Automated Code Generation for Integrated Layer Processing

Torsten Braun[1] and Christophe Diot
INRIA Sophia-Antipolis
2004 route des Lucioles, B.P. 93, F-06902 Sophia Antipolis, France
E-mail: t.braun@ieee.org, Christophe.Diot@sophia.inria.fr
Phone: +33 93 65 77 56, Fax: +33 93 65 77 65
[1] now with IBM ENC Heidelberg, Germany

Abstract

ALF (application level framing) and ILP (integrated layer processing) are protocol design and implementation concepts applied in high-performance communication architectures, e.g. to support multimedia applications. Writing ILP code is rather complex and, therefore, ILP code generation tools can reduce the time to develop efficient ILP protocol code significantly. This paper presents a tool, which allows automated ILP code generation based on high-level specifications of user data types. The tool is based on a stub compiler and integrates ILP loops into (un)marshalling routines automatically.

Keywords

protocol implementation, automated code generation, high-performance communication, protocol architecture

1 INTRODUCTION

The performance of processors has been increasing faster than the performance of memory during the last few years. The resulting memory bottleneck can be reduced by avoiding memory access, e.g. by eliminating copy operations from protocol implementations. Another approach is to use on-processor caches, which are almost as fast as processor registers. Data and instruction caches are currently in the range of a few kbytes, but it is expected that fast caches of future processors will grow into the range of several Mbytes. External cache memories already have sizes of several Mbytes.

Integrated Layer Processing (ILP) tries to avoid memory access and to use caches to store intermediate results of data manipulation processing, e.g. encrypted or decrypted data. ILP is an implementation concept which allows to implement all data manipulation steps in one integrated processing loop (ILP loop) (Clark, 1993). Theoretically, an ILP protocol stack implementation reads once from main memory, keeps the read data within registers or cache memory, and performs all the data manipulations for several protocol layers within the ILP loop. Processed data are finally written back to the destination memory. In this ideal case, ILP requires only one read and one write access to the main memory for each basic processing unit, which is usually a 32- or 64-bit-word.

The Application Level Framing (ALF) concept has been developed together with ILP and provides several features that allow ILP. ALF means that the application breaks the data into suitable aggregates called application data units (ADUs), and the lower levels preserve these frame boundaries as they process the data. Therefore, no segmentation, blocking, or other size changing functions are allowed. Another advantage of the fact that an ADU is the logical unit of *all* protocol functions is that an ADU can be processed highly independent of other ADUs. This prevents buffering packets in packet queues as it is for example necessary for reassembly. Buffering is required if packets are split into several packets or if several packets have to combined into a single one. It has been shown that ALF increases the possibility that data are within the cache while being processed by a pipeline of consecutive protocol functions (Braun, 1996). In particular, with small packet sizes and the expected increase of future on-processor caches the probability increases that a packet fits into an on-processor (first-level) cache.

Writing ILP code by hand is not a trivial task. There are several issues increasing the complexity of ILP code:

- Different data manipulation functions manipulate application data in units with different sizes. For example, checksum calculation operate on 16 bit boundaries and encryption functions usually work on 64 bit boundaries. In this case, 4 basic checksum operations must follow an basic encryption operation to process the basic processing unit of 64 bit. An encryption function working on 48 bit and a checksum algorithm working on 32 bit would result in 96 bit basic processing units.
- ILP code is usually highly optimized in order to achieve efficient code.
- Integrating code from different protocol layers may violate the implementation modularity and limit the flexibility. For each combination of data manipulations, new ILP code has to be written. For a large number of possible combinations this may be very time consuming.

Automated ILP code generation is an approach for solving some of the problems outlined above. A simple but limited approach of automated ILP code generation is a macro preprocessor (Massey, 1993). Another approach which is the one we present in this paper is the extension of a stub compiler to generate the ILP protocol code. This means that (un)marshalling routines generated by the stub compiler are extended to integrate processing of data manipulation routines.

Generally, protocol compilers generate automatic code derived from specifications using description languages such as SDL or LOTOS. The main problem of automatic code generation from protocol specification is the performance which is often much worse than manually generated code. One example for efficient code generation based on formal specifications is described in (Castelluccia, 1996). This paper gives also a good overview of related work in automatic protocol generation.

The following section gives an overview of the INRIA stub compiler (ISC) (Braun, 1996), which was the basic tool used to develop our ILP compiler. Section 3 - Section 5 describe the ISC extensions and the tools developed in order to support automated ILP code generation. The ISC extensions and tools have been developed in three steps. Each section describes one of these steps. Section 3 describes the extensions of ISC for ILP loop integration into the (un)marshalling routines. It shows how ILP loop macros are inserted into the (un)marshalling functions. These ILP loop macros have to be written either manually or generated by a tool. Section 4 introduces the tool we developed to generate ILP loop macros automatically. This tool has also been extended in order to support an advanced ALF/ILP protocol architecture. In that case, a single procedure is generated for sending and receiving an ADU performing (un)marshalling, the other data manipulations, initializations and error checks. The extensions are discussed in Section 5 and, finally, Section 6 describes the performance benefits of the developed tool.

2 INRIA STUB COMPILER

In contrast to other stub compilers which are using ASN.1 or XDR language as input language, the INRIA stub compiler (ISC) (Hoschka, 1994) uses an annotated C language to describe data types. ISC generates routines to convert the data into an appropriate network representation format. Currently, ISC supports PER and XDR network formats.

