AUTOMATED DERIVATION OF ILP IMPLEMENTATIONS FROM SDL SPECIFICATIONS

Sven Twarok[1], Peter Langendoerfer[2], Hartmut Koenig
Brandenburg University of Technology at Cottbus, Department of Computer Science, PF 101344, 03013 Cottbus, Germany, Tel./Fax: +49-355-69-2236 / 2127, email: twarok@unilab.de, langendoerfer@ihp-microelectronics.com, koenig@informatik.tu-cottbus.de

Abstract In this paper we present a mapping strategy for an important manual protocol implementation technique: the Integrated Layer Processing (ILP) which is used for protocol stack implementations with complex data manipulation operations. The Integrated Layer Processing was implemented beside the server and the activity thread model as third code generation procedure in the configurable SDL compiler COCOS. We describe the chosen transformation principle and discuss in particular the semantic problems which have to be solved for this mapping. Finally we present first measurements which show the effect of the transformation. Performance gains up to 20 per cent compared to COCOS server implementations and of about 300 per cent compared to the SDT *Cadvanced* tool were measured.

Keywords: Formal description techniques, integrated layer processing, automated protocol implementation, configurable FDT compiler, SDL

1. MOTIVATION

The increasing range of specialized communication solutions, in particular in mobile communication [Lang01], will increase the importance of the deployment of automatically generated code in real-life protocol implementations. Automated code generation can make the implementation process rather efficient and can fast adapt to changes in the design. The

[1] Sven Twarok is now with UniLab Software AG Braunschweig
[2] Peter Langendoerfer is now with IHP Microelectronics Frankfurt (Oder)

acceptance of automated implementation techniques depends on the efficiency of the generated code. Automated implementation techniques will be only then applied if the efficiency of the generated code comes close to that of manual implementations. The research of the last 10 years in this area has shown that there is still a large potential for efficiency increase by introducing optimizing techniques in the code generation process as discussed in [Lang99]. A remarkable efficiency gain can be achieved by adapting implementation techniques which are used for manual protocol implementations [Koen00]. In particular two approaches have been pursued:

- the integration of advanced implementation techniques and
- a flexible adaption to the implementation context.

The solution proposed for this is that of a configurable FDT compiler. The SDL compiler COCOS (*Configurable Compiler for SDL*), developed in our group, is an example for this approach. It possesses a variable runtime system which can be configured according to the requirements of the implementation context and the applied implementation technique. COCOS supports different implementation techniques as the server model, activity threads, the integrated layer processing, and techniques for avoiding data copying within a protocol stack. The basic principles of COCOS and the applied mapping strategies were presented in [Lang99]. In this paper we describe the generation of implementations according to the Integrated Layer Processing technique which has not been reported so far.

The Integrated Layer Processing (ILP) is an efficient implementation technique for protocol stacks with complex data manipulation operations [Clar90]. It in particular aims at protocol implementations with high data throughput requirements such as multimedia transmissions. The Integrated Layer Processing is especially applied to implement applications which follow the Application Level Framing principle. The Application Level Framing (ALF) approach supports a layer-independent processing of the data units. It aims among others at an application oriented definition of the data units, so called application data units, e.g. video frames, and the avoidance of protocol functions like fragmentation and multiplexing. The idea of the Integrated Layer Processing is to reduce memory accesses which may become a performance bottleneck in high performance communication when handling complex data units. This is achieved by a combined execution of complex data manipulation operations (encryption, decryption, checksum calculation, syntax transformations) in one or two memory access cycles independently of the layer the protocols belong to.

In this paper we show how ILP implementations can be automatically derived from SDL specifications. In Section 2 we give a short introduction to the Integrated Layer Processing implementation technique. Section 3 gives a short overview of related work. Section 4 describes the applied mapping

principle for SDL. Section 5 discusses semantic constraints. Section 6 sketches the code generation process. In Section 7 we present measurements to show the effect of the approach. The concluding remarks give an outlook on further required research steps.

