Großer Beleg
Development of an IDL Compiler

Ronald Aigner
Dresden University of Technology
<ra3@inf.tu-dresden.de>

January 15, 2001

Contents

1 Motivation

Most operating systems have a monolithic structure. This implies, that all functionality, which the operating system provides to its users, is running in kernel mode. This includes components, such as file systems, device drivers, networking components, sometimes the graphical user interface, memory paging etc.

This approach enables the components of the operating system to provide their functionality to each other with low overhead, because every component has direct access to all data and functionality by simply calling a function. Since kernel components have direct access to all kernel internal data, they are able to accidentally or maliciously manipulate it, thus undermining the stability of the operating system.

In contrast to monolithic operating systems, the μ-kernel approach involves minimizing the kernel and implementing servers in user mode. Ideally, the kernel implements only address spaces, inter-process communication (IPC), and basic scheduling. All servers run in user mode and are treated like any other application by the kernel. Each server can have its own address space. They are protected from one another [16].

DROPS stands for Dresden Real-time OPerating System. We develop an operating system with real-time capabilities on top of the L4 μ-kernel. As described above, the whole functionality has to be implemented by separate components, which, nevertheless, have to communicate with each other. Such systems are also called multi-server operating systems.

Multi-server operating systems as well as distributed systems require the components to implement communication code to call each others services. Because this communication code often has the same structure, development tools can be used to ease the implementation. The communication interfaces of these components can be described by using an interface definition language (IDL). Each interface is a semantically related group of member functions. The interface as a whole represents a feature, and the member functions in this interface represent the various exported operations that make up the feature. The concept of an interface fully supports the fundamental notions of object-oriented architecture: encapsulation, polymorphism, inheritance and reusability. The definition of such an interface can be translated into the communication code using an IDL compiler.

My task was to adapt an IDL compiler to the L4 API (see Section 2). The compiler has to produce code in a programming language (C) from an interface definition which generates a message (marshaling code) and evaluates a message (unmarshaling code). The compiler should support the usage of standard data types (`int`, `float`, ...) as well as complex data types (`struct`, `union`, `enum`). Furthermore it should support interface inheritance, the inclusion of target language definitions and different communication relations (client/server as well as simple message passing). The compiler should

generate methods, which can be used to deploy the interface's functions, and templates, which can be filled with the functionality of the server. During the work on the new compiler the preceding work with the Flick compiler should be taken into consideration.

1.1 Component Scenarios

Components have to be able to use the functionality of other components. For the access to another component's functionality four different scenarios can be identified:

1. two communicating components *reside in the same address space and are executed in the same thread*

2. these two components still *reside in the same address space, but are executed in different threads*

3. two components *reside in different address spaces on the same node*

4. the two components *reside on different nodes*

The two communicating components are integrated into a **client-server relationship**. The **client** component requests the **server** component to execute some function for it (serving it). The server receives a request from the client and eventually answers to this request after executing the corresponding function.

If client and server code reside in the same address space and are implemented in the same thread, a simple function call can be used to invoke the server's functionality. In this scenario the two components have access to the same physical resources. Because the function call is executed *synchronously*, no synchronization has to be implemented while using common resources.

If the two components are executed in different threads, the client has to use a communication mechanism to inform the server which functionality should be executed. This execution can be either *synchronously* or *asynchronously*. The asynchronous implementation requires a synchronization mechanism for the access to shared data. Shared data can exist within the address space of the components and both components have free access to it. But thread local data must be made available to the other component.

In the third scenario, which is similar to the second scenario, the client has to signal the server, which functionality it should execute. Relevant parameters have to be made available to the corresponding component by a mechanism, such as explicit memory sharing or copying. Because this scenario allows synchronous as well as asynchronous execution, the access to shared data during the latter has to be synchronized. The fourth scenario

which is again similar to the third, except that the communication mechanisms are different and the sharing has to be implemented in some other way, such as distributed shared memory.

To share data, the third and fourth scenario have to use mechanisms from the underlying layer, such as the operating system. The basis for data sharing is virtual memory, which depends on the hardware and its limitations. The granularity of data sharing is the page size of the hardware. To share data, using memory shares, special communication pages could be used or the memory page, the data resides in, is shared. Often the latter kind of memory share is not fine-grained enough to allow access to the shared object only. If this approach is too risky for the user the data has to be copied.

For the copy scenarios data has to be packed (*marshaled*) to be send from the client to the server, or vice versa. After the other component received the message, the data has to be unpacked (*unmarshaled*). At the client side the steps of marshaling data, which is send to the server, the message passing, receiving the answer from the server, and unmarshaling the returned data are combined into *client stubs*. At the server's side the procedure of receiving a message; unmarshaling the request; calling the appropriate function; marshaling the return data; and sending the return message back to the client are combined into a *server loop*. The functions, which are called at the server's side, implement the functionality of the server. The IDL compiler produce function skeletons (*server skeletons*), which can be filled by the developer.

In the DCE environment the *client stub* is called a proxy. The CORBA environment calls the *client stub* proxy or stub. The name *server loop* does not explicitly appear in either context. The *server loop* is rather separated into the server side stubs, which are responsible for marshaling and unmarshaling the data, and the part of deciding which function to call, which is unmentioned in both environments. While those server side stubs are called stubs in the DCE environment, they are called skeleton in the CORBA environment. I refer to template code for the server's side functions as *server skeleton*.

1.2 Communication Techniques

When looking at these communication scenarios, different techniques might be used to allow a communication between two components. In the first scenario, the data usually doesn't need to be copied. Often it is sufficient to hand a **reference** of the data to the server. If it is necessary that more than one component access the data simultaneously, synchronization is needed. A alternative method to ensure data consistency is to copy the data, e.g. by using a thread local copy.

For the other scenarios, more sophisticated and circumstantial tech-

niques are required. The data can be **mapped in memory** into the server's address space. Memory shares are used to avoid copy operations for large amounts of data. This entails the mentioned possible security breaches. The client needs to trust the server. Because both components operate on the same copy of the data, the access to the data has to be synchronized if possible, though multiple components have simultaneous access.

If the client or server cannot perform any of the above techniques, they have to **copy the data**. During marshaling the data is copied into a message buffer, which is transferred to the server. At the server's side the data is copied from the message buffer into server local variables during unmarshaling. These copy operations can be numerous and time consuming.

1.3 IDL Compilers

The developer implementing the marshaling and unmarshaling code has to take into account all of the above scenarios, decide which one he wants to use and implement one of the communication techniques, described above. It requires a lot of time and the written code is error-prone. To ease these steps (writing client stubs and server loop) the developer can use an automated code generator – a compiler. The interface of the server – the functionality the server exports to clients – is described using an *interface definition language* (IDL). The IDL compiler translates the definition of the interface into the appropriate client stubs and the server loop.

An IDL compiler is supposed to *hide remote invocation*, which means, that the compiler might produce stubs for all four described communication models, depending on the required configuration. The produced stubs are also *less error-prone* and *easier to adapt* to new environments than numerous hand-written stubs. An IDL compiler eases the development process of component communication and should be made available to the DROPS community.

2 The L4 API

My task is to implement an IDL compiler available for component based systems, which are based on the L4 μ-kernel. I will describe the L4 API in more detail to show the specialties, which have to be taken into consideration when building an IDL compiler for the L4 API. The L4 API described in [15] implements three concept, which are necessary to implement a minimal kernel: address spaces; threads and IPC; and unique identifiers.

To enable the communication between two threads, the kernel must provide some mechanism to identify either the communication partner or the communication channel. This is done by using **unique identifiers**.

2.1 Address Spaces

Address spaces operate on page-based virtual memory. The virtual memory concept is used to protect different processes, where each process has it's own address space, by denying a process the access to another process' memory. To allow an abstract memory management outside the kernel, it has to provide kernel-messages to construct and manage address spaces. The L4 μ-kernel supplies three operations *grant*, *map* and *flush* [14]. The grant operation basically changes the ownership of a memory page. The map operation establishes a memory share between two address spaces. The flush operation revokes established memory mappings.

Using these three operations an hierarchical address management can be implemented, which allows a user implementation to enforce it's own memory management.

2.2 Threads and IPC

A thread is an activity being executed inside an address space. The thread executes code, which is loaded into the address space. Because the thread can only access its own address space, the kernel has to provide some mechanisms to allow the thread to communicate with other threads (perform Inter-Process Communication – IPC).

The L4 API provides seven system calls. One of these systems calls is the IPC call, which is used to communicate with other threads. Two communication primitives are sending to another thread and receiving. To optimize performance some variations of these primitives are used, such as concurrent send and receive (reply and wait), etc.

An L4-IPC can contain three different data types. One of these data types is essential to allow the described memory management. It is the description of memory pages (`fpage`). A message may also contain several 32 bit values (`dword`). The third data type to be transmitted is indirect string. An indirect string (`refstr`) describes a memory region by specifying it's start address and the size of this region in bytes. The kernel copies the content of the region into a specified target memory location, which must be big enough to hold this region.

To optimize IPC, L4 implements a special case IPC. If an IPC should transmit only two `dword`s and no `fpage`s nor `refstr`s, these two `dword`s can be transmitted using registers. Therefore this special case IPC is also called Register-IPC or Short-IPC. Because no memory copy operation is involved during the IPC, it has major performance advantages to the normal or Long-IPC.

Eventually the L4 μ-kernel will be improved or further developed. Every change involves an adaptation of existing applications to the new API. One goal of the IDL compiler should be, to hide these changes from the user.

Th L4 API specifies indirectly requirements the compiler has to fulfill. These requirements are the support of different data types, which can be transmitted, and the special case implementation Short-IPC.

3 Related Work

To develop an IDL compiler available to the Dresden development team, I needed to know which IDL compilers exist, what they can achieve and what their limitation are. The following chapter describes the research.

Because IDL compilers need a language which can be used to describe an interface, I compared different interface description languages. The focus was on the most popular ones, the `OMG IDL` and the `DCE IDL`, which are described below. Because the OMG IDL is used to describe components of the Common Object Request Broker Architecture (CORBA), I use the name CORBA IDL to describe the OMG IDL.

3.1 CORBA IDL

CORBA is an object-oriented middleware, developed by the Object Management Group (OMG). In contrast to DCE, CORBA has been object-oriented from the beginning, and it has always been platform and language independent. The CORBA IDL has been selected by the International Standards Organization (ISO) as a "universal" language for describing interfaces to software components [1].

An interface definition file might contain several components. The *type declarations* are used to define additional simple or complex types. *Constant declarations* are used to declare alias names for constant integer expressions. Other components can be exception declarations, module declarations and interface declarations. An *exception declaration* defines an exception which can be thrown by any function of the following interfaces or modules. A *module* is a collection of interfaces, types, constant expressions, exceptions and other modules. By using modules an hierarchical name-space can be established [11].

The *interface declaration* may contain type declarations, constant declarations, exception declarations (which are all valid only in the scope of the interface), attribute declarations and operation declarations. The attribute declarations specify data values, which are members of the interface. Each attribute specification generates four separate codes: two for reading and two for writing the value of the attribute. An attribute in the CORBA IDL is different from an attribute in the DCE IDL. The CORBA IDL specifies an attribute to be a data member of an interface, whereas the DCE IDL specifies an attribute to be an annotation to a member of the specification.

An operation declaration consist of the optional operation attribute `ONEWAY`, the return type, the name of the operation, the parameter list,

exceptions, which can be raised by the operation, and a context expression. The CORBA IDL has an integrated support for exceptions, which can be easily mapped into target languages, which support exceptions [17]. The parameter list consist of comma separated parameters, where each parameter has an attribute (`in`, `inout`, `out`), a type and a name. Because these three attributes are not sufficient for our requirements, we would have to add attributes to the IDL or extend the compiler to use additional annotation files.