The architecture of ISC is shown in Figure 1. The user describes the format of the messages exchanged by the application in the file *IF.h.anno* using the C language with additional annotations. ISC generates three C files: The first file contains type specifications (*IF.h*), the second the encoding functions to convert messages from its local data representation into the XDR network format (*IF_xdrc.c*), and the third one contains the decoding functions to decode a message from the XDR network format into its local data representation (*IF_xdrd.c*). In the final step, a C compiler translates the C files into object code. The object code can be linked to an application in order to provide (un)marshalling functions.

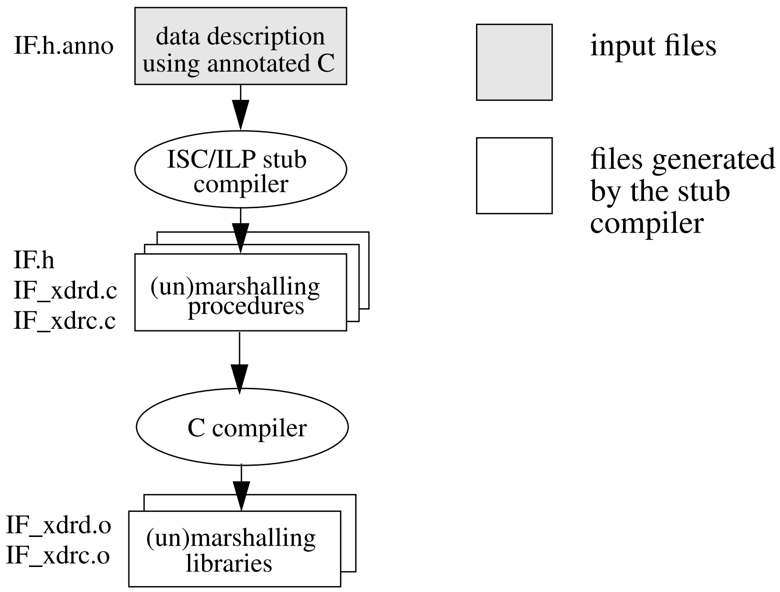

Figure 1 The INRIA Stub Compiler (ISC) (Hoschka, 1994)

3 ILP EXTENSIONS TO ISC

3.1 Architecture

The ILP extensions to ISC (ISC/ILP) described hereafter extend the marshalling and unmarshalling routines by integrating data manipulations processing. ILP performs several data manipulations such as en(de)cryption or checksum calculation within a single loop. The ILP extensions for ISC allow that together with (un)marshalling other data manipulations such as encryption or checksum calculation are performed on the data.

The data manipulations for sending are performed after marshalling a message, the data manipulations for receiving are performed before unmarshalling. The extensions only allow data manipulations, which are logically on a lower level than (un)marshalling (Figure 2). Furthermore, ISC/ILP supports XDR only as network representation format. XDR has the characteristic that it is based on 4 byte boundaries. Therefore, the unit of 4 bytes is selected as the minimum basic processing unit size for all data manipulations. Figure 2 shows the relation between (un)marshalling and the other ILP data manipulation operations:

1. The marshalling operation reads the data to be marshalled from the main memory into registers or cache memory.
2. The marshalling operations are performed on the registers and the marshalled data are stored again in the registers.
3. The other data manipulations combined into the ILP loop operate on the registers.
4. The data is finally written into the destination memory.

The processing steps for received data are the following:

1. Received data is read into the registers.
2. The data manipulations are performed on the registers.
3. Finally, the unmarshalling operation writes the unmarshalled data into the destination memory.

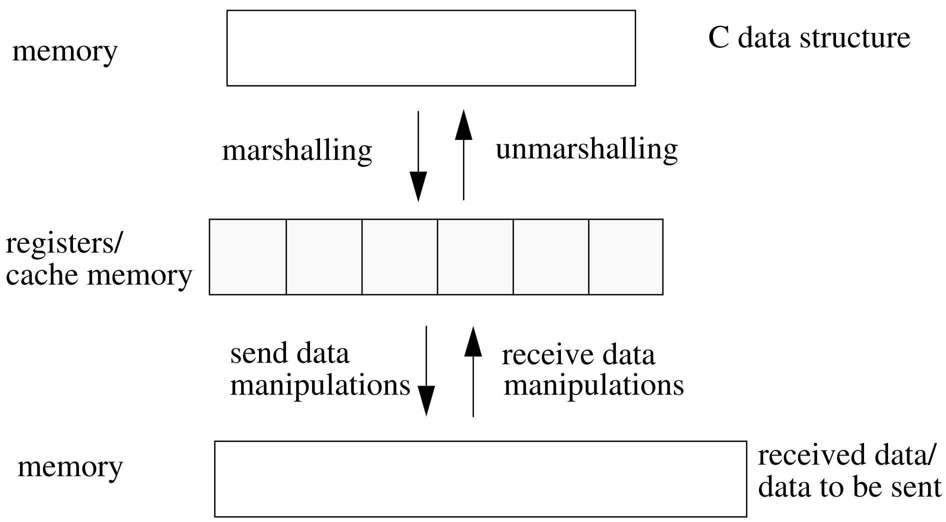

Figure 2 Integration of (un)marshalling routines and data manipulations

ISC/ILP requires as input a description of the data types using an annotated C language. In Figure 3, the file *IF.h.anno* contains this data type description. ISC/ILP generates the C files of the (un)marshalling procedures by adding some additional code fragments, which allow ILP data manipulations during (un)marshalling. In particular, some macro patterns for the ILP loop are inserted into the (un)marshalling routines.