2. PRINCIPLE OF INTEGRATED LAYER PROCESSING

The Integrated Layer Processing (ILP) represents an advanced protocol implementation technique. It was proposed by Clark and Tennenhouse [Clar90]. It is dedicated to the efficient implementation of data manipulation operations in protocols with a high data throughput. Typical examples for such data manipulation operations are the calculation of the checksum for error control, data encryption/decryption, data marshalling/unmarshalling and the compression/decompression of data streams. The approach is based on the observation that the execution of data manipulation operations over a protocol data unit (PDU) consumes more time for reading and writing of the data from and into the memory than for the proper execution of the operation. Since these data manipulation operations due to their functionality belong to different protocol layers many memory accesses are required in a traditional protocol stack implementation. Memory access times decrease at a slower rate compared to increase of the processor execution speed. For that reason memory access has become a bottleneck in high performance communication.

The Integrated Layer Processing aims at reducing the number of memory accesses by a combined execution of all data manipulation operations (DMOs) of a protocol stack. The DMOs are grouped into the so called ILP loop (see Figure 1). The DMOs can belong to different layers. They can even be located in different address spaces. The execution of the ILP loop does not influence the protocol execution, because the ILP loop combines operations which otherwise would be executed sequentially when the PDUs pass the different protocol layers. For execution, the DMOs are nested. They pass the data from one operation to the other whereby only fragments of the PDUs are processed each time. The DMOs aim at possibly small data fragments which fit into the register set of the processor. Thus only one read and one write operation per PDU are required for executing the ILP loop. The DMOs often use a different number of bytes as basis for their calculation. The calculation of the checksum is processed byte-wise, while an RSA encryption needs 16 byte blocks. To adjust the PDU fragments in an ILP loop word filters were proposed in [Abbo93] which concatenates bytes according to the length needed for the next DMO.

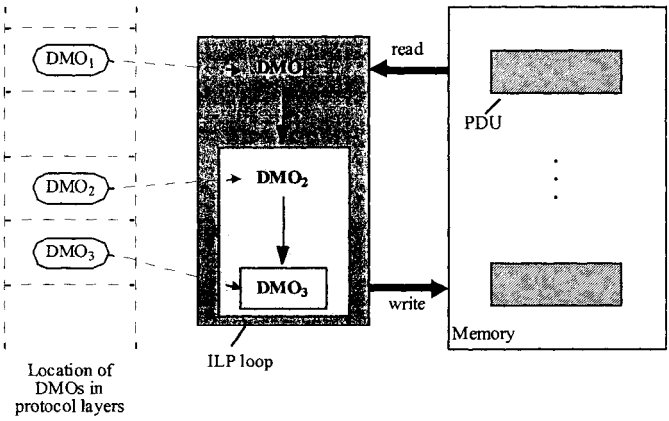

Figure 1: Principle of the Integrated Layer Processing

The ILP approach aims at the execution of the correct protocol paths, also called common paths [Leue96]. If an protocol error occurs in one of the protocols another execution path must be pursued to handle this error. In this case the results of the already executed DMOs of the higher layers are not needed any more and will be discarded. Low protocol error rates do not much influence the performance gain of the approach. If the error rate increases other implementation techniques will be more appropriate. We also discuss this effect later in this paper.

The Integrated Layer Processing can bring remarkable performance gains. For simple data manipulation operations like the calculation of the checksum, performance gains of up to 50 per cent were observed [Clar90], [Abbo93]. Further experiments revealed that the performance increase depends on the complexity of the data manipulation operations. It is less for more complex operations [Brau95]. In [Ahlg96] it was shown that ILP can even in part lead to efficiency losses compared to traditional implementation techniques if all data and state information of the data manipulation operation to execute cannot be stored in the register set of the processor. These data are moved to the cache or the memory. If the data are held in the cache the decrease may not be that strong but if the required information is neither found in the registers nor in the cache the performance loss may be dramatic. The deployment of the technique therefore requires a careful preparation to use its benefits.

3. RELATED WORK

The use of ILP has been investigated for formal description techniques (FDTs). For SDL, an approach for deriving the common paths was proposed in [Leue96], but no mechanisms for error recovery were discussed. The treatment of semantic constraints arising, for instance, from transitions with save statements also has not been solved. To our knowledge a compiler using ILP for one of the standardized FDTs has not been implemented yet. Automated derivations of ILP implementations from formal description techniques have been only reported for Esterel so far [Brau96], but the paper scarcely gives hints on the handling of semantic constraints. It is reported that the generated code achieves the efficiency of hand-coded implementations. Esterel though is more implementation oriented compared to the standardized FDTs. The problems for applying the technique to SDL arise from semantic constraints like the asynchronous communication between the processes. A practicable solution for SDL was proposed in [Twar00]. It is presented in the following.