CORBA IDL supports the `ANY` type parameter. This type permits the value of an arbitrary type to be transmitted between client and server. The value carries a code which identifies its type [2]. The drawback of such an `ANY` type is that the marshaling and unmarshaling code must be supplied by the user.

3.2 DCE IDL

The DCE IDL is based on the C programming language but is extended by attributes. The original DCE IDL did not implement some of the features, which are supported today, such as interface inheritance or exceptions.

An IDL file contains type definitions, constant declarations, imports and interface declarations. *Type definitions* allow the developer of an interface to specify additional user defined types. *Constant declarations* can be used to define readable aliases for recurring constant integer expressions. The *import* keyword specifies the names of one or more IDL or C header files to include. The import directive is similar to the C `include` directive, except that only data types are assimilated into the importing IDL file.

An *interface declaration* consists of an interface header and an interface body. The interface header contains attributes of the interface and the name. It also contains the optional names of the base interfaces, the interface is derived from. The interface body, which is enclosed in braces (`'{'`, `'}'`), contains the data types that will be used in remote procedure calls and prototypes for the functions that will be executed remotely. An interface body can contain imports, constant declarations, type declarations, and function declarations.

A *function declaration* consist of the function attributes, the return type, the name, a parameter list and an exception list. A function can have several attributes (`IDEMPOTENT`, `REFLECT_DELETIONS`, `ONE_WAY`, etc.), which specify the behavior of the function. The compiler can use all these attributes to generate highly optimized target code for this function. The exception list includes the exceptions, which this function might throw. The compiler transparently transmits exceptions if they are thrown. A parameter has a attribute list, which is enclosed by `'['` and `']'`, a type and a parameter name. Parameter attributes specify the direction of transmission for the parameter and field attributes, which are used if the parameter is a field, to

allow the compiler to optimize the transmission of the field.

The Component Object Model (COM) is based on the DCE IDL. From a commercial perspective COM has been very successful, having spawned the industry's largest commercial market in reusable binary components [17].

3.3 Commercial IDL Compilers

The first IDL compilers have been developed to generate stubs for network communication. These compilers supported the development of distributed application, which operated on different nodes. Depending on the network's characteristics, the stubs did not have to perform well. The networks were the performance bottlenecks of the communication and thus the compilers had no need to optimize the generated stubs. Communication over networks transfers the data in network packages, which are arrays of bytes. Therefore no differentiation of data types, which should be transmitted had to be made.

Most of the work to be done, was to hide the remote invocation of the server, which mostly meant, that platform dependent data formats were automatically converted[1].

As Eide, Ford et.al. described in "Flick: A Flexible Optimizing IDL Compiler"[5]:

> Performance of IDL-generated code, however, has traditionally not been a priority. Until recently [1997] poor or mediocre performance of IDL-generated code was acceptable in most applications: because inter-process communication was generally both expensive and rare, it was not useful for an IDL compiler to produce fast code.

As communication became faster – either networks themselves became faster or the components were moved onto the same node – the performance bottleneck of the communication shifted from the pure communication mechanisms towards the stubs [3, 10].

3.4 Mach Interface Generator

One of the first μ-kernels was Mach. The corresponding IDL compiler is the Mach Interface Generator. In the following I will describe the Mach kernel, especially its communication mechanisms, and the corresponding IDL compiler.

[1] e.g. little-endian to big-endian and vice versa

3.4.1 The Mach μ-Kernel

MACH is a communication-oriented operating system kernel, which supports the following basic abstractions: a *task*, which is an execution environment and has a paged virtual address space and protected access to system resources; a *thread* is the basic unit of execution, which consist of the hardware state necessary for independent execution; a *port* is a communication channel, which is implemented as a message queue, which is managed and protected by the kernel; a *port set* is a group of ports, which can be used to send messages to any of several ports; a *message* is a typed collection of data objects used in communication between threads; a *memory object*, which is mapped into a task's virtual address space.

Message-passing is the primary means of communication, both between two user level tasks, and between tasks and the operating system kernel itself. The only functions implemented by system traps are those directly concerned with message communication; all the others are implemented by messages to a task's `task_port`.

The MACH kernel functions can be divided into the following categories:

- basic message primitives and support facilities,
- port and port set management facilities,
- task and thread creation and management facilities,
- virtual memory management functions,
- operations on memory objects.

MACH and other server interfaces are defined in a high-level remote procedure call language called MIG; from that definition, interfaces for C are generated.

3.4.2 The Interface Generator

The Mach Interface Generator (MIG) generates remote procedure call (RPC) code for a client-server communication. MACH servers execute as separate tasks and communicate with their clients by sending MACH inter-process communication (IPC) messages. The Mach IPC interface is language-independent and fairly complex. MIG is designed to automatically generate procedures in C to marshal or unmarshal the IPC messages that are used to communicate between processes. The user must provide a specification file defining parameters for both the message passing interface and the procedure call interface [4].

MIG generates three files from the interface specification: a file containing the client's side code to send and receive messages; a client header file, which defines the types and routines needed during compilation time; and

another file containing the code for the server to unpack a message, to call the appropriate function and pack a return message.

The specification file may contain the following elements:

- **Subsystem identification** which defines a name prefix for all generated files, and a message identification number, which is the start number to enumerate the methods
- **Type specifications** which I will explain in more detail below
- **Import declarations** which define the files to be included into the client and/or server files
- **Operation descriptions** specify the functions of the interface, which I will describe in more detail below
- **Options declarations** change the values of default *specifications* (see Operation descriptions), such as wait time outs or name suffixes

The **Type specification** defines the types, which are used to declare parameters of operations. Types can be simple, structured, pointer or polymorphic types. Simple type specifications introduce an alias name for a defined number of bits of a specific type. For instance does the following instruction:

```
type my_string = (MSG_TYPE_STRING,8*80);
```

specify a type `my_string`, which is 80 bytes long and of type `char*` or `char[]`.

The structured types are `array`s or `struct`s of `n` elements of simple or structured types. Whenever an `array` or `struct` of variable size is defined, an additional parameter is included into an operation's parameter list, which contains the actually transmitted size. Pointer types define parameters which are sent as out of line data. They should be used for large or variable amounts of data. A polymorphic type allows the user to transmit different types using the same function. An additional parameter defining the transmitted type is added to the operation's parameter list. The polymorphic type is similar to the CORBA type `ANY`.

There are five different kinds of **operation descriptions**, which describe the way a client's call to a server function is implemented. The first description does not expect an answer from the server and returns an error code if the function fails. The second does not expect an answer from the server either, but does not return an error code. If this function fails a formerly specified error function is called. The third and fourth do expect an answer from the server. The third returns an error code, whereas the fourth description specifies an error function. The fifth description waits for

an answer from the server and returns a value, which is not the error code. It too has to specify an error function as well.

The parameters of a function consist of the type, which is one of the previously defined types, a name, and multiple optional *specification*. A specification defines whether the parameter is an `in`, `out` or `inout` parameter. It might also be a wait time out, which defines how long the client waits for an answer from the server. The specification can also contain a parameter which is the request or reply port for the message.

This IDL is tightly coupled to the Mach kernel. It defines kernel specific types, such as ports and brings many attributes into the IDL, which do not belong into a pure interface description, such as wait time-outs. The types, which can be defined with this IDL are very simple and are closely tied to the C target language and the platform (specify the size of a type in bits). Finally it can be said, that the MIG IDL and MIG itself are very strict, very simple and not suited to be adapted to another target platform.

The Mach Interface Generator does produce, nonetheless, efficient code for the Mach μ-kernel. This can be achieved through the deep knowledge of the target API. This example shows, that a compiler, which is designed for one specific target language and communication API, can generate very efficient target code.

4 Flick

As already mentioned in Section 3.3, Eide, Ford et.al., from the University of Utah, realized that the performance of IDL-compiler generated code was very poor. Inter-process communication became fast enough to be no longer the performance bottleneck of a communication relationship between a client and a server.

They stated, that "IDL compilation must evolve from an ad hoc process to a principled process incorporating techniques that are already well-established in the traditional programming language community." Although IDL compilation is a specialized domain, IDL compilers can be greatly improved through the application of concepts and technologies developed for the compilation of common programming languages.

To achieve this, Utah incorporated different practices to generate optimized target code. Some of these strategies are **code inlining**, **discriminator hashing** and **careful memory management**. I'll describe these features in more detail in Section 4.1. They incorporated their strategies into "The Flexible IDL Compiler Kit (Flick)" [5].

The Flick compiler has a modular design, as illustrated in Figure 1. It consist of three modules. The first module is the *front-end*, responsible for reading the IDL input and converting it into an abstract representation, called *Abstract Object Interface* (AOI). The AOI is considered to be the

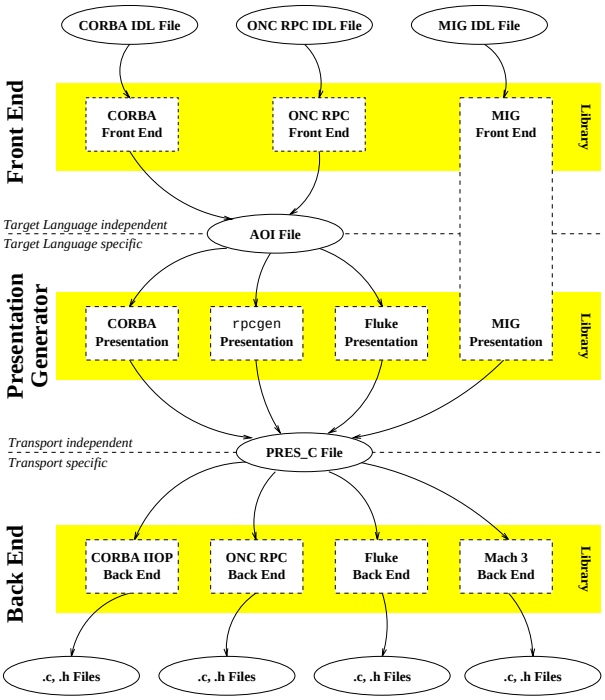

Figure 1: The Flick Compiler structure

high-level "network-contract" between client and server.

Flick's second module, the *presentation-generator* (PG), transforms the the AOI representation of the interface into a "programmer's contract", which includes the declaration of how data is passed between client and server. For different target languages different presentations are generated, because each language has different ways to define data.

The third module is the compiler's *back-end*. This module reads the presentation, generated by the PG, and translates it into the target languages representation of the client stub and server side code.

4.1 New Strategies in Flick

Traditional IDL compilers used nested function calls to marshal and unmarshal data. For example: to marshal a constructed type the marshaling function calls a function for a constructed type and this marshaling/ unmarshaling function calls functions for each of its members. This methodology implies numerous function calls which decrease the performance of the marshaling stub tremendously. Flick avoids these function calls by creating inline function (or macros). Eide et.al. qualified the effect in [5] to be (compared to PowerRPC and `rpcgen`) about half the object code size, meaning that Flick stubs are half the object code size of `rpcgen` stubs. Other compilers generated even larger object codes.

Another disadvantage of traditional IDL compilers is dynamic resizing of the transmission buffer. Whenever a new piece of data is marshaled, the stub checks, if the buffer is big enough to hold this new value. If the buffer is too small, it is resized. Flick optimize this strategy by checking the total size of the transmission buffer once. This way permanent checking and reallocation can be avoided.

When using `in` parameters (parameters send to the server) at the server's side, Flick uses the following techniques to avoid additional memory allocation. It uses these parameters directly from the transmission buffer. This technique can be applied, whenever the parameters have the same memory layout in the client's and server's address space. This also implies, that the buffer cannot be used to receive further requests, if the server still uses references to it in one of his service functions.