These ILP loop macros (`data_manipulations_send` and `data_manipulations_receive`) must contain the code for the ILP loop to be integrated. They have to be defined in the include file *IF_ilp.h*. This file must be included into the (un)marshalling procedures (*IF_xdrc.c* and *IF_xdrd.c*). Example code for the (un)marshalling procedures is given in Section 3.2. ISC/ILP also generates an include file *IF.h* containing the final C structures for application programs.

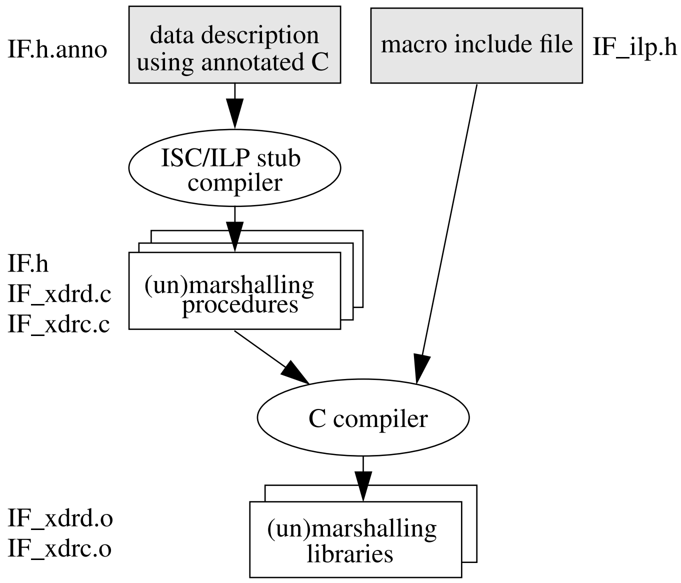

Figure 3 ILP extension of the INRIA Stub Compiler

3.2 ILP marshalling procedures

ISC/ILP generates the files *IF_xdrc.c* and *IF_xdrd.c* containing the (un)marshalling functions. The `data_manipulations_send` macro pattern and the `data_manipulations_recv` macro pattern are inserted into the marshalling and unmarshalling routines, respectively. The application programmer has to generate an include file *IF_ilp.h* to define the appropriate data manipulations macros, which have to be executed for marshalling and unmarshalling the data types described in the annotated C file *IF.h.anno*. Section 4 describes a tool that can be used to generate the macro patterns automatically.

The interface of the data manipulation macros for sending and receiving are shown in the following code example of the files *IF_xdrc.c* and *IF_xdrd.c*, which are generated by ISC/ILP. The parameter `intreg` is the 4 byte register, to which the marshalled data are written, while `write_ptr` and `read_ptr` are pointers to the final memory address of the data. The parameter `ilp_var` is a variable for internal data manipulation purposes, the data structure must be defined within the file *IF_ilp.h*. The `ILP_VAR_TYPE` structure may contain some components required for the several data manipulations, e.g. an intermediate variable for storing the result of the checksum calculation.

The ILP macros are embedded within `#ifdef - #endif` statements in the files *IF_xdrc.c* and *IF_xdrc.c*. A macro is inserted after writing each 4-byte-word to the destination memory. The marshalling output is written into a register and all the data manipulations on this register are performed before writing it to the final memory. The following code shows the (un)marshalling routine for the data type `DATATYPE`.

The macros are inserted in the unmarshalling functions before each 4-byte-word is unmarshalled. The selection whether ILP data manipulations are included using inlining or not is done by a compiling the *IF_xdrc.c* and *IF_xdrd.c* files with the corresponding compilation directive `ILP`.

IF_xdrc.c:

```
int * DATATYPE_xdrc(ilp_var, write_ptr, type_ptr)
ILP_VAR_TYPE       * ilp_var;
int                * write_ptr;
DATATYPE           * typeptr;
{
#ifdef ILP
     int intreg;
#endif
#ifdef ILP
     intreg = ...; /* marshalling and storing the next 4 bytes in register*/
                   /* ILP data manipulations and write data according to */
                   /* the write_ptr to the memory                       */
     data_manipulations_send(intreg, write_ptr, ilp_var);
     write_ptr++;  /* increment the write_ptr                           */
#else
     *write_ptr++ = ...;/* write marshalling output directly to memory */
#endif
     ...
     return(write_ptr);
}
```

IF_xdrd.c:

```
int * DATATYPE_xdrd(ilp_var, read_ptr, end_ptr, type_ptr)
ILP_VAR_TYPE       *ilp_var;
int                *read_ptr,*end_ptr;
DATATYPE           *type_ptr;
{
#ifdef ILP
     int intreg;
#endif
     int *read_ptr = (int *) type_ptr;
#ifdef ILP
                   /* read received data, perform ILP data manipulations */
                   /* and store the result in the register              */
     data_manipulations_receive(read_ptr, intreg, ilp_var);
     read_ptr++;   /* increment read_ptr                                */
     ... = intreg; /* unmarshalling                                     */
#else
     ... = *read_ptr++; /* read and unmarshall directly                 */
#endif
     ...
     return(read_ptr);
}
```

The application programming interface (API) of the encoding and decoding functions for each of the (un)marshalling functions can also be found in the code example. The first parameter of both functions is a pointer to a variable (`ilp_var`) for internal data manipulation purposes. In addition, the marshalling function requires pointers to an output buffer (`write_ptr`) and to the variable to be marshalled (`type_ptr`). It returns a pointer to the end of the output buffer after marshalling. The unmarshalling function needs similar parameters: a pointer to the input buffer (`write_ptr`), a pointer to the end of the input buffer (`end_ptr`), and a pointer to a variable for the unmarshalled data (`type_ptr`). The function again returns a pointer to the end of the unmarshalled message.