4. MAPPING ON ILP IMPLEMENTATIONS

The mapping of SDL specifications on ILP implementations is not straight-forward. There are several problems to solve. First it is necessary to have an SDL specification of several protocol layers or of the whole protocol stack, respectively. We assume in the following that this specification is given. Furthermore, data manipulation operations are usually not entirely specified using SDL statements. In addition, SDL exchanges only complete signals between processes. The specification therefore does not provide sufficient support for a byte-wise exchange of the signals between DMOs. The compiler cannot always nest the DMOs. In order to use nevertheless the benefits of ILP implementations we distinguish two kinds of ILP loops:

- the simple ILP loop and
- the advanced ILP loop.

The loops differ in the nesting degree of the DMOs, but both variants lead to optimized implementations.

Simple ILP loop

In *simple ILP loops* the DMOs are not nested. Instead of this they are mapped into a common procedure and executed sequentially. Figure 2 shows the principle. The transitions which are involved in the loop remain preserved in the implementation. The PDUs of the different layers can be

uniquely identified. Protocol sequences which cannot be mapped on this loop are implemented in the usual way. No additional measures are applied.

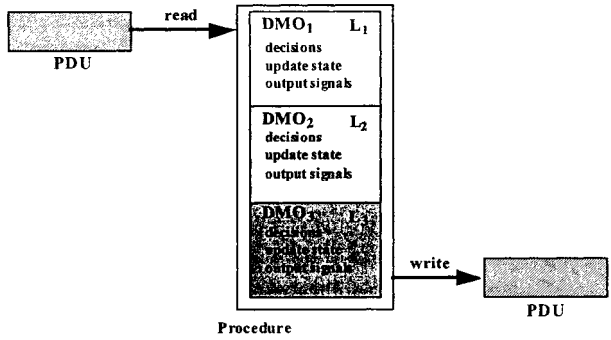

Figure 2: Structure of the simple ILP loop

The performance gain of simple ILP loops results from the avoidance of data copy operations between the DMOs. Besides no runtime support is needed for the process change. This is implicitly done by the ILP loop. The overhead needed for the signal exchange and the execution of the respective operation is avoided.

Advanced ILP loop

The advanced ILP loop fully implements the ILP concept as proposed in [Clar90]. The DMOs are nested and enforce a byte-wise processing of the data units. Because of this the PDUs of the different layers cannot be identified any more. All decisions concerning the correct transmission of the PDUs have to be delayed until the processing of the data unit is completed (see Figure 3). After that the decisions are evaluated. If the PDUs were correctly transmitted the respective protocol state is updated and the `output` statements are executed. For this, the transitions belonging to the loop must be restructured. This can be done automatically based on the following rules:

1. Move all decisions which influence the protocol flow behind the DMO execution,
2. Postpone outputs until these decisions are made,
3. Suppress all outputs if a protocol error occurs and trigger the error handling procedure,
4. Include word filters if the DMOs use different data formats (see below).

The performance gain of the *advanced ILP loop* is of course higher than that of the *simple loop* because only registers are used for the data exchange between DMOs.

Specification of ILP loops and word filters

Another important problem is the specification of the DMOs which belong to a certain ILP loop. The SDL specification does not contain any information on this. This information has to be included into the implementation specification which is the starting point for automated code generation. The implementation specification refines the protocol specification for the given target systems and includes all information for automatically deriving an implementation. The reasons for the use of an implementation specification are given in [Lang99], [Koen00].

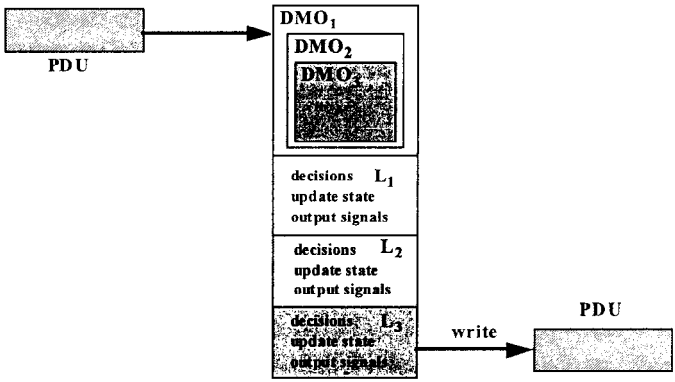

Figure 3: Structure of the advanced ILP loop

In our approach implementation oriented information is included by means of iSDL statements [Lang99]. iSDL is an implementation oriented annotation to control the code generation process. The statements are included as comments into the SDL specification and translated together with the specification. iSDL provides statements for specifying ILP loops and word filters.