Flick optimizes the marshaling of fixed sized portions of the message (*chunks*) by addressing the buffer using the start-address of a chunk and a constant offset. Thus the target language compiler has the chance to further optimize these instructions.

4.2 L4 Adaptation

The L4 adaptation of Flick has been done by Volkmar Uhlig during his master thesis [18]. My work is mainly based on his and draws a lot of

experience from it. To adapt Flick to the L4 API, the back-end of the compiler had to be changed, specifically the methods and classes, which write the target code.

Flick's class structure is very flat and made it harder to adapt it to the L4 API than expected. The adaptations were restricted to the functions, which print the target code. These files can be found in the Flick directory at

`c/pbe/l4/client.cc and c/pbe/l4/server.cc`

These files create stubs, which mainly contain macros, which do the message buffer handling, sending and receiving messages, and error handling.

All of these macros had to be changed to work with the L4 message structure. They had to handle the L4 message specific buffer and the special types (see below) defined for L4. Because these macros are used everywhere throughout the stubs, this included macros for declaring buffers or variables as well as macros for initializing buffers and macros which marshal scalar values (or other data) into the message buffer.

The back-end files also contain functions, which can be used, if the stubs should be manipulated manually. E.g., the server loop always has to be written by hand, and these functions are convenient for this task. They manipulate the message buffer, thus hiding the message internals from the developer.

Major problems occurred when we tried to incorporate the different data types, which the L4 message structure supports. Because restriction apply to the order in which parameters have to marshaled, we had to change the internal behavior of the compiler. The compiler usually writes data right into the message buffer. We had to implement multiple runs, checking each time which data type the parameter is and whether it can be marshaled in the current run or not. I will avoid these multiple runs in a new IDL compiler by sorting the parameters prior to marshaling.

Because no abstraction about `fpage`s and no abstractions for indirect strings (see Section 2) existed in the CORBA IDL, we had to include these types directly into the IDL. This contradicts the idea of using the IDL as target independent interface description. The data type of indirect strings could have been implemented using some kind of annotation, which tells the compiler, that a string should be treated as a `refstring`, but the CORBA IDL has no possibility to annotate an interface definition.

Other adaptations were the manipulation of the error-handling code to work with the new back-end and the L4 specific types, such as `l4_fpage_t` or `l4_threadid_t`.

One drawback of this adaptation is, that whenever the L4 API might change, all the code changes have to be repeated. Even though we collected experiences on how to adapt Flick to a L4 API, we think that the changes would comprise almost everything we already changed.

To really exploit all features of the compiler, we would have to change the structure of the compiler. The optimization only takes an untyped message buffer into account. As soon as different data types can be transmitted, further optimization cannot be applied.

Furthermore, Flick's structure did not allow to integrate a optimization for short and long IPC during compile time. A Flick generated stub decides whether to use a Short- or Long-IPC during run-time. It marshals all data into the message buffer (which has to be allocated) and after marshaling checks whether it has only two `dword`s in it's buffer. In that case it uses a Short-IPC.

4.2.1 The IDL

One of the applications of the IDL compiler and thus the IDL, should be the usage of existing data structures, written using the C programming language. These data structures exist in code, which we like to reuse for the development of components. Therefore the support of existing C data-structures by the IDL had to be checked. Because the CORBA IDL is platform independent, a C data structure cannot be copied to the IDL, but has to be redefined in the IDL. Different approaches are possible.

When porting existing C code into an skeleton or framework, which has been generated by an IDL compiler, the existing data structures have to be adjusted. When using the CORBA IDL to describe an interface and existing C code will be wrapped by the interface's framework, it is very hard to describe the existing data structures using the CORBA IDL. As Eide describes in [7] it is necessary *to reverse-engineer the CORBA IDL description of the protocols*, which implies a lot of work.

The CORBA IDL supports existing data structures, no matter which language they are written in, as *native types*. A native type can be used in the interface specification, but the marshaling and unmarshaling routine for this type have to be specified manually by the user. These user defined types cannot be used for optimization algorithms, because their layout is unknown to the compiler. Does the communication API change, the marshaling and unmarshaling routines have to be adapted as well.

Another approach could be the usage of the structures, which are produced by the compiler from a CORBA IDL specification. To further use the existing data structure for internal handling, the internal structure must be copied to and from the compiler generated message structure. Doing this, another unwanted copy operation is inserted into the communication path. To avoid the copy operation the compiler generated structure should be used instead of the existing. This implies a major code change which is time-consuming and might introduce new errors into the existing software.

An alternative could be the incorporation of the native C structures into the IDL. But this results in a rewrite of the compiler, because internals

data flow and structures of the compiler might not work properly without adaptation.

4.2.2 Lessons learned from Flick

Opposed to the proposal of having a modular design, which allows the easy adaptation, the Flick compiler has to be severely changed if a new IDL or target language has to be incorporated. Not only the front-end, which is responsible for generating an abstract description of the IDL, has to be re-implemented, but also a new presentation generator is needed, when changing the IDL. The same problem arouses, if a new target language or communication API should be supported. In that case not only the back-end has to be replaced, but - again - the presentation generator.

Flick does not support typed messages. It assumes, that all data, which will be transmitted, has to be copied into one linear buffer. Flick only differentiates the sizes of the data. Because L4's communication primitives support multiple data types, an incorporation of this support into Flick requires major changes. To allow Flick to optimize different data types, most of the infra-structure has to be changed, which is similar to a complete rewrite. The current L4 adaptation of Flick has rather been a quick implementation to make these different data types possible, but permits no optimization.

5 Design

As described in earlier sections, we ran into a lot of difficulties while trying to adapt the Flick IDL compiler to the L4 API. The most problems occurred when incorporating the different message types supported by L4. These entailed the lack to optimize these types and the violation of abstract nature of the interface definition language.

The code generated by the Flick IDL compiler still inserts overhead into the marshaling and unmarshaling code. We think we can eliminate the overhead using a better compiler design. Especially run-time optimization of a short IPC reduces performance. Detailed analysis of the Flick stubs is described in Section 7 as a cost analysis.

In a previous work ([18]) the Flick compiler has been adapted to be used with the DROPS environment, described in Section 1. One of the goals was to build wrappers for component servers around existing code, which we took from a Linux distribution, e.g. device drivers, IP stack and similar modules. Thus a lot of our code has to be integrated into server skeletons generated by Flick.

5.1 Interface Description Language

To find an appropriate IDL for the problems described in Section 4.2.1, I defined four goals, which should be met by the IDL:

1. **easy to learn**,
2. **no rewrite of existing code** should be necessary and
3. it should be possible to **annotate the IDL** to give the compiler more information to optimize the stubs.

Because an IDL is always a very small language – it only declares an interface and contains no instruction code – it is also fairly easy to learn. I consider DCE's IDL easier to learn if you are familiar with C or C++, because the syntax is almost the same. The CORBA IDL is easy to understand and, if you are not familiar with C or C++, it should be equally simple to learn both languages.

Integration of existing code requires a close syntax of the IDL to the target language. Because the syntax of the DCE IDL is closely related to the C syntax, it is possible to include C header files, containing type definitions, right into the IDL. The compiler understands and uses these C types when generating the marshaling stubs. It also can fully incorporate these types into its optimization algorithms. The compiler can generate wrappers for existing code, which use the code's data structures, thus avoiding unnecessary copy operations.

Stated by Ford in [9, 8] annotated interface definitions can result in faster stubs. This improvement is achieved by specifying information in the annotation, which regards the communication. This additional information is essential when generating stubs to apply the right optimization strategy. The annotation is usually placed into an extra file, which clearly separates the abstract description from the implementation specific optimization information.

Some of these optimization techniques can already be incorporated into the IDL, if they are abstract enough. For example, it is important for the optimizer to know which size a variable-sized array actually has. This can be achieved by using an attribute, which describes the exact size of the array. The DCE IDL has such an attribute, called `size_is`. The compiler might then decide, depending on the value of this attribute, which marshaling technique is the best. If the value is only known during run-time, it might still enable the stub to decide which mechanism to use to achieve better performance results.

But sometimes the general attributes, specified in the IDL, are not specific enough to allow the compiler to further optimize the stubs. For these cases it is necessary to specify attributes, which are specific for a certain target platform. These annotations regard the transport of the data, not

the interface. To be able to specify annotations outside the interface definition file, an additional file can be used. The DCE IDL calls these files Application Configuration Files (ACF).

The ability to annotate an IDL and to add attributes in a sufficient manner are clear advantages of the DCE IDL over the CORBA IDL. The disadvantage of the DCE IDL is it's close relation to C, which is adequate if the target language is also C, but becomes inadequate if the target language is different from C.

I decided to use the DCE IDL, because the CORBA IDL did not satisfy the given IDL goals and an extension of the CORBA IDL or the development of a new IDL would have been too time-consuming.

5.2 Interface Inheritance

The concept of inheritance is known from object oriented programming using classes. A class can be derived from a base class, inheriting the base's classes methods. The derived class may add functionality or change the functionality of the base class by overloading it's functions. This way the behavior of the functions is changed.

The difference between class and interface inheritance is the not existing implementation of the interface's functions. At least not within the scope of the interface's definition. Thus overloading of functions is impossible. The only reason to overload an interface is to add functionality – functions – to the interface's definition. This can also be done using multiple inheritance to collect the functionality of multiple interfaces into one interface.

As Hamilton showed in [13], interface inheritance is a good instrument to address the software evolution of, for instance, device drivers. Thus, it is necessary to integrate this feature into the IDL.

The integration of interface inheritance into the IDL compiler let some problems occur. For a single interface, four files are generated.

- The client header, which contains the client function's declarations
- The client implementation file, which contains the client stubs
- The server header file, which declares the function skeletons and the server loop
- The server implementation file, which contains the server skeleton functions and the server loop.

When adding a derived interface, the compiler has to generate these files for each interface. Does the compiler generate the code for all interface in the mentioned four files, the developer has to separate these functions by hand to use a subset of them. But this solution produces an own server loop for each interface. The developer must combine these server loops by hand

to obtain one server loop, which can server all interfaces in the inheritance chain. One server loop should be created. Thus, I removed the server loop from each interface's files and created one instance of a server loop for one compilation run and inheritance chain. (We have four to five files now. Actually it's even one more, which is described in Section 6.2)

5.3 Compiler

Compilers are generally constructed with two parts, the front and back end. The front-end reads the file, which is compiled, and the back-end writes the target code. Common language compilers include additional, optional modules to optimize the generated code and generate intermediate data (such as assembler code, etc.).

When a compiler translates an input file into a target file, it passes several compiler stages. These stages are similar to the modular design and represent the steps taken to generate the target code. The first stage reads the input files, the second stage usually contains the analysis of the parsed code, which checks the data flow and tries to optimize it. Flick has been the first free IDL compiler which paid some attention to this stage. Even more emphasis has to be put on this stage to accomplish the generation of fast target code. The third stage writes the target code.

From the lessons learned from the adaptation of Flick, I defined three compiler goals.

1. it should generate **fast target code** – this is the main goal of the compiler, and the reasons for implementing a new compiler at all
2. it should be **easy to maintain** – meaning that bugs can easily be found and fixed, or changes can be incorporated fast
3. it should be **easy to adapt** – this describes the idea to change major parts of the compiler easily and quickly. For instance, to implement a new communication API.

I designed an IDL compiler, named IDL[42], with a similar structure to the Flick architecture. The compiler is separated into three modules, as shown in Figure 2 and described above. This modular design should enforce the easy adaptation to new IDLs or target languages.

