The TACO Protocol Processor Simulation Environment

Seppo Virtanen
Turku Centre for Computer Science (TUCS)
Lemminkaisenkatu 14 A, 20520 Turku, Finland
seppo.virtanen@utu.fi

Johan Lilius
Department of Computer Science and
Turku Centre for Computer Science (TUCS)
Abo Akademi University
Lemminkaisenkatu 14 A, 20520 Turku, Finland
johan.lilius@abo.fi

ABSTRACT

Network hardware design is becoming increasingly challenging because more and more demands are put on network bandwidth and throughput requirements, and on the speed with which new devices can be put on the market. Using current standard techniques (general purpose microprocessors, ASIC's) these goals are difficult to reach simultaneously. One solution to this problem that has recently attracted interest is the design of programmable processors with network-optimized hardware, that is, *network* or *protocol processors*. In this paper a simulation framework for a family of TTA protocol processor architectures is proposed. The protocol processors consist of a number of buses with functional units that encapsulate protocol specific operations. The TACO protocol processor simulator is a C++ framework based on SystemC. Functional units are created as C++ classes, which makes it easy to experiment with different configurations of the processor to see its performance.

Keywords

microprocessor, protocol, simulation, codesign

1. INTRODUCTION

Network hardware design is becoming increasingly challenging because more and more demands are put on network bandwidth and throughput requirements, and on the speed with which new devices can be put on the market. Using current standard techniques (general purpose microprocessors, ASIC's) these goals are difficult to reach simultaneously. General purpose microprocessors are no longer an appealing alternative for networking hardware on account of their lack of optimized execution units for network processing. All the networking functionality must be implemented in software, which in turn leads to such high CPU clock frequency requirements that general purpose processors in the speed range are very expensive, or are simply not available. Also, many general purpose processor features, like FPU's, can usually not be taken advantage of in networking devices. For these reasons among others, ASIC's have been widely used for networking devices. ASIC's can provide more processing speed with lower clock frequency than general purpose processors. However, ASIC design is difficult and expensive and they tend to have a long time-to-market. Also, ASIC's are usually not programmable and thus need to be redesigned for updated or new network protocols, making them inflexible in dynamic market segments.

One solution to this problem that has recently attracted interest is the design of programmable processors with network-optimized hardware, that is, *network* or *protocol processors*. Such a processor is an attempt to harness the processing speed of ASIC's and the programmability of general purpose processors for optimal protocol processing speed. A recent article [4] presents four commercial protocol processors. Some of these designs are already available on the market or are made available in sample quantities.

The challenge in protocol processor design is finding an architecture that is a good compromise between a general purpose processor and a custom, protocol-specific processor (ASIC): ideally the architecture should be optimized for a family of protocols. Looking at the available documentation of the commercial processors, we can see that most of them try to leverage the knowledge about parallel processing by essentially providing an interconnection fabric together with a number of general purpose computing elements. However this approach raises a number of questions for the designer:

1. How can the designer evaluate the performance of the architecture for a given protocol? I.e. given a protocol, which of the existing architectures should the designer choose?

2. How should one program for the given architecture to obtain optimal performance? I.e. given a protocol and given an architecture, what kind of software (protocol implementation) architecture gives optimal performance?

At the moment the designer has to resort to experience and ad-hoc experiments when evaluating the architectures. Thus it seems to us that more research is needed to understand the exact needs of protocol processing in terms of both software and hardware architecture.

In our research project TACO (Tools for Application-specific HW/SW Codesign) we are focused on developing a framework for the design of programmable protocol processors. We have taken a different approach compared to the commercial protocol processor suppliers. Instead of designing an ad-hoc parallel architecture, in the TACO development work flow the protocol processor is developed guided by the development of the protocol software. One of the key tasks in the TACO work flow is the identification of frequently occurring protocol processing operations that vary little between different protocols and that can be implemented in hardware to increase execution speed and reduce code size. Thus the design process should be such that it makes it easy to identify these operations, while the processor architecture should be such that it allows easy integration of the operations. We believe that object-oriented techniques [2] provide a good basis for the software part of the design-flow, and that the Transport Triggered Architecture (TTA) [1, 11] is a good candidate for the hardware architecture. Objects encapsulate state and functionality, while classes describe the common properties of a number of objects. Thus objects and classes are the right abstractions for modelling protocol processing operations. Many object-oriented methods contain techniques and notations for *domain analysis*, that is identification of common data-structures and functionality within a domain of interest. Thus the object-oriented techniques help us find candidates for the protocol specific operations. These operations are then integrated as functional units (FU's) on a TTA processor. A TTA processor is formed of FU's that communicate via an interconnection network of data buses, controlled by an interconnection network controller unit. TTA offers several advantages, in that it is extensible, allowing it to be customized to different protocols, and it is optimized for data transfer, which is the main function of a telecommunication protocol.

In this paper we will concentrate on describing the architecture of the TACO protocol processor and its simulator. We will also briefly sketch our development process and discuss some problems we encountered when implementing the simulator in SystemC [9].

2. TACO PROCESSOR ARCHITECTURE

The main function of a telecom protocol is to reliably transfer data from the sender to the receiver. Interleaved within this data transfer task are different signaling tasks like control flow, connection setup or teardown etc. However, in a well designed protocol these signaling activities should occur seldom enough not to incur any extra penalty on the performance of the data transfer. In selecting an architecture for the TACO processors an important criteria was therefore the ability to have efficient data transfers.