An ILP loop is specified by means of the statements *define simple/advanced ILP loop,* e.g.

```
/*{$ define simple ILP loop L13 }*/
      .  .  .
   input sig1    /*${ belongs to L13}*/
      .  .  .
   input sig3    /*${ belongs to L13}*/
```

The statement indicates the transitions containing the DMOs in the order of their appearance. Note that the *belongs to* statement is attached as comment to the respective transition in the SDL specification.

A word filter for *advanced ILP loops* is defined as follows:

```
/*{$ define word filter 2_to_4 }*/.
```

The use of the filter is specified at places where it is required, e.g.

```
/*{$ P1 2_to_4 P2 }*/.
```

5. SEMANTIC CONSTRAINTS

There are two possibilities to implement the ILP loop execution: a separate and an integrated implementation. In the first variant the ILP loop is executed separately; the results are stored until they are needed for protocol execution. This is a straight-forward implementation of the concept. It, however, requires an additional memory access. From the point of view of automated code generation this variant is less advantageous, because it requires separating the DMOs and the actualization of state information. Therefore, the integrated implementation is preferred, in which the ILP loop is executed together with the protocol. This, however, requires that the protocols or the respective SDL processes are in states in which they accept the respective PDUs. There may be situations in which this is not given. The ILP loop cannot be executed then. Another problem is the detection of transmission errors. They cannot be detected before executing the loop. The handling of these situations is discussed in the following.

State errors

The execution of a PDU or a signal, respectively, may not be possible due to the current states of the SDL processes involved in the ILP loop. If a signal is executed then all processes must be in a state in which they await the PDUs contained in this signal. If these processes accept only signals exchanged within the loop and/or if they only possess one state, as supposed in [Leue96], the ILP loop can be always executed. The DMOs are processed consecutively. This is not allowed in situations when, for instance, a signal is saved by a `save` statement or when a signal is read from the input queue and discarded. Figure 4 depicts such a situation. The process *encrypt* may only encrypt video frames when it got a sending credit. Otherwise it waits in the state *wait _for_credit.* Incoming frames are saved until new credit is

available. If the ILP loop would force the encryption of incoming frames then the frames will be lost because the receiver would not accept them without credit. Besides the state would change in DMO *encryption* although it should not change at all. This can influence the protocol execution. Similar problems appear when signals are discarded in a state. In such cases the implementation has to switch to the conventional protocol execution (see below).

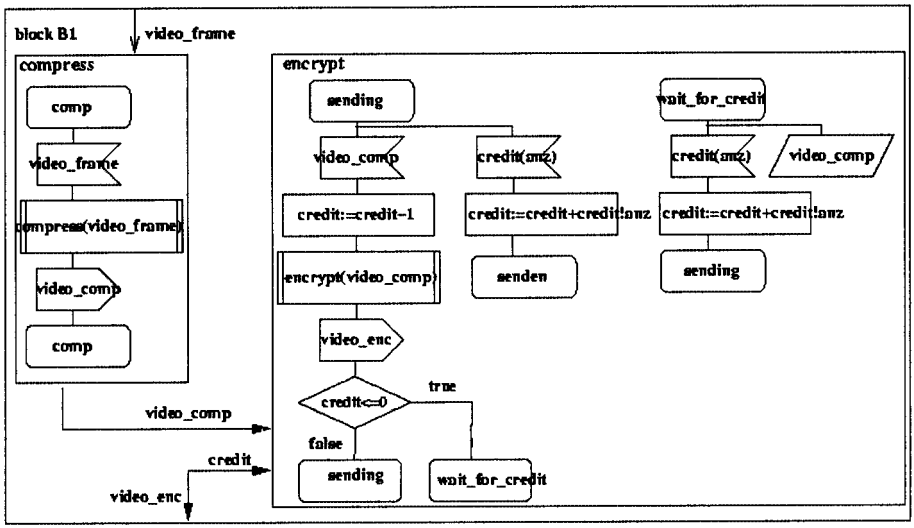

Figure 4: Refinement of the DMOs *compress* and *encrypt*

Transmission errors

Transmission errors can cause execution paths other than the common path the ILP loop aims at. In a *simple ILP loop* the error is indicated after proving the checksum, because the DMOs are processed PDU-wise. The execution can immediately branch off to another execution path to handle the error. In an *advanced ILP loop* with a byte-wise processing of the signals, errors are only detected after all DMOs are completed. At that point the signals are already converted into a representation which cannot used any more for error correction, e.g. for forward error control.