During the first stage the compiler also generates a data representation. This data representation contains all data, which is transferred using a message. Meta-information, such as attributes are kept in the front-end classes, to clearly separate the actual data from the annotated data. The data-representation is analyzed and manipulated by the second stage – the Optimizer.

[2]The IDL compiler developed at the University of Karlsruhe is also called IDL[4]. Please see this name as project name, which still might change.

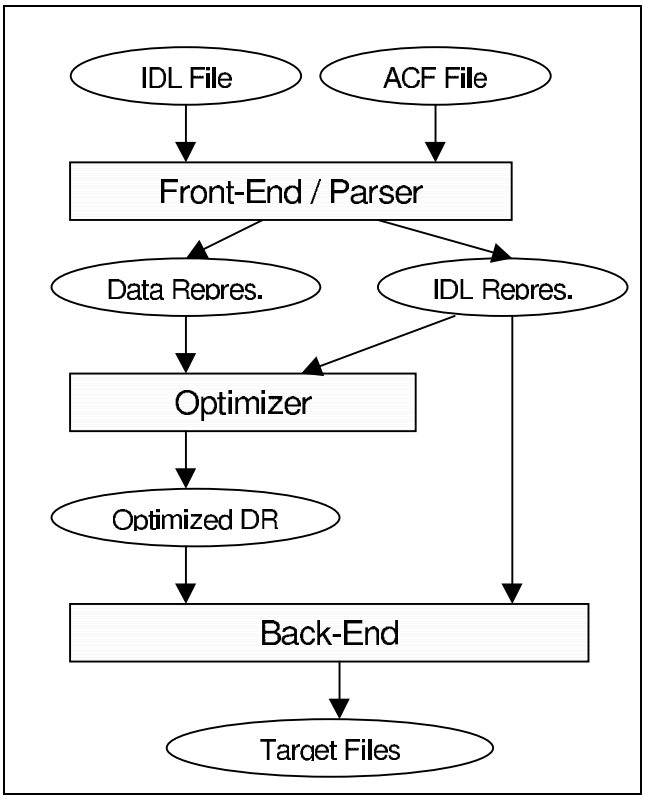

Figure 2: The IDL4 Compiler structure

Goal of the optimization stage is to manipulate the data- representation in a way, which allows the back-end to write fast target code. Steps which might be taken during optimization are described in Section 6.3.

The third compiler stage is the back-end. It is responsible to produce the target code. Opposite to the Flick approach – to write macros into the target files and define the macros in separate header files – IDL^4 writes the code directly into the target files. This approach allows the compiler to generate better code, because the structure of a client stub, for instance, is different if different kernel calls are used. E.g. no message buffer has to be created, initialized or filled if a short-IPC is used.

Eide describes in [6] how stubs have to be divided to be suitable for different models of distributed communication. The reasoning in the paper ([6]) is to allow asynchronous communication by queuing messages and to receive multiple replies to one request. He separated the stubs into "decomposed stubs". These decomposed stubs consist of:

- *pickling stubs*, to marshal requests and reply messages;
- *unpickling stubs*, to unmarshal requests and reply messages;
- *send stubs*, to transmit pickled messages to other nodes;
- *server work functions*, to handle received requests;
- *client work functions*, to handle received replies; and
- *continuation stubs*, to postpone the processing of messages.

I incorporated these ideas into my IDL compiler using the described modular design. The back-end should support the writing of these "decomposed stubs". The **Implementation** Section contains a more detailed description of the ways to incorporate these designs into my compiler.

One application of the decomposed stubs could be to selectively write the parts of these stubs. The compiler could, for instance, write only the marshaling and unmarshaling parts (*pickling and unpickling stubs*). The developer could then use these parts to implement simple message passing.

5.4 Optimization Strategies

During the work with the Flick IDL compiler I researched some optimization strategies Flick incorporates and found some disadvantages in the code generated by Flick, which are due to Flick's design, and which we like to avoid.

The most important enhancements used to generate fast target code are described in Section 4.1. These are the strict inlining of code to avoid the overhead of function calls; allocating the message buffer once and avoiding

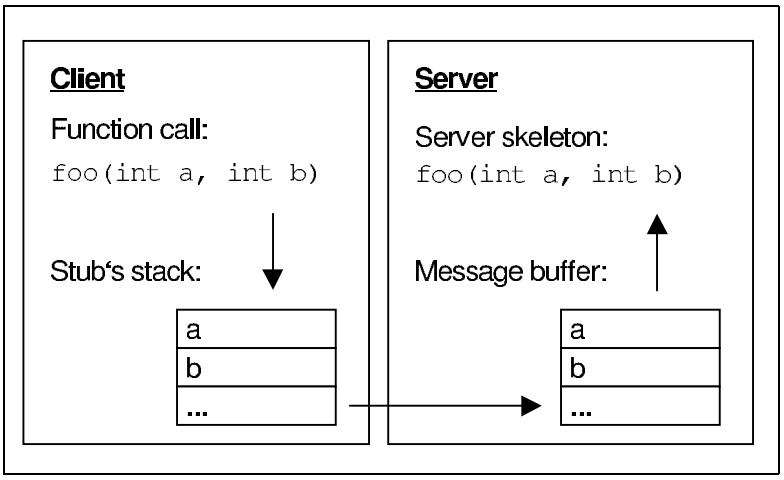

Figure 3: Minimize copy operations using direct stack transfer

frequent size checks and resizing; avoiding copy operations by using the message buffer directly wherever possible; aligning data during marshaling.

To incorporate the first technique, the compiler writes stubs as pure C code, without any function calls, macros, etc. To enforce the idea of static sized buffers the compiler analyzes the data, which will be transmitted. It is possible to use static sized buffers and constant offsets in the buffer more intensively, thus allowing the target language compiler to optimize the code better than using variable sized buffers and offsets.

The third optimization technique has been highly optimized by the System Architecture group in Karlsruhe, as described in [12]. To avoid additional copy instructions, the specialized compiler generates a client stub, which transfers the client stub's stack directly to the server (with slight modification, e.g. insert the server's address). At the server's side the appropriate server skeleton function is called using the message buffer as stack[3]. The steps are illustrated in Figure 3. This way copy operations into and from the message buffer can be avoided. But this optimization can only be applied if the target language compiler uses this scheme to call functions (push parameters on the stack) and only for a few parameter constellations (no use of pointers or references).

I have to follow one of the major restrictions, which are implied by these optimizations. The IDL compiler generates code for the target language compiler (in the DROPS project the `gcc` compiler). I compiled some C code examples with `gcc` and tested the resulting object code. Sometimes `gcc` can produce shorter object code from C code, which has more instructions than

[3]The message buffer is actually located in the server's stack in such a way, that the function call can easily be performed.

other C code. Some examples can be found in Appendix B.

Another strategy which can be implemented into the compiler is to optimize copy operations for complex data types. Instead of copying every single element into and out of the message buffer, the whole data structure can be copied as one piece of data. Again, this allows the following compiler to optimize this operation more efficiently. This optimization technique can only be used if the memory layout of the complex structure is the same in the client's *and* the server's address space. If the complex structure contains pointers to other data, these pointers have to be adapted to the server's memory layout.

To use the provided space in a message buffer more efficiently, multiple values might be compressed into one value in the message buffer. A possible scenario for packing multiple values into one `dword` is the usage of sub-byte types. A sub-byte type is a data types, which uses less than 8 bits to contain its values (e.g. the boolean type, which uses only one bit).

Again, this optimization strategy is based on the knowledge of the target language, the following compiler and the target communication API.

To conclude, I would like to define the following goals for optimization algorithms and strategies for IDL compilers:

- minimize copy operations
- a consolidated knowledge of the target language's compiler
- a expertise knowledge of the communication API and it's special cases
- try to optimize as much as possible during compile time

Other optimization strategies have to be found, tested and implemented into the compiler to further enhance the code generation process.

6 Implementation

During the design phase I defined three goals the IDL compiler has to meet. The first goal – to produce **fast target code** – is the major goal of this compiler. I will discuss its implementation in more detail later in this section. The second goal – to be **easy to maintain** – is a prerequisite to improve the compiler's functionality and code generation. If a developer is not able to change the compiler, it cannot be maintained. I tried to reach this goal by using an object-oriented design and implementation. The third goal was driven by the need to adapt to a changing communication interface. The L4 specification is evolving and this evolution must be supported by the compiler. To achieve this goal, the compiler has to be **easy to adapt** to new target platforms.

The second goal of the IDL compiler – to be easy to maintain – is achieved by using an object-oriented programming language. `C++` supports the object-oriented design through the concept of classes. I organized the logical parts of the compiler into classes. The methods of these classes represent their functional parts. When deriving from a class a logical unit is "replaced" or modified. When overloading the methods of the base class, the functionality of the logical unit changes. Whenever functionality has to be adapted, a programmer has to search the class (logical part) of the compiler, where the code to be changed is located. He might then either derive his own class from this class and overload the appropriate methods, or he can directly change the methods of the class. This separation should ease the maintenance of the compiler.

To locate a class the developer has to find its file. A class resides in two files: a header file and an implementation file. The header file ends on `.h` and the implementation file ends with `.cpp`. Both files of a class have the same name. A class' name consist of a prefix, which shows which module it belongs to, and a short descriptive name. All classes have the prefix `C` which is followed by the module prefix. The front-end classes have the prefix `FE`, the back-end classes the prefix `BE` and the data-representation classes have the prefix `DR`. Classes with no module prefix do not clearly belong to one of the modules but rather to the whole compiler. The class representing an interface in the front-end would be named: `CFEInterface`. The filenames to not contain the leading `C`, which means that the files for class `CFEInterface` would be `FEInterface.h` and `FEInterface.cpp`.

The decision to use an object-oriented programming language and to strictly follow this programming philosophy enables me and other developers to define clean interfaces between the compiler modules and to be able to change modules without changing the whole compiler. This can be achieved by hiding class internal data from other classes and allowing access to it through defined interfaces. The modular design fulfills the requirements of the third goal by using a clean separation of the modules.

To allow the clean separation of the modules I had to define the scope of each module. The front-end module is responsible for parsing an IDL file, checking it's syntax and grammar and generating an in-memory representation of the IDL file. This in-memory representation should contain all necessary elements to be able to exploit all features of the optimizer. As described in Section 4 this comprises all kinds of attributes or annotations. The front-end also has to generate a data-representation from the IDL. Currently this is done during the creation of the front-end's in-memory representation of the IDL.

The job of the optimizer module is to manipulate the data-representation in such a way, that the back-end can produce fast target code with it. Because additional stages, e.g. modules, might be added in the future and these modules need a original representation of the IDL, the optimizer is

not allowed to manipulate the in-memory representation of the IDL.

The back-end's job is to write the target code to the target files. It often writes multiple presentation of IDL elements to the target files. A method of an interface appears in the client header and implementation file in different ways (in the header file as declaration and in the implementation as client stub). The same method exists also as server side declaration and function skeleton. To be able to produce all these different presentations the back-end produces these presentation from one original – the in-memory presentation of the IDL. So – one might ask – if the back-end uses the in-memory presentation of the IDL, what do we need the data-representation for? The data-representation is used to generate the marshaling and unmarshaling stubs of the data, as described in Section 6.2.

6.1 The Front-End

As outlined above, the front-end is responsible to parse an IDL file and generate an in-memory representation of it. The parser is created using the Gnu tools `flex` and `bison`. I defined the IDL key words in the file `scanner.ll`. The `flex` tool generates from this file a syntactical scanner, which scans the input file for these keywords or *tokens*. The scanner is a function, which is called by the grammar parser, and returns the value of the next token it scanned in the file.