4 ILP LOOP GENERATION

The purpose of this tool is to combine the various data manipulation functions into a single ILP loop that will be provided to the modified ISC/ILP stub compiler by the macro library *ilp.h* in Figure 3. ILP is only efficient, if the data manipulations are inlined and if function calls are avoided (Braun, 1995). Using function calls would cause N function calls for each processing unit with N as the number of different data manipulations. Therefore, it is necessary to implement all data manipulations within a macro library if it is not sure that the compiler expands all functions to be inline macros.

A major problem with generating the ILP loop is putting all the data manipulations into the correct order and connecting them to overcome the mismatch of the different processing unit sizes of the various data manipulations. For this task an ILP configuration tool *ilpconfig* has been developed. Several characteristics as the order and the processing unit sizes of the various data manipulations have to be specified using a special but simple configuration language.

4.1 ILP configuration tool

The ILP configuration tool takes the data manipulation specification and generates the macros `data_manipulations_send` and `data_manipulations_receive` for the send and the receive ILP loops using a macro library (*ilp.h*) predefined for the different data manipulations (Figure 4). The include file generated by the ILP configuration tool is finally included by the (un)marshalling routines. The developed configuration tool simplifies ILP programming further. The user has only to describe the data structures using the annotated C language (*IF.h.anno*) and the configuration of the data manipulations within the ILP loop (*IF_ilp.cfg.h*) as described in Section 4.2.

The ILP configuration tool generates the files *IF_ilp.h* and *common_ilp.h*. The file *IF_ilp.h* includes the ILP loops for sending and receiving data. The file *common_ilp.h* contains a definition of a data structure for internal purposes and for variables and results required by the different data manipulation functions. Furthermore, it contains an initialization routine for initializing variables of the defined data structure. The configuration tool may be used for different input files containing data description using annotated C language. In that case, the file *common_ilp.h* contains the data structure, which is common to all processed input files.

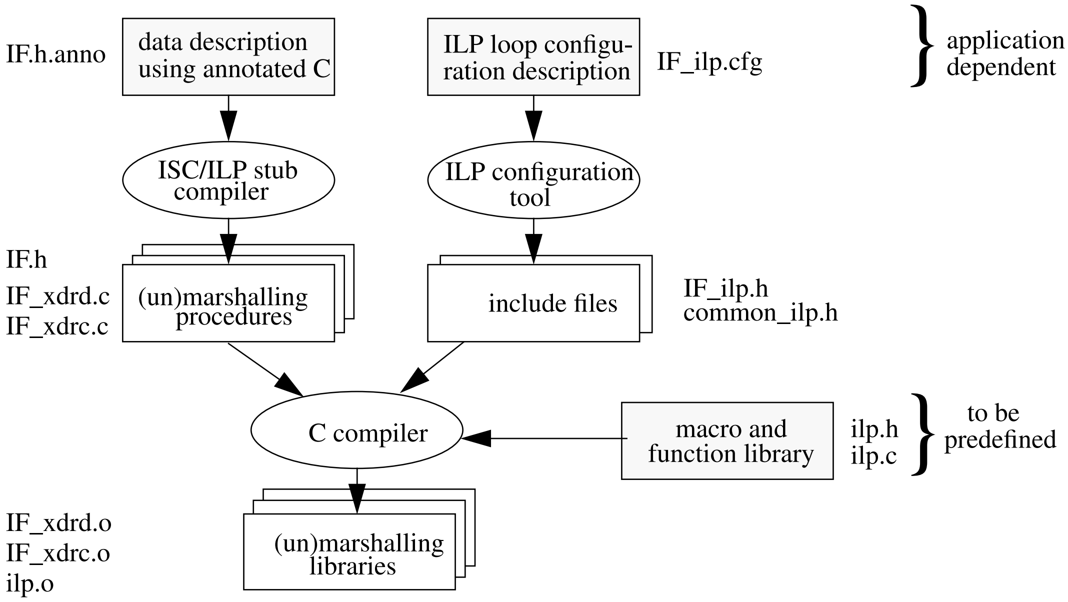

Figure 4 ILP Configuration Tool

4.2 Configuration language

To specify the desired ILP loop for sending and receiving data, a simple ILP configuration language has been developed. This language relieves the application programmer from implementing an ILP loop for each possible combination of data manipulations. Instead, it is only necessary to provide a macro library (*ilp.h*) and a C library (*ilp.c*). The file *ilp.h* must contain the macros for the different data manipulations according to a certain uniform format described below. The file *ilp.c* contains mainly some initialization functions and some functions to process the result after the ILP loop.

```
#define macro_name(integer1,...,integerN,parameter1,...,parameterN) {...}
```

If the basic processing unit size of a data manipulation is N*4 bytes, N integer registers are required in the parameter list. In addition to these registers, the parameter list may contain additional parameters, which have to be described in the ILP description using the RESULT and the VARIABLE keywords. Each data manipulation takes the N registers as input, manipulates them, and substitutes the initial contents by the final result.