Solutions

In order to avoid these situations appropriate measures have to be taken. Mostly a double implementation is used. Another possibility is a recovery mechanism.

Double implementations mean that beside the ILP implementation a conventional implementation, either a server model or a activity thread implementation, is generated. Before the processing of the PDU the states of

all protocols are checked. If a protocol is not in the appropriate state then the implementation switches to the conventional execution path. A special treatment is required for handling transmission errors when applying the *advanced ILP loop.* To ensure the correct protocol behaviour the PDU and all state information (variables, timers, states, sequence numbers) have to be stored before the ILP loop execution. When a transmission error is indicated the PDU and the original protocol state have to be restored before the conventional execution starts. Thus the signal is processed a second time to determine the kind of the error and to react appropriately.

Recovery mechanisms are based on log files and reset points. The actualization requires rather large efforts. Since transmission errors in fibre optics network with error rates of 10^{-15} are very rare the efforts for recovery mechanisms are not justified. In particular for automated code generation double implementations should be preferred, because all required information already exists and can be used by the code generator. Recovery mechanisms have to be supplemented manually in the implementation.

Overtaking of signals

An overtaking of signals in SDL can only take place if two or more signals are sent over the same `signalroute-channel` connection. ILP loops do not cause such a behaviour. All signals are sequentially executed within an ILP loop so that the order of the signals is preserved. However, double implementations, i.e. the combination of ILP with server model or activity thread implementations, can lead to signal overtaking. This can be avoided by applying mechanisms like the transition reordering described in [Lang99].

6. CODE GENERATION PROCESS

The generation of ILP implementations in COCOS is principally the same as for the other implementation techniques. It comprises two steps. First the implementation specification, i.e. the SDL specification together with the iSDL annotation, is transformed into an *intermediated representation* which consists of connected lists of the processes, of the states and of the transitions. The intermediate representation is used for various checks and optimizations [Lang99]. It is also the input for the code generation process which is carried out by the *code synthesis* component (see Figure 5). The *synthesizer* inspects the intermediate format to identify the iSDL statements which control the code generation by indicating the mapping strategy and the respective parameters. The selected implementation model identifies the respective code segments for inclusion

into the generated code sequences. The *code segments* contain the implementations of the SDL statements in the target language, currently in C. They may be different for certain statements, e.g. the input statement, due to the different implementation techniques. The code segments can be parametrized. The actual parameters are included during code synthesis, e.g. the signal types for each `output` statement. The *synthesizer* further accesses the *code repository*. It contains user implemented functions such as word filters for the *advanced ILP loop* as well as segments for inlining manual coded sequences. The *code synthesis component* also determines parameters for the initialization functions of the runtime environment, e.g. the maximum length of the common input for all SDL processes. If an ILP implementation is supposed to be generated then the ILP loop is integrated either in a server model or an activity thread implementation as shown in Figure 5.