The grammar parser is generated from the file `parser.yy` by the `bison` tool, which is a Gnu implementation of the `yacc` tool. The grammar file contains rules how an IDL file has to look like. The grammar rules describe the look of elements of the IDL in a syntax, which is similar to the extended Backus-Naur Form (EBNF). The grammar parser is also a function, which is called from the compiler's `main` function.

The grammar parser generates the in-memory representation of the IDL using front-end classes. Each front-end class belongs to an element of the IDL file. E.g. does the class `CFEInterface` represent an interface. While creating the in-memory representation of the interface the appropriate data-representation is generated synchronously.

This shows that the whole first stage of the IDL compiler is situated in the grammar parser function. It checks the syntax and grammar of the IDL file and generates the in-memory representation of the IDL. The object-oriented concept exists in the parser by creating objects as in-memory representation of the IDL and by the clear definition of the parser's job.

6.2 The Back-End

The back-end classes, responsible for writing the target files, are marked with the prefix `BE`. They are separated into two groups. One group is responsible

for the logical structure of the target files and the other group is responsible for the elements within the files.

The logical structure basically represents the target files themselves. Depending on the compiler's parameters, different classes are generated and used. The standard configuration of the compiler generates five files. The client header file, which contains the declaration of the client stubs. The client implementation file, which defines the client stubs. The server header file, declaring the server skeleton functions. The server loop file, which contains the server loop function. And a header file containing the function identifiers. For each of these files a corresponding class is created.

Almost every class of the back-end – at least each classe, which is responsible for writing the target code – implements a `Write` method. The `main` function calls the `Write` method of the back-end's root class which in turn calls the `Write` methods of all its child classes, which are the target files. These `Write` methods do all the work. They check which options are set and call the appropriate nested `Write` methods. The back-end's root class checks whether only a client header files is created (including the client stubs as inlined functions) and then calls the `Write` method of either only the client header class or both the client header and implementation classes.

I mentioned that classes are the logical units of the compiler and their methods the functional parts. If the layout of the target files should change (e.g. an additional `#include` directive should be written to all client header files), then the appropriate `Write` method can be modified to accomplish this. To be able to have a more fine-grained access to every single write operation I introduced into every back-end class specific virtual methods. Their behavior can be changed by either changing them directly or by overloading their functionality in a derived class. To add the `#include` directive the method responsible for writing the `#include` directives in the client header class can be modified by adding a statement writing the line into the file.

To adapt the back-end to another target communication API a minimum of back-end classes has to be exchanged. Currently the basic back-end contains 25 classes. I overloaded 8 of them to adapt the back-end to the L4 API. Most of these changes are minor changes which, for instance, add an `#include` directive into a header file or add another prefix to the written names (for the L4 adaptation this prefix contains an additional `l4`). The changes, which really affect the target code are located in three classes. To adapt the L4 back-end to another communication API only the communication relevant methods and classes have to be adapted.

To have a very detailed control over the code generation process every element which appears in the target file is represented by an own back-end class – similar to the front-end representation of the IDL. Thus this class can be specialized to write the best code for the corresponding element. The back-end class knows best how to collect additional data and what to write

to generate fast target code. For instance exists a back-end class to write types and a back-end class to write constant declarations.

When the back-end is generated the classes draw the needed information from the in-memory representation of the IDL and the data-representation.

I mentioned in Section 5.2 that the files, which are generated, are actually one more than mentioned (five to six files). The additional file contains the function identifiers as `#define` directives. These numbers start for the top most base interface with one and increase with each method by one. The top most base interface is the interface in the inheritance chain, which is not derived from another interface. Each interface has it's own function identifier file. These files contain a base number, which defines the starting number for this interface. All function identifiers are numbered relative to this base number. If the function identifiers of an interface have to be moved to another base this can easily be done by changing the base number.

6.3 Optimizer

As described in Section 5.4 multiple strategies have to be followed to generate fast target code. Some of these strategies concern the general design of the compiler, such as the optimizer module and some concern the written code.

The optimizer module currently consists of one class (and the derived classes) which does the optimization steps. This class analyzes the data-representation to check the parameters of a function for their size to be able to determine whether they should be transmitted using a `dword` array or an indirect string.

Most of the work currently done to generate fast code has been put into the back-end. Because the generated code of Flick has been one of the major performance bottlenecks I concentrated on the code generation. The optimizations cannot be performed by only manipulating the data-representation but also has to be taken into consideration when writing optimized code. To enhance the special case implementation of the short IPC, the compiler checks the data-representation whether the parameters fit into the registers and then writes optimized code for a short IPC into the target code.

Other optimization strategies are the handling of variables and the message buffer. For instance can a server side implementation of a function use the message buffer directly, if the buffer contains `in` data which does not overlap with `out` data. The `in` data also has to be located entirely in the message buffer. If complex data structures have to be restored this optimization strategy might be of no concern if the costs to restore the structure overweight the improvements of the copy avoidance. These manipulations only concern the code generation process, not the data-representation.

We recognized major performance drawbacks whenever data had to be copied. So one of the optimization goals is copy minimization. This opti-

mization can be achieved by analyzing the data flow in the client stub and server loop and by avoiding unnecessary copy operations.

The current basic implementation of the optimizer class implements no real optimization strategies. Currently it can only analyze and flatten complex data-structures to ease the target code generation. The L4 adapted optimizer class does also analyze the elements for their sizes to find out if the short IPC optimization can be applied.

Part of the optimization process is the marshaller. The marshaller writes the code which copies the parameters of an function to or from the message buffer. Therefore the marshaller needs a very detailed knowledge about the communication API. It is responsible for the fast marshaling code.

6.4 The Factory Idea

I talked a lot about changing the code of the classes. Whenever code has to be changed to adapt a class to a new target platform, a developer might search for the appropriate method and manipulate it. But these changes are permanent to this class and when using this class for yet another target platform the old method's implementation could have been useful but is lost now. Thus it is more practicable to overload this method in a derived class. But this class might be used in several places, which all must be found and manipulated to use the new – derived – class.

To avoid this additional work every back-end class is created using a central instance – the **Class Factory**. Whenever a specific class has to be generated the appropriate method of the class factory is called and this method returns the wished class. When class changes appear, the class factory can be adapted to generate the new class. Even when the class factory's methods have to remain unchanged, it has to be overloaded as well. The new class factory has to be made public.

Similar problems occur whenever a name has to be written to the target code. For instance: a function's name is written multiple times. Does the naming scheme for the target platform change, all of these places, where the function name is written into the target files, have to be found and changed. This can be avoided by using the **Name Factory**. It is, similar to the class factory, a central place where names are generated. Does the naming scheme change, this name factory class has to be overloaded and the new name factory has to be made public.

6.5 The Context Concept

To make the factories public, a reference to each of these classes might be handed to every method in need of it, or put them into some global place, e.g. define global variables. The latter approach does violate the object-oriented approach. Both approaches reach their limits if more or other classes as well

should be handed to the methods. To avoid this, the concept of a **Context** was introduced. This context is different to the "context" used by Flick, e.g. the CORBA IDL, which uses the name "context" to describe the state a client stub or server is in when sending or receiving messages.

I use the term "context" to describe all information, necessary to write target code. This includes, besides the two factories, the target file and the parameters the compiler has been called with. The back-end can change the context, e.g. the target file – it can be the client header file or the client implementation file, etc. These changes are a reason to hand every method a reference to the context, rather then setting a global variable. Whenever the context changes the old value of the global variable has to be stored. The storage of the old context is done more easily in the method which changes the context and eventually restores it.

7 Performance Evaluation

I measured the following numbers on an intel Pentium (133MHz) computer with 512KB of Cache and 64MB of RAM. The communication stubs were generated for the L4 IPC communication mechanisms. I tested several client and server components, which ran on top of the L4 μ-kernel version 2.0. To simplify the test environment (input/output etc.) most of the components are Linux tasks running on L^4Linux (an adaptation of Linux 2.2 for the L4 μ-kernel). Because I had no other applications running, while measuring the different scenarios, I assume, that the times measured are the same as the times would have been on the pure L4 μ-kernel.

The main objective of these measurements is to compare the performance of the Flick generated stubs with the stubs generated by my IDL compiler. Major performance gains can be seen whenever a short IPC is used to communicate, because the Flick compiler generates stubs, which decide to use a short IPC at run-time, whereas the IDL^4 compiler decides to use a Short-IPC at compile-time.

I measured different scenarios, which are the transmission of a few single `dword`s, the transmission of complex data types, which are copied, and the transmission of data types, which use the communication mechanisms of L4, such as indirect strings.

For each of these scenarios I separated the communication into several components to distinguish the parts with performance enhancements and the parts which add constant overhead to a communication path. These parts are:

- the time from the call of the client stub to the beginning of the marshalling
- the time of the marshalling

- the time needed to send the IPC from the client to the server
- the time at the server from receiving the IPC to the beginning of the unmarshaling, also called the server's switch time
- the time of the unmarshaling to the moment the real function starts to execute
- the time from leaving the server function until the marshaling of the return values
- the time of the marshaling of the return values
- the time needed to send the return IPC
- the time to perform error handling
- the time to unmarshal the return values

or any subsequent combination of them. The most interesting times are the times from the beginning of the client stub to the moment the server function starts to work. This is the overhead an inter-process function call costs. To show whether my IDL compiler creates reasonable client stubs I also hand-coded some of the function's stubs, hand-optimized these stubs and measured their times.

7.1 Short IPC

The L4 short IPC, as described in Section 2, is a special case implementation of inter process communication. It transmits only two `dword`s using registers. No memory copy operations are invoked, which makes this kind of IPC very fast. Whenever possible it should be considered to use this IPC mechanism.

To measure the short IPC communication stubs I used the following IDL to generate the client stubs[4].

```
interface foo {
  int bar([in]int t1);
}
```

The IDL[4] compiler generates the following client stub from this IDL:

```
L4_INLINE int foo_bar(l4_idl_service_t *_service,
    /* in */ int t1,
     l4_idl_connection_t *_connection)
{
    l4_msgdope_t _result;
    int _error;
    dword_t _return_code;
    dword_t _return;
```

[4]The CORBA IDL for Flick looks similar.

```
    _error = l4_i386_ipc_call (_service->server_id,
10      L4_IPC_SHORT_MSG, test_foo_opcode, t1,
        L4_IPC_SHORT_MSG, &_return_code, &_return,
        _connection->timeout, &_result);
    return (int)_return;
}
```

Because this client stub is directly written into the calling code (inlined), an optimizing target language compiler (`gcc` with optimization turned on) writes the variable directly into the IPC call. This is a `mov` operation from a memory address into a register. The costs are memory access costs, meaning if the value is located in the cache, the costs are only cache access costs. Additional to those costs might be the access cost to physical memory and cost for TLB misses.

The Flick compiler generates the following client stub for the IDL. As to be seen, the code is much longer than the code, generated by the IDL^4 compiler. All lines starting with `flick_l4_` are macros, which will be replaced with C code. These macros hide general tasks, which are specific to the communication mechanism, such as marshaling a signed 32 bit value into the buffer. Another difference to the IDL^4 code is the error-handling. The IDL^4 compiler does not yet generate as much error checking code as the Flick compiler. During the measurements I separately measured the error-handling code to determine the differences correctly. The error-handling code is located at the lines 25 to 27 and the macro on line 24 expands to multiple lines of C code, which also contains a lot of error handling code. The pure marshaling of the data is located in line 19, which writes the value of the variable `t1` into the message buffer.

