the function `f1` for the microarchitectures P4(W) *(Celeron Willamette at 1,703 MHz)*, P4(P) *(Celeron D Prescott at 2,933 MHz)*, AMD *(Opteron at 1,995 MHz)*, Core *(Core 2 Merom at 2663 MHz)*.

has to manage the allocation of message buffers, the stub code, and the server-loop manually. On the other hand, we experienced that RPC interfaces are, in contrast to the inner life of components, rather static. To substantiate this claim, we analyzed the source-code revisions of the files referring to the DROPS-based scenario mentioned in Section 2.

During the time frame of four years (2003-03-06 to 2007-03-06), the source code underwent 1350 revisions with overall 2838 file changes. From these revisions, only 53 revisions semantically changed the RPC interfaces by modifying IDL files. Of course, the number of source-code revisions does not provide an accurate measure of engineering costs but we can take these values at least as a hint. Assuming such a correlation, the ratio of RPC-specific revisions to implementation-specific revisions clearly supports our claim. If we had used dynamic marshalling involving manual adaptation of communication code, the amount of manual work would have been negligible compared to the non-RPC-related development work. Even more surprising, the mere usage of the IDL compiler implied substantial collateral costs. With 252 file changes in 53 revisions, circa 9% of overall file changes fall under the category "Adaption to DICE's changes" and were needed because DICE itself is a moving target. With the progress of the project and with the growing demands of the used RPC interfaces, DICE incorporated new features, changed the set of supported IDL elements, and even revised semantics of IDL keywords. The ratio of revisions to file changes indicates that often, changed behaviour of DICE implicated modifications in a lot of different source files.

Another observation is that the generated client stubs are rarely used directly. For the majority of services, there exist client libraries, which wrap the generated stub code and, in turn, provide more convenient client APIs that *completely* hide communication. Consequently, even in the presence of the IDL compiler, a changing RPC interface requires manual adaptation of the client-side support code.

Our experiences with dynamic marshalling are quite young, and were gathered during only one year. Yet, we can report that the required effort in maintaining the communication code was so low that we have not developed a desire for tool support. As the dynamic marshalling technique does not employ an IDL compiler that checks the consistency of types and opcodes between the client and server stub codes at IDL-compile time, we maintained type safety by declaring the opcodes and function prototypes of each interface in a dedicated header file that we included from both the client and the server stub-code implementations. Therefore, the C++ compiler checks the adherence of the stub code on both sides to the interface declaration at compile time. The primitives described in Section 3 turned out to be very simple, yet flexible and convenient. As an illustrative example, an RPC with an `argc-argv`-like interface can trivially be constructed via these primitives:

```
sender << argc;
for (int i = 0; i < argc; i++)
  sender << Buffer(argv[i]);
sender << IPC_SEND;
```

In contrast, an IDL compiler would require explicit support of such an interface as a special feature. As a (subjective) convenience aspect, using the implementation language that the programmer is proficient with also for working on RPC interfaces is an advantage because the programmer can fully exploit its well-understood implementation language instead of switching his mind back and forth between implementation language and IDL with a lot of special keywords to remember. Designing an RPC interface and creating the corresponding client library are consolidated into one task.

5.3 Manageability

Besides the utility value of the RPC solutions for their users, we also identified the manageability of the solutions as a distinctive property directly related to engineering costs. With manageability, we refer to the complexity of the solution, flexibility with regard to providing feature support, and portability.

In Section 2, we motivated our dynamic RPC marshalling approach with the goal of avoiding the tremendous complex-

ity of the IDL compiler (DICE comprises circa 48,000 C++ SLOC, Magpie [2] comprises circa 42,000 Python SLOC) and the generated stub code. In contrast, dynamic marshalling does not require custom tool support and therefore, eliminates 48,000 SLOC from our tool chain. Our RPC framework providing the dynamic marshalling functionality is implemented in less than 500 SLOC. The costs of maintaining such a simple solution are orders of magnitude lower than the costs of maintaining a complex IDL compiler.

For the presented scenario based on our existing software stack, the 32,000 lines of generated stub code for 14 IDL interfaces with 139 remote procedures make up 40% of our overall source code. Our newly implemented experimental multi-server OS provides similar basic functionality such as loading of ELF binaries, memory management, a basic GUI, and some simple interactive demo applications. Because of its more simplistic nature, its 16 RPC interfaces feature only 42 remote procedures, which we realized with our dynamic marshalling approach. Out of the 17,000 lines of human-written code, the RPC interfaces make up less than 10% (1,400 SLOC). Although both implementations are not fully comparable with regard to the implemented functionality, the observed ratios of RPC-related code to non-RPC-related code strongly indicate that dynamic marshalling yields to significantly reduced stub-code complexity.

The DICE IDL compiler provides a multitude of features to support the mechanisms of the underlying kernel. For example, DICE provides support for timeouts, can transmit indirect strings, and supports the mapping of memory regions. As mentioned in Section 4, the message-descriptor format of the L4-version-2 kernel API imposed difficulties in supporting some kernel features such as flexpage mappings into our dynamic marshalling framework. Instead of integrating these (rarely used) features into the generic framework, we implemented them as special cases outside the framework. However, in a second experiment, another engineer of our group re-evaluated the dynamic-marshalling approach by creating an alternative framework. His implementation provides support for all L4-version-2 kernel features, yet does not exceed a complexity of 500 SLOC. Given a talented developer, the dynamic marshalling approach does not seem to lag behind an IDL compiler with regard to supported features.