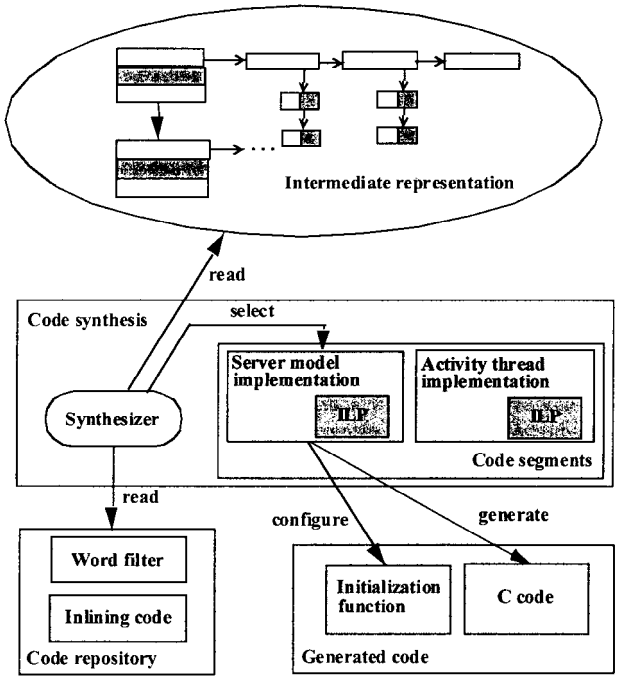

Figure 5: Code generation in COCOS

7. MEASUREMENTS

In order to evaluate the performance of the transformation we used the protocol stack depicted in Figure 6. The protocol stack is based on the XDT (*eXample Data Transfer*) protocol [Koen97]. It is an example protocol used for teaching protocol engineering. XDT is a connection-oriented data transfer protocol which applies the *go back N* error correction principle. For ILP, some data manipulation operations were added. They are located in the application layer as shown in Figure 6. Two DMOs are applied, one for calculating the checksum and one for coding/decoding the PDUs. The coding DMO applied for this experiment is very simple. It only byte-wise changes the payload data of the PDUs. The SDL/PR specification of the whole protocol stack has 991 lines. This corresponds 21 Kbyte.

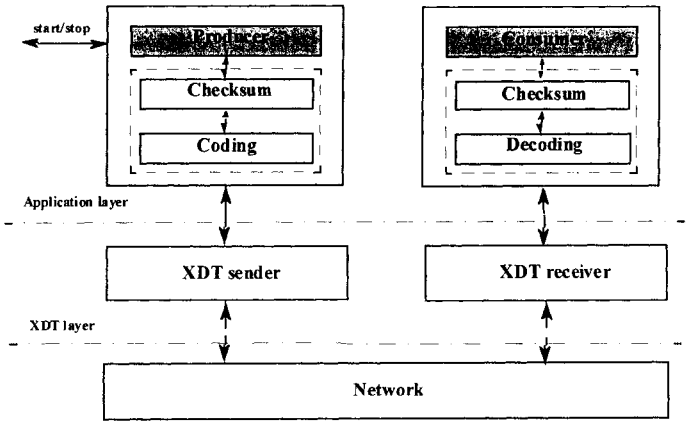

Figure 6: Structure of the used protocol stack

We generated a *simple* and an *advanced ILP loop* implementation and compared it with a pure server model implementation. The ILP loops comprise all transitions which belong to the checksum and coding/decoding DMOs. The loops are generated at sender and at the receiver side. The rest of the implementation is implemented straight-forward according to the server model [Lang99]. The measurement distance is the time between the sending of a PDU and the reception of the acknowledgment at sender side. It is marked by the signals *start* and *stop* in Figure 6. In each measurement cycle 10000 PDUs with a payload of 25 bytes were transferred. The measurements were repeated 50 times. Figure 7 shows the results the measurements for 3 workstations with varying performance parameters. Workstation A is a HYPER SPARC station with one processor (140 MHz) and 512 Kbyte cache, workstation B is a SPARC 20 with four processors

(125 MHz) and a cache of 256 Kbyte, whereas workstation C represents a ULTRA SPARC station with two processors and a cache of 2 Mbyte.

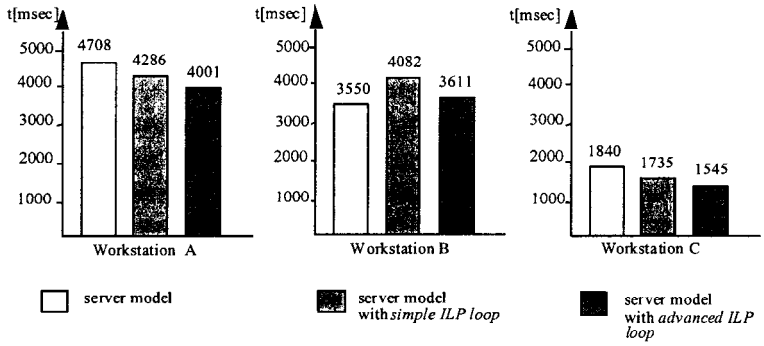

Figure 7: Measurement results

A comparison of the measurements at workstations A and C show that *simple loop* implementations bring a performance increase of about 5 to 9 per cent, while the *advanced ILP loop* implementations still add 7 to 11 to a total gain of almost 20 per cent. This shows that ILP may significantly improve the performance of protocol stack implementations. We further have to take the fact into account that our configuration is pretty simple. The two DMOs are adjacent and located in the same layer. Thus the cache reduces memory accesses. Several DMOs in different layers would more strongly demonstrate the effect of the approach. The phenomenon of workstation B that the server model implementation was about 1.8 per cent faster can be explained by the small cache of that workstation. The variation of the PDU execution time (measured in a separate experiment) was considerably higher at workstation B than for the other ones. This explains the fact that needed data and state information are not any more available in the cache (see also Section 2). When using larger PDUs of 200 byte payload this effect disappears (at least for low transmission error rates), because there are more memory accesses in the server model implementation (see below).