The code also shows the handling of short-IPC by the Flick compiler. It first marshals all parameters (in this case `t1`) into a message buffer (lines 13 through 23). It then hands the message buffer to the message invocation macro (line 24). This macro uses the first two values in the message buffer for the register IPC. Flick introduces unnecessary copy operations into the short-IPC stubs. The Flick generated stubs look like the following:

```
sdword_t foor_bar(sm_service_t service,
    sdword_t t1, sm_exc_t *_ev)
{
  char *_buf_current, * _buf_start;
5 l4_msgdope_t _rc;
  dword_t _nr_received_strings;

   l4_ipc_buffer_t __msgbuf; /* temporary send buffer */
   l4_ipc_buffer_t * _msgbuf = &__msgbuf;
10 sdword_t _return;

  _return = 0;
   flick_l4_client_start_encode ();
  {
15   flick_l4_encode_new_glob(8);
```

```
        flick_l4_encode_new_chunk(8);
        flick_l4_encode_unsigned32 (0, req_test_foo );
        /* Begin encode phase on parameters */
        flick_l4_encode_signed32 (4, t1);
        flick_l4_encode_end_chunk(8);
        flick_l4_encode_end_glob (8);
      }
      flick_l4_client_end_encode ();
      _rc = flick_client_call (service , _msgbuf);
      if (L4_IPC_IS_ERROR(_rc))
        flick_l4_decode_client_error (return _return,
          FLICK_ERROR_COMMUNICATION, l4, l4);
      flick_l4_client_start_decode ();
      {
        flick_l4_decode_new_glob (8);
        flick_l4_decode_new_chunk(4);
        flick_l4_decode_signed32 (0, _ev->_type);
        flick_l4_decode_end_chunk(4);
        switch (_ev->_type)
          {
          case exc_l4_no_exception:
            {
              /* Begin decode phase on parameters */
              flick_l4_decode_new_chunk(4);
              flick_l4_decode_signed32 (0, _return);
              flick_l4_decode_end_chunk(4);
              flick_l4_decode_end_glob (8);
              break;
            }
          case exc_l4_system_exception:
            {
              flick_l4_decode_end_glob (8);
              flick_l4_decode_system_exception (_ev , l4 , l4);
              break;
            }
          default:
            {
              flick_l4_decode_client_error (return _return,
                FLICK_ERROR_VIRTUAL_UNION, l4, l4);
              flick_l4_decode_end_glob (8);
            }
          }
      }
      flick_l4_client_end_decode ();
      return _return;
    }
```

The hand-coded stub uses the knowledge about the IPC implementation directly and looks like the following:

```
L4_INLINE int foo_bar(int t1) {
    l4_msgdope_t _result;
    int _error;
    dword_t _return_code;
```

	Flick	IDL^4
client stub marshal	71	0
IPC send	326	315
server switch	96	33
server unmarshal	32	25
server function in	41	33
server function out	39	26
server marshal	94	32
IPC receive	374	352
client stub error handling	47	35
client stub unmarshal	47	0
sum	1167	851

Table 1: Measurements of Performance of Short IPC (in cycles)

```
5       dword_t _return;
        _error = l4_i386_ipc_call (server_id,
            L4_IPC_SHORT_MSG, test_foo_opcode, t1,
            L4_IPC_SHORT_MSG, &_return_code, &_return,
            L4_IPC_NEVER, &_result);
10      return (int)_return;
    }
```

Because the IDL^4 generated code does not yet contain general error-checking code, the hand-coded stub and the IDL^4 generated stub look very similar. Another improvement, which I did not write down, is to use the assembler code for the IPC call directly, thus avoiding the use of the C-bindings of the IPC calls.

All generated client stubs are `inline` functions, which allow the compiler to include their code directly into the calling environment and to optimize the client stub's code in the surrounding context (registers etc.). Thus unnecessary copy operations to and from the stack are avoided.

I measured the client stub times and the times needed at the server side. The numbers can be seen in Table 7.1. First I needed to know, which overhead is introduced to the call of a function by using an IPC. Then it has been interesting, which parts of the total overhead are unavoidable (IPC) and which can be optimized by the compiler (marshaling, unmarshaling, etc.).

As the measurements show, it is possible to optimize the marshaling and unmarshaling code of the stubs for the short-IPC even further. The difference between the IDL^4 stub and the Flick stub is 316 cycles, which is about 25% performance enhancement. The numbers for the hand-coded stub are not listed, because they are very similar to the IDL^4 generated stubs. A hand-coded stub can perform better, if the developer uses special knowledge about the client and server (he can make assumption the compiler

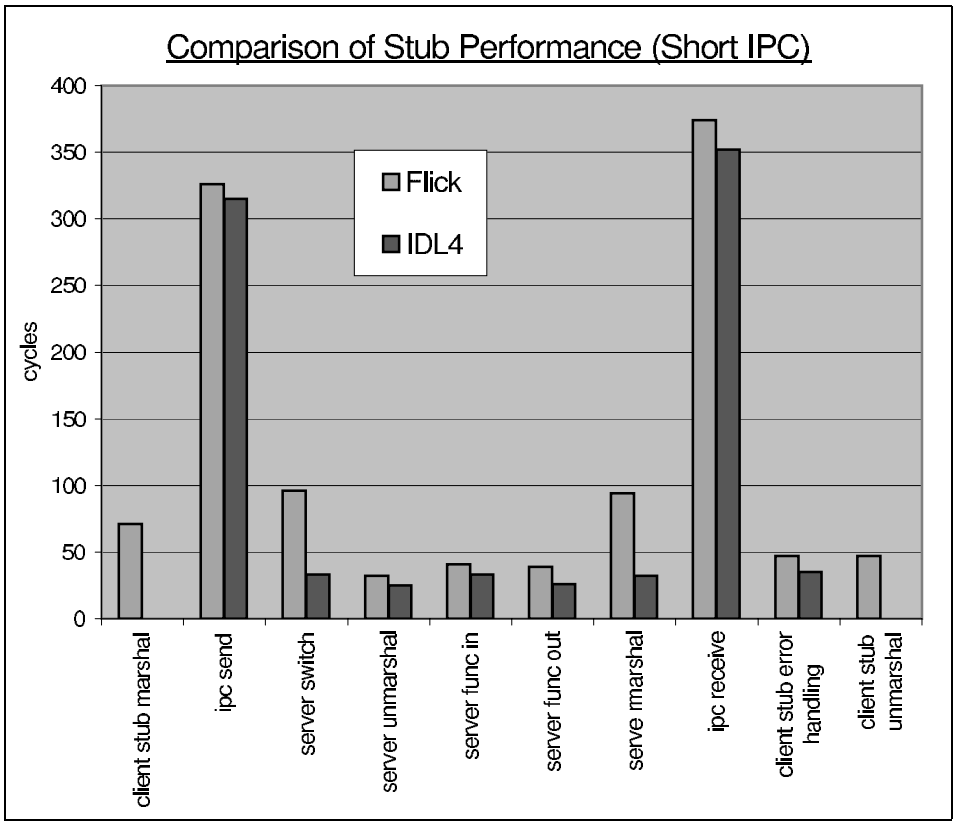

Figure 4: A comparison of the performance of the stubs parts for a Short-IPC.

cannot make) and writes the communication code as pure assembler code.

To see the differences in between the single parts of the whole stub, I visualized the numbers in Figure 5 and Figure 6. The single parts are named on the right and start in the figure at the top (twelve o'clock) and continue in clockwise direction. It can be seen, that the management code (marshaling, unmarshaling, server switch statements, etc.) make up a larger portion in the Flick stub, than in the IDL[4] stub. This difference becomes very clear, when looking at the Figure 4.

The total overhead introduced by the client stub compared to a usual function call is about 400 cycles when entering the function and about the same, when leaving it. Of these 400 cycles the pure IPC makes up 75% – about 300 cycles. With the total overhead costs of about 200 cycles (100 when entering and 100 when leaving the function) we can perform an remote procedure invocation if the parameters fit into a short-IPC.

I expect this value to approximately enlarge by the factor 1.5 as soon as the stubs and server side functionality encloses a full error-handling facility.

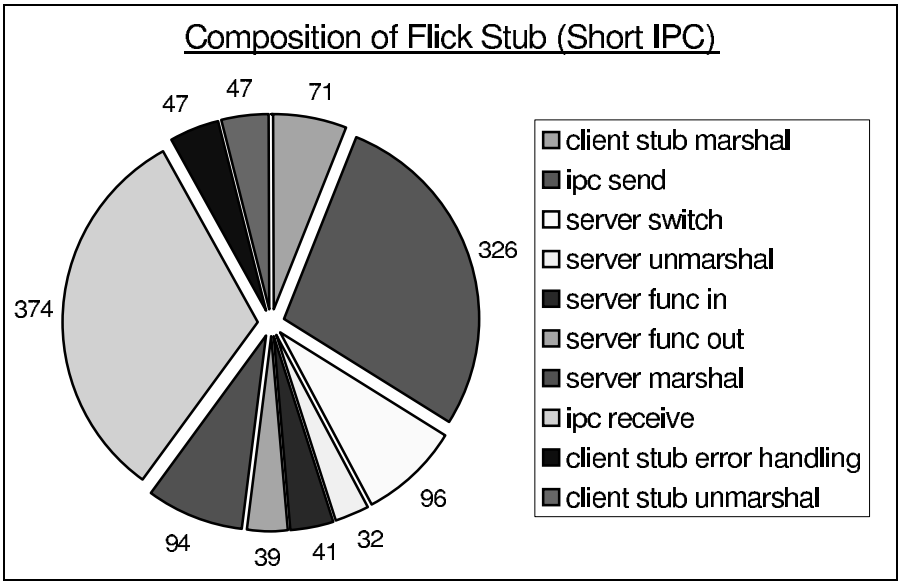

Figure 5: The shares of the single parts of the Flick stub.

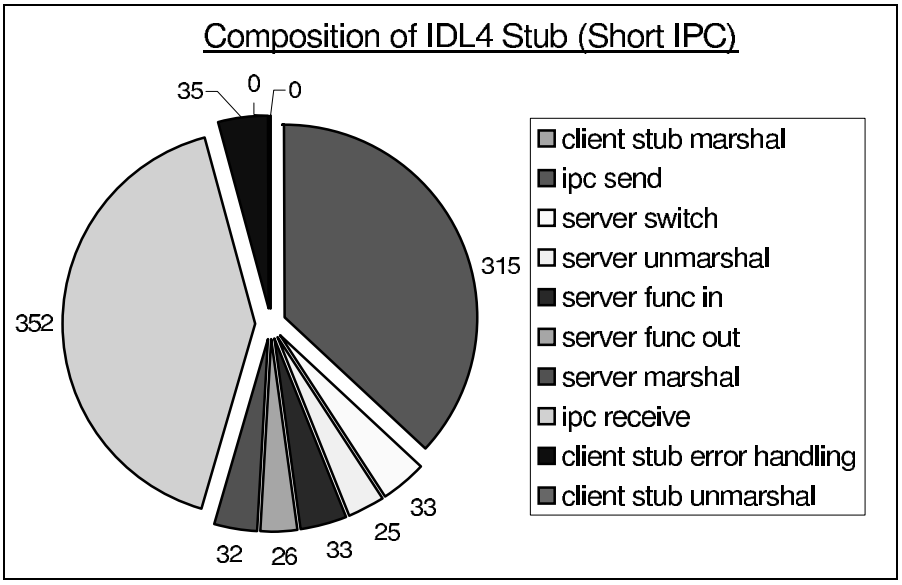

Figure 6: The shares of the single parts of the IDL^4 stub.