A complete ILP loop configuration description consists of a set of several data manipulation functions. Each single data manipulation function is described within a separate block independent of the other functions. Therefore, each ILP description (*IF_ilp.cfg*) consists of M blocks, with M as the number of data manipulation functions of the ILP loop. The beginning of a block is indicated by `[BLOCK_NAME]`. Each block itself consists of an arbitrary number of assignments. For one data manipulation function there exists exactly one block for both the send and receive data manipulation.

```
description:
     block {block}

block:
     "["blockname"]" assignment {assignment}

assignment:
     keyword"="value

keyword:
     "POSITION"|"NEXT"|"MACRO_NAME_S"|"MACRO_NAME_R"|"LENGTH"|"VARIABLE"|
     "RESULT"|"FINAL_S"|"INIT"
```

There are several assignments for each block. Some of them are mandatory and must appear in each block (M). Others are optional, but must appear with a certain value exactly once in a description file(O^1) or at most once in a block (O). Finally, there are optional keywords, which may appear arbitrarily often within a block (A). The allowed keywords are shown in Table 1 . Several data manipulations require other parameters for initialization (e.g., key tables for encryption) or results (e.g., checksum calculation). To specify additional parameters, the `VARIABLE` and `RESULT` assignments must be used. The assignment

```
VARIABLE=(t_name,v_name,i_value)
```

means that a parameter `v_name` of type `t_name` is required. The variable may be initialized with the optional value `i_value`, which is a constant value. The syntax for the `RESULT` assignment is quite similar:

```
RESULT=(t_name, v_name,i_value)
```

The data types for `VARIABLE` and `RESULT` variables are currently limited to simple types such as scalar types or pointers to scalar types. The difference between `VARIABLE` and `RESULT` variables is that `RESULT` variables are used to build an advanced protocol architecture for ILP processing as explained in Section 5. `RESULT` variables may be processed after the ILP loop by special functions. Note that also the following definitions are only necessary to support the advanced ILP protocol architecture as described in Section 5. The special functions are defined using the `FINAL_S` and `FINAL_R` assignments according to the format

`FINAL_S=(function1(res_list1), ..., functionN(res_listN))` or

`FINAL_R=(function1(res_list1), ..., functionN(res_listN)).`

A `res_list` is a list of an arbitrary number of `RESULT` variables, which must be defined the corresponding block. The special functions are processed after the ILP loop and must be implemented in a separate library to be linked to the application program. The input parameters of these functions must match the sequence of their definition in the ILP configuration description. However, the functions expect a pointer value to the specified variable type. The last assignment is the `INIT` assignment.

```
INIT=initfunction.
```

The function `initfunction` shall contain some initializations of the internal variable to be used. This function also allows initializations, which must be performed only once, e.g. encryption tables at the beginning of an application program. In contrast to the initializations of the `RESULT and VARIABLE` variables, which are performed for each ILP loop (e.g., for each packet), the initializations described in `initfunction` are performed only once within an application.

Table 1 Configuration language

keyword		**remark**
POSITION	O[1]	`POSITION` allows to describe whether the data manipulation is on highest level (`POSITION=TOP`) or on the lowest level (`POSITION= BOTTOM`). Only the values `TOP` and `BOTTOM` are allowed. There must exist exactly one assignment (`POSITION=TOP`) and exactly one assignment (`POSITION=BOTTOM`) within a description file.
NEXT	M	pointer to the next lower data manipulation function on the send side.
MACRONAME_S	M	macro name for the data manipulation in the send ILP loop.
MACRONAME_R	M	macro name for the data manipulation in the receive ILP loop.
LENGTH	M	length of the basic processing unit size of the data manipulation function. Input processing unit size must equal the output processing unit size. The case of different input and output length parameters is currently not supported.
VARIABLE	A	parameter for initializations
RESULT	A	parameter for results
FINAL_S	O	function for final result processing on the send side
FINAL_R	O	function for final result processing on the receive side
INIT	O	one-time initialization routine

4.3 Example

The following example code shall illustrate the use of the ILP configuration tool. The example shows the description of an ILP loop where SAFERK-64 encryption (Massey, 1993) and TCP checksum calculation are integrated into the (un)marshalling routines. TCP checksum calculation is the lowest data manipulation of the ILP loop.

There are two data manipulations, namely `CHECKSUM` and `ENCRYPTION`. `CHECKSUM` uses a `RESULT` variable with the name `xsum` of type `u_int`, which is initialized by `0` at the beginning of each ILP loop. For each ILP loop the function `TCP_xsum_final` may perform final processing steps on the result variable.

The `CHECKSUM` data manipulation is on the lowest level of the ILP loop (`BOTTOM`), while `ENCRYPTION` is on the highest level of the ILP loop (`TOP`). The top level function is performed first for sending and last for receiving, while the bottom level function is performed first for receiving but last for sending.

The macros being used for the ILP data manipulations (`writereg`, `readreg`, `tcp_checksum`, `saferk64_encrypt`, `saferk64_decrypt`) must be defined in the macro library *ilp.h*, while the file *ilp.c* is used to define functions for initialization and final result processing. `writereg` writes data to be manipulated and sent into registers, while `readreg` reads received data into the registers for receive data manipulations. The ILP configuration tool generates the files *common_ilp.h* and *IF_ilp.h* from these predefined files. The file *common_ilp.h* contains the definition of the internal ILP variable type `ILP_VAR_TYPE` and an initialization procedure required before each ILP loop.