Another important aspect in this context is the influence of the error rate. Transmission errors force ILP loop results to be discarded and to start an error recovery procedure or continue with the conventional implementation. In order to determine the influence of transmission errors on the efficiency of ILP implementations we investigated the efficiency loss depending on the error rate. For this, we changed the order of the DMOs at receiver side, i.e. the checksum was calculated before the PDUs were decoded. Due to the combined execution of the operations all data are also decoded in error case. Unlike a non-ILP implementation the conventional implementation must be

re-established before error handling. We compared the modified *advanced ILP loop* implementation with a server model and an activity thread implementation using various error rates. This was done by falsifying a corresponding number of signals during transmission. The measurements were carried out at workstation B with a signal payload of 200 bytes and 10000 transmissions. The results are depicted in Figure 8. The measurements show that from an error rate of 6.5 or 10 per cent, respectively, upwards the activity thread and the server model implementation are more efficient. With a payload of 100 bytes this change only appears at an error rate of 13.5 or 19 per cent, respectively. The error rate is therefore not a conclusive measure for the performance of ILP implementations. The latter depends on the point where the execution stops when an error occurs and the number of operations to be repeated.

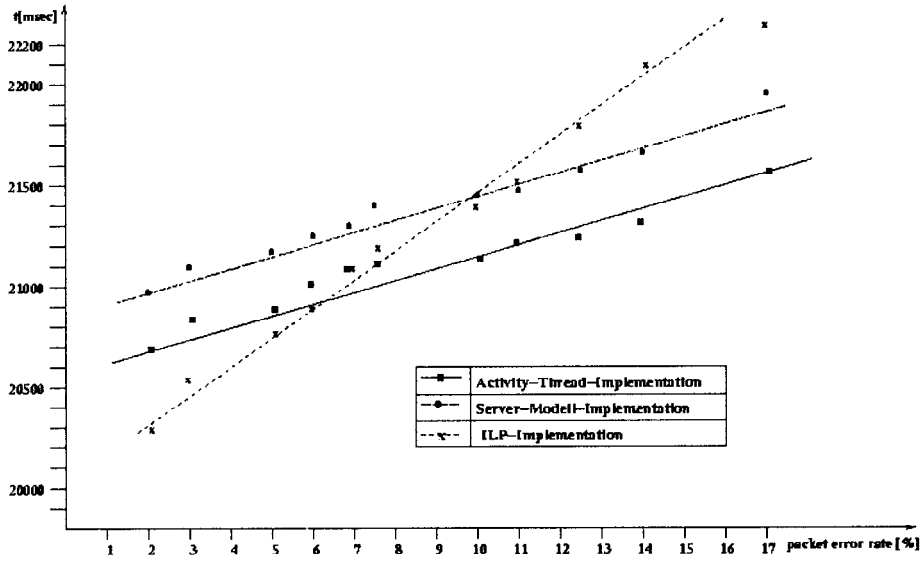

Figure 8: Influence of transmission errors on the efficiency of ILP implementations

The same experiment was also carried out with the *Cadvanced* code generator of the SDT tool version 3.2 [Tele98]. The execution times for error rates of 0 and 10 per cent were 65196 msec and 65342 msec, respectively, i.e. a performance gain of about 300 per cent could be achieved by using ILP compared to conventional, commercially available code generation tools.

8. FINAL REMARKS

In this paper we have presented a mapping approach for the automated derivation of ILP implementations from SDL specifications. We have discussed the semantic constraints and the code generation process. The measurements confirmed the performance increase reported from manual implementations. In addition we compared the *advanced ILP loop* implementation described in Section 6 with a manually coded server model implementation of the XDT protocol stack under the same conditions (200 byte payload, 10000 measurements). It showed that the manual implementation is only by a factor of 2.5 faster. Although this is only a conditional comparison, it points out that the integration of advanced manual implementation techniques like Integrated Layer Processing, activity threads or the offset technique [Lang99] may considerably improve the efficiency of automated code generation from formal descriptions.