	IDL[4]	Flick	hand-coded
client marshal	5	78	4
IPC send	500	496	511
server switch	33	86	33
server unmarshal	31	145	33
server function in	37	58	22
server function out	34	66	22
server marshal	26	144	33
IPC receive	626	669	619
error handling	26	33	29
client unmarshal	23	57	22
sum	1341	1832	1328
sum[5]	189	634	169

Table 2: Comparison of marshaling times of variable sized string (in cycles)

If the error handling code can be written in such a way, that it introduces minimal overhead if the common case – no error – is executed, the error-handling might enlarge the stubs not at all.

7.2 Complex Data Types

To measure complex data types, I chose a "real-world example". I used the interface specification of a filesystem and extracted the `read` function. This function is used quite often and a performance enhancement would multiply by the times of its use. The interface description consists of the following lines.

```
interface filesystem {
    long read([in] handle_t handle, [in,out] long pos,
              [in,out] long len, [out, size_is (len)] char* data);
}
```

The generated stubs have to send four `dword`s from the client to the server and expect to receive two `dword`s and one data array back from the server. The data array or `byte` array is transmitted using an indirect string. To transmit an indirect string only two values have to be provided: the start address of the array and the size of the array in bytes. The marshaling code for indirect strings stays the same no matter which size the indirect string is. For my measurements I send a character array with the size of twelve bytes, containing the string "Hello World!". The size of the indirect string has only effects on the time the IPC needs to transmit this byte array. Because this time stays the same for all stubs, I did not vary the size of the byte array. The results of the measurements are in Table 7.2.

To underline the possible performance gains with IDL[4], I summarized only the marshaling parts, without the IPC and error-handling. The dif-

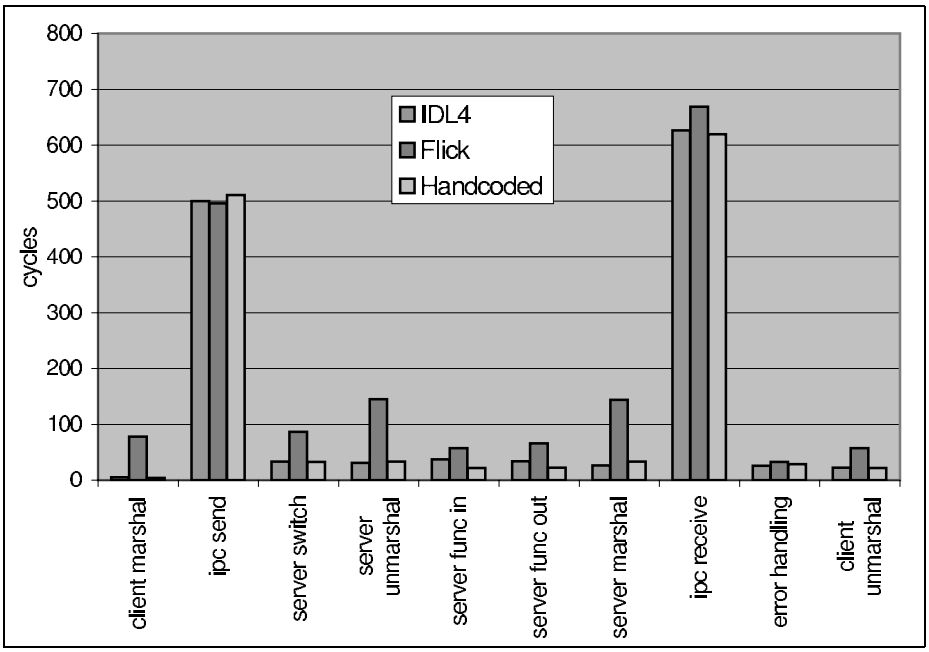

Figure 7: Comparison of marshaling stubs for complex data structures.

ference between the IDL^4 generated marshaling code and the hand-coded marshaling stub are negligible. The Flick marshaling code performs more than three times slower than the IDL^4 generated marshaling code. The share of the marshaling code in a Flick generated stub is about 35%, whereas the share of the marshaling code in a IDL^4 generated stub is about 15%. I visualized the differences between the three stub in Figure 7.

As I mentioned, does the marshaling performance not change if the byte array has a different size. The IPC's time increases, because it has to copy more data from one address space to the other. Further investigations have to be made to find alternative ways to transmit large amounts of data, such as memory shares, etc.

Because the IDL compiler can have no direct influence on the performance of the IPC, I did not investigate the use of variable sized arrays any further. I rather compared the marshaling times for other complex data types. Complex data types can be, for instance, structured types, which are composed from other types, or array of other data types. Because arrays with a variable size are usually marshalled using indirect strings I measured the performance of marshaling fixed sized arrays. Flick generated stubs for fixed-sized array, which marshal the array element by element. IDL^4 generated stub copy the array as a whole into the message buffer using the `memcpy` routine. In Table 7.2 the results for the measurements of marshaling stubs for various sizes of fixed-size arrays can be found.

size	Flick	`memcpy`
1	45	2
2	54	2
4	65	2
8	84	6
16	129	15
32	326	46
64	481	82
128	805	128
256	1447	273
512	2729	563
1024	5301	719
2048	10477	998
4096	20725	1811

Table 3: Comparison of marshaling times of variable sized byte arrays (in cycles)

Structured data types are marshaled by Flick generated stubs also on a element by element basis. The IDL^4 compiler analyzes the structured data type to find out, whether it can copy it as a memory region at once, or if it has to marshal it element by element. So for most cases it is possible to marshal a complex data type using a direct memory copy operation, instead of copying element by element.

8 Conclusion, Open Topics and Future Work

The current implementation of the IDL compiler shows, that it is possible to use techniques, known from common programming language compilers, with IDL compilers. This knowledge must be applied to IDL compilers to meet the requirements for IDL compilers of today.

The IDL^4 compiler is a good approach to incorporate these techniques into an IDL compiler. The ideas have to be investigated further and strategies have to be researched how to incorporate more sophisticated ideas into this compiler.

The optimization techniques can and must be further enhanced. Measurements of copy operations must be performed to incorporate a cost minimization policy into the compiler. Former measurements of copy operations can be used to implement first prototypes of cost-minimization algorithms.

The optimizer should support all data types available in the IDL in the near future, which means that it shall be able to optimize the use of them.

Because DROPS is a component based system, it should be, for instance, one of the goals to develop a component model based on IDL^4.

To have a fast and sufficient implementation of the compiler it supports only the DCE IDL right now. To be able to reach a broader user community, other IDLs and target platforms should be supported, which could be the CORBA IDL or different programming languages as well as different communication APIs, such as the L4 API version X.0.

The support of annotation files has to be pushed to be able to configure the communication path in more detail.

9 Summary

The development of an IDL compiler today involves almost the same complexity as the development of a common programming language compiler. The interface description language is not as complex as a common programming language, but the compiler's mechanisms are the same. Today's IDL compilers must implement the same strategies as usual programming language compilers do.

An IDL compiler is a necessary tool to ease the development of component based systems. But to be of real help, an IDL compiler has to generate highly optimized target code. It has to implement much of the knowledge the developer would use himself, if he would hand-code the stubs.

I think I implemented an IDL compiler, which eases the development of components and generates fast client stubs. Thus it is not simply a compiler to straighten software design, but it is a good development tool.

10 Acknowledgements

I like to thank my professor H. Härtig, my coach L. Reuther, my fiance Z. Tradel and all those who helped my finish this paper "on time". I like to thank especially V. Uhlig for his technical assistance and all those hours of discussion.

Thank you!

References

[1] ISO/IEC DIS 14750, March 1999. http://www.iso.ch/cate/d25486.html.

[2] Thomas J. Brando. Comparing dce and corba. Document MP 95B-93, MITRE, March 1995.

[3] D. D. Clark and D. L. Tennenhouse. Architectural considerations for a new generation of protocols. In *SIGCOMM '90 Symposium*, pages 200–208, 1990.

[4] R.P. Draves et al. "MIG - the mach interface generator". Technical report, Carnegie Mellon University, August 1989.

[5] Eric Eide, Kevin Frei, Bryan Ford, Jay Lepreau, and Gary Lindstrom. "Flick: A Flexible, Optimizing IDL Compiler". In *PLDI '97*, 1997.

[6] Eric Eide, Jay Lepreau, and James L. Simister. *"Flexible and Optimized IDL Compilation for Distributed Applications"*, volume 1511 of *Lecture Notes in Computer Science*, chapter Language, Compilers, and Run-Time Systems for Scalable Computers, pages 288–302. Springer, May 1998.

[7] Eric Eide, James L. Simister, Tim Stack, and James Lepreau. "flexible IDL compilation for complex communication patterns". *Scientific Programming*, 1999.

[8] Bryan Ford, Mike Hibler, and Jay Lepreau. "Separating Presentation from Interface in RPC and IDLs". Technical Report UUCS-95-018, University of Utah, December 1994.

[9] Bryan Ford, Mike Hibler, and Jay Lepreau. "Using Annotated Interface Definitions to Optimize RPC". Technical Report UUCS-95-014, University of Utah, March 1995.

[10] A. Gokhale and D.C. Schmidt. Measuring performance of communication middleware on highspeed networks. *Computer Communication Review 26*, 4, October 1996.

[11] Object Management Group. Corbaservices: Common services specification, revised edition, March 1995. 95-3-31 edition.

[12] Andreas Haeberlein, Jochen Liedtke, Yoonho Park, Lars Reuther, and Volkmar Uhlig. Stub-code performance is becoming important. In *WIESS 2000*, October 2000.

[13] Graham Hamilton and Sanjay Radia. Using interface inheritance to address problems in system software evolution. Technical Report SMLI TR-93-21, Sun Microsystems Laboratories, Inc., November 1993.

[14] Jochen Liedtke. On μ-Kernel construction. In *The 15th ACM Symposium on Operating System Principles*, December 1995.

[15] Jochen Liedtke. L4 reference manual - version 2.0. 486, Pentium, Pentium Pro, September 1996.

[16] Jochen Liedtke. Towards Real Microkernels. *Communications of the ACM*, 39(9):70–77, September 1996.

[17] Owen Tallman and J. Bradford Kain. "COM versus CORBA: A Decision Framework". *Distributed Computing*, September-December 1998.

[18] Volkmar Uhlig. A multi-server filesystem and development environment. Master's thesis, Dresden University of Technology, October 1999.