IF_ilp.cfg:

```
[CHECKSUM]
MACRONAME_S=tcp_checksum
MACRONAME_R=tcp_checksum
LENGTH=4
RESULT=(u_int,xsum,0)
POSITION=BOTTOM
FINAL_S=(TCP_xsum_final(xsum))
FINAL_R=(TCP_xsum_final(xsum))

[ENCRYPTION]
MACRONAME_S=saferk64_encrypt
MACRONAME_R=saferk64_decrypt
NEXT=CHECKSUM
LENGTH=8
POSITION=TOP
VARIABLE=(char*,key)
VARIABLE=(u_int*,logtab)
VARIABLE=(u_int*,exptab)
INIT=saferk64_ilp_init
```

The file *IF_ilp.h* contains the macros for ILP processing to be included into the (un)marshalling routines. The data manipulation macro for sending collects the data in 4 byte steps (the basic unit size of the XDR marshalling routine) and performs the data manipulations for each L bytes (L: lowest common multiple (LCM) of the data manipulations). The receive data manipulation macro takes L bytes and delivers the data to marshalling in 4 byte steps. The code example below shows the output of the ILP configuration tool for the example input file *IF_ilp.cfg*.

IF_ilp.h:

```
#include "ilp.h"
#include "common_ilp.h"

/*-----------------ILP_LOOP_SEND-----------------------*/
#define data_manipulations_send(intreg,dst,ilp_ptr)\
switch((ilp_ptr)->counter){\
  case 0:\
    (ilp_ptr)->buffer0=(intreg);\
    (ilp_ptr)->address0=(dst);\
    (ilp_ptr)->counter++;\
    break;\
  case 1:\
    (ilp_ptr)->counter=0;\
    saferk64_encrypt((ilp_ptr)->buffer0,(intreg),(ilp_ptr)->key,\
          (ilp_ptr)->logtab,ilp_ptr)->exptab);\
    tcp_checksum((ilp_ptr)->buffer0,(ilp_ptr)->xsum);\
    tcp_checksum((intreg),(ilp_ptr)->xsum);\
    writereg((ilp_ptr)->buffer0,(ilp_ptr)->address0);\
    writereg(intreg,dst);\
    break;\
}
```

```
/*-----------------ILP_LOOP_RECEIVE--------------------*/
#define data_manipulations_recv(src,intreg,ilp_ptr)\
switch((ilp_ptr)->counter){\
  case 0:\
    (ilp_ptr)->counter++;\
    readreg(src,intreg);\
    readreg(((int*)(src))+1,(ilp_ptr)->buffer0);\
    tcp_checksum((intreg),(ilp_ptr)->xsum);\
    tcp_checksum((ilp_ptr)->buffer0,(ilp_ptr)->xsum);\
    saferk64_decrypt((intreg),(ilp_ptr)->buffer0,(ilp_ptr)->key,\
          (ilp_ptr)->logtab,ilp_ptr)->exptab);\
    break;\
  case 1:\
    (intreg)=(ilp_ptr)->buffer0;\
    (ilp_ptr)->counter=0;\
    break;\
}
```

5 TOOL EXTENSIONS TO SUPPORT AN ALF/ILP PROTOCOL ARCHITECTURE

5.1 ALF/ILP protocol architecture

Several guidelines for future protocol architectures supporting ILP have been proposed in (Braun, 1995). To implement ALF/ILP concepts efficiently, the following design properties are essential:
- fixed size packet headers,
- packet trailers for user data dependent fields (e.g., checksum values), and
- separate packets for user data and control functions

We have developed an ALF-based communication architecture which is based on these proposals. Application data units (ADUs) are the basic processing unit in an ALF-based communication architecture and an ADU is the only data structure to be used for protocol processing. Usually, there exist different types of ADUs, e.g. ADUs for control purposes and ADUs for pure application data, such as graphical data to be displayed on a screen. However, the existence of several ADU types should be transparent for ADU processing in order to allow a single ILP loop for all kinds of ADUs. For composition and parsing ADUs to be sent or received, the (un)marshalling routines of the INRIA stub compiler (ISC) are used. This allows to specify an ADU as an union of several ADU subtypes. The (un)marshalling procedure provides full processing of such an ADU independent of its type. Only one (un)marshalling procedure is necessary in this case for all kind of ADUs.

The input file *IF_ilp.h.anno* for ISC/ILP can be provided by an application programmer or by another tool such as the ALF Compiler Control described in (Braun, 1996). The ALF compiler takes an annotated C specification of ADUs containing only user data and an ESTEREL protocol specification both provided by the application programmer as input. It generates an annotated C specification of user data ADUs with additional control ADUs required for protocol functions described in the ESTEREL protocol specification.

Integrating ILP functions into the (un)marshalling procedures causes two problems:
- Different ILP manipulations require different alignment boundaries.
  If an ADU is not aligned to those boundaries, padding bytes have to be inserted. However, the amount of padding bytes required for the different functions might differ.

- Protocol data unit fields may depend on the ILP data manipulation.
  Putting those fields to the header of an ADU might cause additional copy operations within an implementation or prevents the use of ordering-constraint data manipulation functions (Braun, 1995).