Further research is needed to refine the techniques and to reveal places which slow down the execution speed of the generated code. One of such points is the coding/decoding of PDUs. Automated implementations of these procedures are often less efficient than manually coded ones. An inlining of manual codings in the automated code generation process might be appropriate support in this case. Moreover, it pointed out that the decision which implementation technique is selected is not straight-forward. Several experiments proved that the possible performance increase depends on the structure of the specification and on the target hardware. The better the specification supports the used implementation technique the better is the performance gain to be expected. The experiments further revealed that the different implementation techniques are not orthogonal to each other. The combination of the techniques does not bring a linear performance increase, since some optimizations are implicitly contained in other techniques. Besides the effect of certain techniques can be also achieved by combining of techniques, e.g. activity threads in combination with the offset technique can achieve similar performance parameters as ILP implementations. The selection of the appropriate techniques therefore requires a careful analysis of the specification and the implementation context. Performance predictions based on the implementation specification are a valuable means to support this process. Such a component is currently being integrated into the COCOS compiler.

An issue for a broad application of automated implementation techniques as well as other FDT based methods is the lack of complex formal specifications of real-life protocols. If larger specifications are available they are often not fully specified, especially PDU coding and decoding operations. The completion of the specification is still a time consuming process which

takes several man-months. This mostly cannot be done in an academic environment. It hinders a realistic comparison of automatically generated protocol implementations with manually coded ones in a large scale. The availability of such specifications is also decisive prerequisite for a broad and thorough application of FDT based design, validation and implementation technologies.

REFERENCES

[Abbo93]Abbott, M.; Peterson, L.: Increasing Network Throughput by Integrated Protocol Layers. IEEE/ACM Transaction on Networking 1(1993)5

[Ahlg96] Ahlgren, B.; Gunningberg, P.G.; Moldeklev, K.: Increasing Communication Performance with a Minimal-Copy Data Path Supporting ILP and ALF. Journal of High Speed Network 5(1996) 2, 203-214

[Brau95] Braun, T.; Diot, C.: Protocol Implementation Using Integrated Layer Processing. Proc. ACM SIGCOMM, 1995.

[Brau96] Braun, T. et al.: ALFred - An ALF/ILP Protocol Compiler for Distributed Application Automated Design. Rapp. de Recherche No. 2786, INRIA, 1996.

[Clar90] Clark, D. D.; Tennenhouse, D. L.: Architectural Considerations for a New Generation of Protocols. Proc. ACM SIGCOMM '90, pp. 200 - 208.

[Koen97] Koenig, H.: eXample Data Transfer Protocol (XDT). Technical Report I-18/1997, BTU Cottbus, Department of Computer Science, 1997

[Koen00] Koenig, H.; Langendoerfer, P.; Krumm, H.: Improving the Efficiency of Automated Protocol Implementations Using a Configurable FDT Compiler. Computer Communications 23 (2000) 12, 1179-1195

[Lang97] Langendoerfer, P.; König, H.: Increasing the Efficiency of automatically generated code using implementation specific annotations. Proc. 3rd Workshop High Performance Protocol Architectures (HIPPARCH), P. Gunningberg, C. Huitema (eds.), Uppsalla, 1997.

[Lang99] Langendoerfer, P.; Koenig, H.: Deriving Activity Thread Implementations from Formal Descriptions Using Transition Reordering. In Wu, J.; Chanson, ST.; Gao, Q. (eds.): Formal Methods for Protocol Engineering and Distributed Systems. Kluwer Academic Publishers, 1999, pp. 169 - 184.

[Lang01] Langendoerfer, P.; Koenig, H.; Kraemer, R.: Evaluation of well-known Implementation Techniques for Application in Mobile Networks. Accepted for Special Issue of the Journal of Supercomputing, Kluwer Academic Publishers

[Leue96] Leue, S:, Oechslin, P.: On Parallelizing and Optimizing the Implementation of Communication Protocols. IEEE/ACM Trans. on Networking 4 (1996) 1

[Tele98] Telelogic Malmoe AB: SDT 3.4 User's Guide. SDT 3.4 Reference Manual, 1998

[Twar00] Twarok, S.: Extension of the COCOS code generators for deriving ILP implementations. Master thesis, BTU Cottbus, Department of Computer Scvience, 2000 (in German)