DICE supports multiple back ends enabling one IDL interface to be used to generate communication stub code for both L4 IPC and socket communication. The support of those multiple platforms can greatly shorten development cycles when firstly implementing functionality on Linux using DICE's socket back-end and later porting the result to the L4 platform. The low complexity of our dynamic marshalling framework, however, suggests that it can be as easily ported or re-implemented for different platforms. In fact, porting the framework from L4 IPC to socket communication involved only two evenings of work.

6. CONCLUSIONS

Employing IDLs is the time-tested solution for realizing RPC communication in distributed systems up to large-scale heterogeneous networks. IDL-based frameworks such as CORBA enable network-transparent inter-component communication independent from implementation languages, OS platforms, and computer hardware.

In microkernel-based OSs as a flavour of distributed systems, IDL compilers do a respectable job of making low-level microkernel mechanisms accessible to user-land developers while preserving optimized IPC performance of modern microkernels. The major strengths of an IDL as being a platform and implementation-language abstraction, however, remain unused when building low-level components for a microkernel-based OS. Furthermore, this specific application domain on the one hand follows the fundamental principle of enhancing robustness and security by minimizing the complexity of critical software. On the other hand, the usage of IDL contributes substantially to the critical system complexity. In our work, we elaborated on dynamic marshalling as an alternative approach to implement RPC communication between low-level components of microkernel-based OSes. Assuming to pay a performance penalty when compared to IDL-compiled communication code, we originally intended to use dynamic marshalling just as a pragmatic interim solution during the experiments with our research OS prototype to enable us to postpone the costly development or adaption of an IDL compiler for our platform. To our delight, we observed that the efficiency of our approach is in fact on par with communication stubs as generated by the DICE IDL compiler. We also experienced that our simple dynamic-marshalling framework provides us with all the flexibility that we needed, avoids complex code-generation magic, and is still convenient to use. The biggest advantage, however, is the significant complexity reduction of the communication-stub code and the tool chain, and thereby our increased confidence in the overall robustness of our software.

7. ACKNOWLEDGEMENTS

I want to thank Christian Helmuth for constructing our experimental research OS together with me and Alexander Warg for our valuable discussions and for re-evaluating the approach by the means of his alternative implementation. Furthermore, I am grateful to Ronald Aigner for sharing his expertise in IDL-based communication, for open discussions, and for validating the measurements. The final version of the paper was improved thanks to the very helpful advice from the reviewers of the ACM OSR Special Issue on Secure Small-Kernel Systems. I especially thank Michael Hohmuth for setting up the special issue and for his greatly appreciated suggestions.

This work was conducted as part of the ROBIN project funded by the European Union.

8. REFERENCES

[1] DICE website. URL: http://os.inf.tu-dresden.de/dice.

[2] Magpie website. URL: http://www.ertos.nicta.com.au/software/kenge/magpie/latest/.

[3] SLOCCount website. URL: http://www.dwheeler.com/sloccount/.

[4] E. Eide, K. Frei, B. Ford, J. Lepreau, and G. Lindstrom. Flick: A flexible, optimizing idl compiler. In *ACM SIGPLAN '97 (PLDI)*, Las Vegas,

NV, 1997.

[5] N. Feske and C. Helmuth. Design of the Bastei OS architecture. Technical Report TUD-FI06-07-Dezember-2006, TU Dresden, 2006.

[6] A. Gefflaut, T. Jaeger, Y. Park, J. Liedtke, K. Elphinstone, V. Uhlig, J. Tidswell, L. Deller, and L. Reuther. The SawMill multiserver approach. In *ACM SIGOPS European Workshop 9/00*, 2000.

[7] A. Haeberlen, J. Liedtke, Y. Park, L. Reuther, and V. Uhlig. Stub-code performance is becoming important. In *Proceedings of the First Workshop on Industrial Experiences with Systems Software (WIESS)*, San Diego, USA, 2000.

[8] H. Härtig, R. Baumgartl, M. Borriss, C.-J. Hamann, M. Hohmuth, F. Mehnert, L. Reuther, S. Schönberg, and J. Wolter. DROPS: OS support for distributed multimedia applications. In *Proceedings of the Eighth ACM SIGOPS European Workshop*, Sintra, Portugal, Sept. 1998.

[9] J. Liedtke. On μ-kernel construction. In *Proceedings of the 15th ACM Symposium on Operating System Principles (SOSP)*, pages 237–250, Copper Mountain Resort, CO, Dec. 1995.

[10] S. Microsystems. *ONC+ Developer's Guide*. Sun Microsystems, 1995. Latest version available from: http://docs.sun.com/app/docs/doc/802-1997.

[11] Norman Feske and Christian Helmuth. A Nitpicker's guide to a minimal-complexity secure GUI. In *Proceedings of the 21st Annual Computer Security Applications Conference (ACSAC)*, 2005.

[12] M. Shepperd and D. Ince. *Derivation and validation of software metrics*. Oxford University Press, Inc., New York, NY, USA, 1993.

[13] K. Thompson. Reflections on trusting trust. *Commun. ACM*, 27(8):761–763, 1984.