The first problem is solved by calculating the number of padding bytes matching the requirements of all data manipulations. The amount of padding bytes is calculated so that the manipulated data including the original ADU and the padding bytes are a multiple of the lowest common multiple of all length values of the used ILP data manipulations. The second problem is solved by adding the result values, i.e. the values depending on the data manipulations, to the end of packet. These values are also (un)marshalled using (un)marshalling routines generated by a stub compiler such as ISC to support heterogeneous environments. The selected solutions result in a special data structure of an ADU appropriate for ILP processing. The structure is shown in Figure 5.

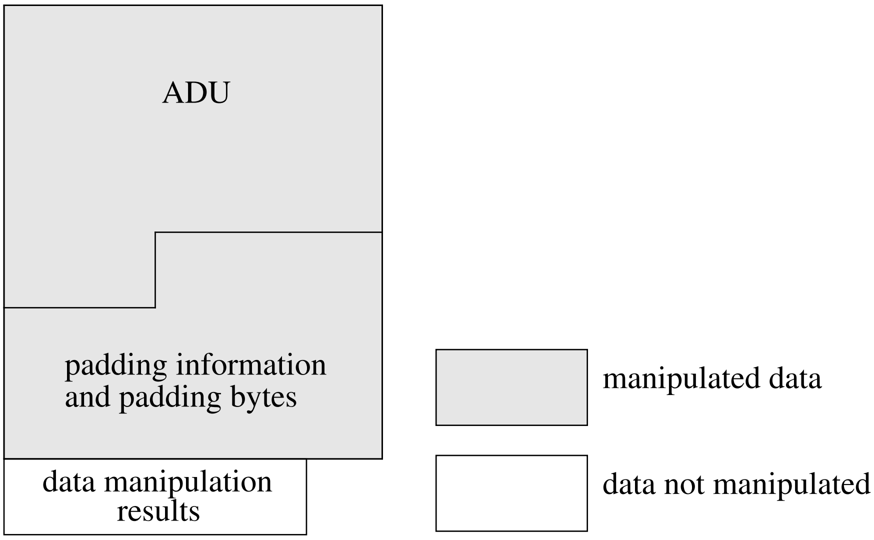

Figure 5 ADU structure for ILP

5.2 Tool extensions and Application Programming Interface

To support the advanced ALF protocol architecture, the ILP configuration tool generates some additional files for ILP processing of an ADU. First, a file "*ILP_RES.h.anno*" is generated, which serves as an input file of a second run of the ISC/ILP stub compiler, in which (un)marshalling routines for the ILP results without additional ILP data manipulations are generated. Furthermore, the ILP configuration tool generates functions for sending and receiving ADUs.

The functions for sending performs the following steps:
- marshalling and ILP data manipulation as defined for the ADU
- adding padding bytes to the marshalled and manipulated ADU
- adding the ILP data manipulation results to the end of the marshalled and manipulated message using a marshalling function without ILP data manipulations

The function for receiving performs:
- unmarshalling and data manipulations of the ADU and the padding information
- unmarshalling of the data manipulation results

- comparison of data unmarshalling results and unmarshalled results calculated at the receiver

At the last step, it is checked whether the received data was completely unmarshalled that must be the case in an ALF architecture. Furthermore, the lowest data manipulation function is detected, where the results for sending and receiving data differ in order to detect errors. The configuration tool generates a file *IF_ilp.c*, which contains the C code of the functions for sending and receiving ADUs.

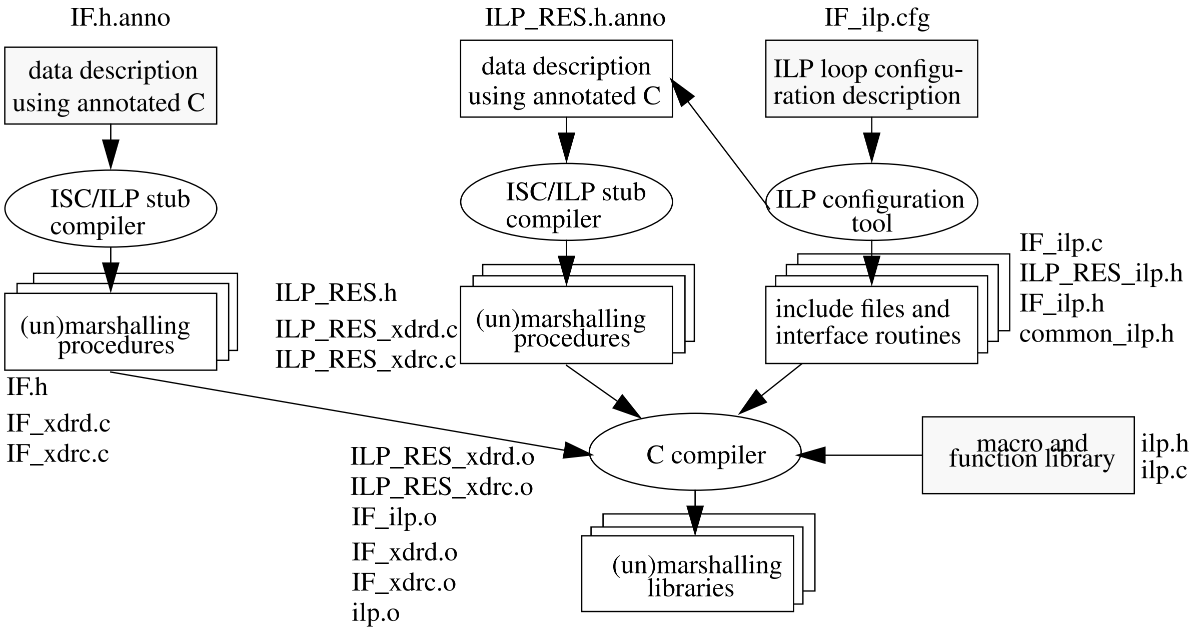

Figure 6 ILP configuration tool extensions for ILP processing of ADUs

The interfaces of these functions for sending and receiving data look as follows:

```
int *ilp_send_DATATYPE(ilpresult, buffer, DATATYPE_ptr)
ILP_RES *ilpresult;
int *buffer;
DATATYPE *DATATYPE_ptr;
```