A Example Compilation

The described example compilation uses a very simple IDL to keep the explanations short:

A.1 The IDL File

```
/* idl file for sample interface */

// this typedef may also occur in the interface body
typedef struct {
  int a;
  int b;
  struct {
    int x;
    int y;
  } c;
} struct_t;

interface foo {
  int bar([in] struct_t *t1);
}
```

A.2 The Front-End Representation

The front-end representation consist of a tree like structure, containing all data read from the IDL file. The root of this structure is used to collect all top level elements of this structure. The tree for the above structure would look like this:

- `CFERoot` – contains all top level elements
  - `CFETypedDeclarator` – contains the `typedef`
    - `CFEStructType`
      - `CFETypedDeclarator` (`a`)
        - `CFESimpleType` – type integer
      - `CFETypedDeclarator` (`b`)
        - `CFESimpleType` – type integer
      - `CFETypedDeclarator` (`c`)
        - `CFEStructType` – structure type
          - `CFETypedDeclaratpr` (`x`)
            - `CFESimpleType` – type integer
          - `CFETypedDeclarator` (`y`)
            - `CFESimpleType` – type integer
  - `CFEInterface` – interface "foo"
    - `CFEInterfaceHeader` (contains names and attributes)
      - `CFEIdentifier` – the name of the interface
      - `CFEVersionAttribute` – version attribute
    - `CFEInterfaceBody` – all elements of the interface

- `CFEOperation` – the "bar" function
  - `CFESimpleType` – return type
  - `CFEIdentifier` – name ("bar")
  - `CFETypedDeclarator` (`t1` – 1st parameter)
    - `CFEUserDefinedType`
      - `CFEIdentifier` – name of user defined type
  - `CDRFunction` – reference to corresponding DR function

A.3 The Data-Representation

Data-representation exist on a per operation basis. This means the data-representation is kept very shallow and simple and belongs to a specific operation. Because data-representations also exist for all complex or user-defined data-types, these data-representations are kept at a central place to allow optimizations of these data-representations in a central place.
For the above operation, the data-representation tree would look like this:

- `CDRFunction` – collects all parameters
  - `CDRSimpleParameter` – the return value
  - `CDRConstructedParameter` (`t1`)
    - `CDRSimpleParameter` (`a`)
    - `CDRSimpleParameter` (`b`)
    - `CDRConstructedParameter` (`c`)
      - `CDRSimpleParameter` (`x`)
      - `CDRSimpleParameter` (`y`)

A.4 Optimization

The optimization module, currently does not perform many tasks. It flattens constructed data types and sorts parameters for their size. It performs all these operations on the data-representation. Thus the above data-representation would look after optimizing like this:

- `CDRFunction` – collects all parameters
  - `CDRSimpleParameter` – the return value
  - `CDRSimpleParameter` (`t1.a`)
  - `CDRSimpleParameter` (`t1.b`)
  - `CDRSimpleParameter` (`t1.c.x`)
  - `CDRSimpleParameter` (`t1.c.y`)

A.5 The Back-End Representation

The back-end representation creates the single back-end classes dynamically and calls their `Write` methods. I describe the basic ("logical") structure of the back-end and line out the `Write` methods, which are called. Each method uses the front-end's in-memory representation to pick out the relevant information and create the appropriate back-end classes. It uses the back-end context, which looks like this:

- `CBEContext`
  - `nOptions` – the arguments of the IDL[4]
  - `CBEFile` – the current output file
  - `CL4BEClassFactory`
  - `CL4BENameFactory`

The order and grouping the single back-end classes appear in, could look for the above IDL like the following one, if IDL[4] is called with the following argument: `--create-inline`:

- `CBERoot::Write()`
  - → `CBERoot::WriteInterfaces()`
    - → `CL4BEClient::Write()`
      - → `CL4BEClient::WriteHeader()`
        - → `CL4BEClient::WriteRemarks()`
          - prints copyright comments to the target file
        - → `CL4BEClient::WriteIncludes()`
          - prints the `#include` directives
        - → `CL4BEClient::WriteTypeDefs()`
          - iterates over all `typedef`s of the interface
          - → `CBETypedDeclarator::Write()`
            - → `CBETypedDeclarator::WriteAttributes()`
              - iterates over attributes of declarator
              - → `CBEAttribute::Write()`
                - prints the name of the attribute as comment
            - → `CBETypedDeclarator::WriteType()`
              - → `CBEType::Write()`
                - because this is a `struct`:
                  - `struct` and the name are written
                  - for all members a new `CBETypedDeclarator` is created
                  - → `CBETypedDeclarator::Write()`
            - → `CBETypedDeclarator::WriteDeclarators()`
              - iterates over declarators

- → `CBEDeclarator::Write()`
  - · writes the variable name

→ `CL4BEClient::WriteConstDecl()`
- · iterates over all constant declarations
- · non constant available

→ `CL4BEClient::WriteFunctions()`
- · iterates over all functions of the interface
- → `CL4BEFunction::Write()`
  - → `CL4BEFunction::WriteInlinePrefix()`
  - → `CL4BEFunctionHeader::Write()`
    - → `CL4BEFunctionHeader::WriteAttributes()`
      - · iterates over attributes
      - → `CBEAttribute::Write()`
    - → `CL4BEFunctionHeader::WriteDeclaration()`
      - → `CBEType::Write()` – return type
      - · print the function's name
      - → `CL4BEFunctionHeader::WriteParameters()`
        - → `CL4BEFunctionHeader::WriteBeforeParameters()`
          - · print additional parameters
        - · iterate over declarators
        - → `CBETypedDeclarator::Write()`
          - · I'll skip this one, because the principle should be clear now.
        - → `CL4BEFunctionHeader::WriteAfterParameters()`
          - · print additional parameters
  - → `CL4BEFunctionBody::Write()`
    - → `CL4BEFunctionBody::WriteClientStub()`
      - → `CL4BEClientStub::Write()`
        - → `CL4BEMarshaller::WritePrepareMarshalIn()`
          - · prints the variable initialisation
        - → `CL4BEMarshaller::WriteMarshalIn()`
          - · prints the marshaling code
        - → `CL4BEMarshaller::WritePrepareUnmarshalOut()`
          - · prints preparation code for unmarshaling
        - → `CL4BEClientStub::WriteInvoke()`
          - · prints the invocation call
        - → `CL4BEClientStub::WriteInvokeErrorHandling()`
          - · prints handling code for invocation call
        - → `CL4BEClientStub::WriteServerErrorHandling()`
          - · prints handling code for server errors
        - → `CL4BEMarshaller::WriteUnmarshalOut()`
          - · prints unmarshaling of return parameters
        - · print return statement

→ `CL4BEClient::WriteImplementation()`
- · not executed, because inline functions are generated

→ `CL4BEServer::Write()`
  · I'll skip this one, because the principle should be clear now.
→ `CBEOpcodes::Write()`
  · I'll skip this one, because the principle should be clear now.

A.6 The Target Files

File 1 (`test_client.h`): This file contains the complete client stub. The marshaling happens on line 33.

```
#ifndef __TEST_CLIENT_H__
#define __TEST_CLIENT_H__

#include "l4.h"
#include "test_opcodes.h"

typedef {
    int a;
    int b;
    struct {
        int x;
        int y;
    } c;
} test_struct_t ;

L4_INLINE
int test_foo( l4_idl_service_t *_service,
    /* in */ test_struct_t *t1,
     l4_idl_connection_t *_connection)
{
    l4_msgdope_t _result;
    int _error;
    int _return;
    dword_t _return_code;
    struct {
        l4_fpage_t fpage;
        l4_msgdope_t size;
        l4_msgdope_t send;
        dword_t dwords[6];
    } _msg_buf_in;
    _msg_buf_in.size = L4_IPC_DOPE(6, 0);
    _msg_buf_in.send = L4_IPC_DOPE(6, 0);
    memcpy(&(_msg_buf_in.dwords[2]), t1, 4);
    _error = l4_i386_ipc_call (_service->server_id, &_msg_buf_in,
        test_foo_opcode, 0, &_msg_buf_out, &_return_code,
        (dword_t*)&_return, _connection->timeout, &_result);
    if (_error)
    {
        THROW_EXCEPTION(_connection, _error);
        return (int)-1;
    }
    if (_return_code)
    {
        // TODO: insert error handling code for server errors.
        return (int)-1;
    }
    return _return;
}
#endif // __TEST_CLIENT_H__
```

File 2 (`test_server.h`): This file contains the declarations of the server skeleton and the server loop. It also defines the structure for the server side.

```
#ifndef __TEST_SERVER_H__
#define __TEST_SERVER_H__

#include "l4.h"

typedef {
    int a;
    int b;
    struct {
        int x;
        int y;
    } c;
} test_struct_t ;

int test_foo(/* in */ test_struct_t *t1);

/* the server loop function declaration */
int test_server_loop(void);

#endif // __TEST_SERVER_H_
```

File 3 (`test_opcodes.h`): The function identifier for the function `foo` is defined here.

```
#ifndef __TEST_OPCODES_H__
#define __TEST_OPCODES_H__


#define test_base_opcode 1
#define test_foo_opcode filesystem_base_opcode + 0

#endif // __TEST_OPCODES_H__
```

File 4 (`test_serverloop.c`): The server loop.

```
#include "test_opcodes.h"
#include "test_server.h"

int test_server_loop ()
{
    dword_t _opcode, dw1, _return_code;
    l4_threadid_t _id;
    l4_msgdope_t _result;
    int _error;
    // msg buffer
    struct {
        l4_fpage_t fpage;
        l4_msgdope_t size;
        l4_msgdope_t send;
        dword_t dwords[6];
    } _buffer;
    // init buffer
    _buffer.size = L4_IPC_DOPE(6, 0);
    _buffer.send = L4_IPC_DOPE(2, 0);
    // first receive
    do {
        _error = l4_i386_ipc_wait(&_id, &_buffer, &_opcode, &dw1,
            L4_IPC_NEVER, &_result);
    } while ((_error == L4_IPC_ENOT_EXISTENT) ||
             (_error == L4_IPC_REMSGCUT));
    // the server loop
    while (!_error) {
        switch (_opcode) {
        case test_foo_opcode:
            {
                test_struct_t *t1;
                int _return;
                memcpy(t1, &(_buffer.dwords[0]), 4);
                _return = test_foo(t1);
                _return_code = 0;
                _buffer.send = L4_IPC_DOPE(2, 0);
                do {
                    _error = l4_i386_ipc_reply_and_wait(_id, &_buffer,
                        _return_code, _return, &_id, &_buffer, &_opcode,
                        &dw1, L4_IPC_TIMEOUT(244,9,0,0,0,0),
                        &_result);
        // error handling for "reply_cancelled_by_another_thread"
        // and "send_aborted_by_another_thread"
                } while((_error == L4_IPC_SECANCELED) ||
                        (_error == L4_IPC_SEABORTED));
            }
            break;
        }
        if (_error) {
            switch(_error) {
            case L4_IPC_ENOT_EXISTENT:
                // client doesn't_exist_any_more
```

```
                case L4_IPC_SETIMEOUT:
                    // client does not respond to reply
55              case L4_IPC_REMSGCUT:
                    // message cut
                    // set server into ready state for other clients
                    // first receive
                    do {
60                      _error = l4_i386_ipc_wait(&_id, &_buffer,
                            &_opcode, &dw1, L4_IPC_NEVER, &_result);
                    } while ((_error == L4_IPC_ENOT_EXISTENT) ||
                             (_error == L4_IPC_REMSGCUT));
                    break;
65              }
            }
        }
        return _error;
    }
```

B gcc Optimizations

To be able to determine, which code is best for the target compiler to optimize, I analyzed different code samples. One of the code sample belongs to the optimization technique *bit-stuffing*. If parameters are stuffed into one `dword` to be eventually transmitted as short-IPC, the bit-positions of the parameters have to change.
Lets say we have four values of type `byte` (8 bits). If we like to stuff them into a `dword` (32 bit), we have to shift the first byte to the highest byte of the `dword`, the second byte to the second highest byte of the `dword`, and so on. Then we have to perform a bit-wise `OR` operation to receive a stuffed `dword`. The code could look like this:

```
dword dw;
byte b1, b2, b3, b4;

dw = (b1 << 24) | (b2 << 16) | (b3 << 8) | b4;
```

If we like to stuff the `byte` values into an array of `dword`s, the code could look like this:

```
dword dw[];
byte b1, b2, b3, b4;
int offset ; // in dwords

dw[offset] = (b1 << 24) | (b2 << 16) | (b3 << 8) | b4;
```

Another approach to stuffing `byte` values into a `dword` array it to cast the `dword` array into a `byte` array and then write the values into the array one by one. This code would look like this:

```
dword dw[];
byte b1, b2, b3, b4;
int offset ; // in bytes

(*(byte*)(&dw))[offset] = b1;
(*(byte*)(&dw))[offset+1] = b2;
(*(byte*)(&dw))[offset+3] = b3;
(*(byte*)(&dw))[offset+2] = b4;
```

This code looks larger, not so elegant and seems to produce also a lot of byte code. But the `gcc` compiler optimizes the latter code more efficiently then the more elegant bit-shifting code.