The first parameter (`ilpresult`) is a pointer to the generated type for storing the ILP data manipulation results. This variable may be initialized or evaluated by the application program. The second parameter (`buffer`) is a pointer to the output buffer, where the marshalled and manipulated data must be written. The last parameter (`DATATYPE_ptr`) is a pointer to the data type describing an ADU. The function returns a pointer to the end of the buffer, to which the data has been written during marshalling and data manipulations. The function for receiving data has the following interface:

```
int ilp_receive_DATATYPE(ilpresult_r, ilpresult_s, buffer, bufferend,
DATATYPE_ptr)
ILP_RES *ilpresult_r;
ILP_RES *ilpresult_s;
int *buffer;
int *bufferend;
DATATYPE *DATATYPEptr;
```

The first two parameters contain the data manipulation results. The first one (ilpresult_r) contains the results calculated by the receiver during unmarshalling and data manipulation. The second one (ilpresult_s) contains the result calculated at the sender. If one or more data manipulation results differ an error value indicating the lowest incorrect data manipulation is returned. This value describes the lowest level where the results differ. The next two parameters describe the beginning (buffer) and the end (bufferend) of the received data, and the last parameter is the pointer of the unmarshalled variable (DATATYPE_ptr). The function returns an error value as described above.

6 EVALUATION

To estimate the achievable benefits of ILP, a file transfer application with an encryption function on top of a user-level TCP has been implemented and the performance of the application has been measured. The results (Braun, 1995) show that it is possible to obtain performance benefits by integrating marshalling, encryption and TCP checksum calculation. The experiments yielded in a throughput gain of only 10-20% in contrast to the 50% gain achieved for simple loop experiments (Clark, 1990). This makes the benefit of ILP debatable in existing communication systems and workstations.

The ILP code for the experiment described in (Braun, 1995) has been written manually. The code generated by ISC/ILP is very similar to the code generated manually. Experiments with automated code generation did yield noticeable performance differences.

This fact is not surprising, because the ILP compiler inlines hand-written macros from the macro library into the ILP loop. The number of read/write accesses and CPU operations is nearly the same in both implementations. Therefore, the same performance improvements by ILP can be achieved using ISC/ILP for automated ILP code generation instead of manual coding the ILP loops. The evaluation in (Braun, 1995) showed that ILP benefits are mainly reduced access to (cache or main) memory.

7 CONCLUSIONS

As discussed in (Braun, 1995), the ILP performance improvements may be very small. The smaller the expected ILP benefit the smaller are the efforts a programmer is willing to spend for ILP. A tool for automated ILP loop generation decreases the efforts for ILP implementations. Therefore, ILP implementations are rather acceptable considering the cost/benefit tradeoff if there are tools for automated coding helping to minimize implementation costs.

The fact that the code generated automatically does not perform worse than manually generated code increases the attractiveness of automatic code generation tools for communication software development.

The advantages of such a tool are obvious. The time to develop efficient protocol code can be kept small. Automatically generated code is much less faulty. This reduces the efforts to test and debug the code. Replacing data manipulations by other ones only requires to run the ILP code generation tool and to re-compile the resulting (un)marshalling procedures.

8 ACKNOWLEDGEMENTS

Torsten Braun performed this work at INRIA Sophia-Antipolis during a fellowship sponsored by the Commission of the European Communities under the Human Capital and Mobility Program HCM (no. ERBCHBGCT930254). We want to thank Philipp Hoschka and Isabelle Chrisment for valuable discussions about design and implementation issues.

9 REFERENCES

Abbott, M. B. and Peterson, L.L (1993) Increasing Network Throughput by Integrating Protocol Layers, *IEEE/ACM Transactions on Networking*, Vol. 1, No. 5, pp. 600-610

Braun, T. and Diot, C. (1995) Protocol Implementation Using Integrated Protocol Processing, *ACM SIGCOMM* , pp. 151-161

Braun, T; Chrisment, I.; Diot, D.; Gagnon, F. and Gautier, L. (1996) The HIPPARCH approach to ALF/ILP based Automated Implementation of Distributed Applications, HPDC 5 Symposium, Syracuse, August 6-9 1996.

Castelluccia, C; Dabbous, W. and O'Malley, S. (1996) Generating Efficient Protocol Code from an Abstract Specification, *ACM SIGCOMM*

Clark, D.D. and Tennenhouse, D.L. (1990) Architectural Considerations for a New Generation of Protocols, *ACM SIGCOMM*, pp. 200-208

Massey, J.: SAFER K-64 (1993) A Byte-Oriented Block-Ciphering Algorithm, *LNCS* 809, Springer-Verlag, 1993, pp. 1-17

Hoschka, P. and Mazziotta, S. (1994) Introducing ISC: the INRIA stub compiler, *Internal Working Report*, INRIA-Rodeo