The TACO processor architecture is a slightly modified transport triggered architecture (TTA) [1, 11]. In TTA processors data transports are programmed and they trigger operations - traditionally operations are programmed and they trigger transports. A TTA processor is formed of functional units (FU's) that communicate via an interconnection network of data buses, controlled by an interconnection network controller unit, as seen in figure 1.

The connection between a functional unit and the interconnection network is managed by input and output socket units as shown in figure 2. Each functional unit has one or more operand registers, trigger registers and result registers. An operation is triggered when data is transported to a trigger register.

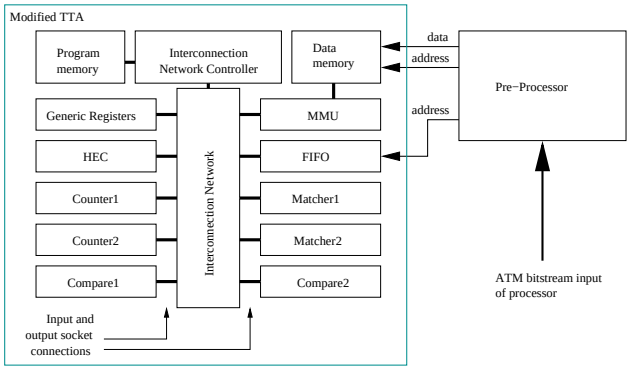

Figure 1: A TACO protocol processor for ATM [13].

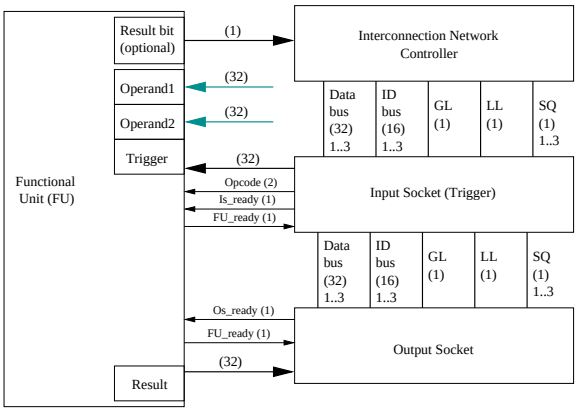

Figure 2: Connectivity between FU's, sockets and the interconnection network. GL=global lock, LL=local lock request, SQ=squash.

TTA's are in essence one instruction processors, the only instruction being move data. Thus, the instruction word of a TTA processor consists mostly of source and destination addresses of sockets called socket ID's. The socket ID's are transported on ID buses from the interconnection network controller. There are as many ID buses as there are data buses in the interconnection network. Upon finding its socket ID on one of the ID buses, a socket opens the connection between an FU and the corresponding bus on the interconnection network. The maximum number of instructions (i.e. data transports) that can be carried out in one clock cycle is equivalent to the number of data buses in the interconnection network.

The benefit of TTA is its modularity and scalability. Functional units can be added to the architecture or they can be refined and changed as long as they provide the same interface to the sockets connecting them to the interconnection network. Our architecture is therefore more of a template for protocol processors, instantiated for a specific protocol

by selecting an appropriate number of buses and the appropriate functional units. In previous work [12] we have analyzed a number of communications protocols and identified some typical protocol processing elements that are common to the protocols: bitstring matching, integer comparison, checksum calculation (especially CRC) and indexing (counters). Wireless and timing-critical protocols also need capabilities for maintaining timers and generating random values. All of these protocol processing tasks are distinct enough to be considered for implementation as FU's.

Some protocols also benefit from protocol data unit (PDU) pre-processing (the pre-processor in figure 1). The tasks performed in pre-processing are protocol dependent and may include synchronization to the incoming bitstream, data integrity verification (by means of performing a protocol dependent error check on incoming data) and incoming PDU storage into the processor s data memory (using DMA). The memory addresses of first data words of PDU headers can be stored into a FIFO to provide quick access to the data that requires processing. The pre-processor unit in our architecture is optional and protocol dependent.

TACO processors are programmed by specifying data moves between functional units with 20 instruction bits for each move (i.e. each bus). For example for a processor with three data buses, the instruction word length of 64 bits is required: it consists of three sets of 20 bus instruction bits followed by four bits for managing immediate integer generation.

The 20 instruction bits are divided into a source socket ID and a destination socket ID accompanied with four guard bits (a guard ID) used for conditional execution. The long instruction word (containing all the instruction bits for each bus and the four immediate generation control bits) is decoded by the interconnection network controller, which also distributes the socket ID's onto the ID buses.

As presented in figure 2, some FU's have a result bit connected directly to the interconnection network controller. With this structure it is possible to directly use the result from an FU in guard bit evaluation. This feature is especially useful in matchers, compare units and error check units. The interconnection network looks for a certain combination of the result bits specified by the guard ID for each bus. If the result bits from FU's do not satisfy the condition specified by the guard ID, the specified data move on the bus is squashed, i.e. ignored.

3. THE TACO DEVELOPMENT PROCESS

The TACO processor design process consists of 2 parts: identification of the functional units that are needed in the TACO processors, and the design of the control structures for the protocol at hand. The end result of this process is a list of functional units and their quantities that the protocol processor must support, the number of buses in the protocol processor, and the assembler code for the protocol implementation.

Although any design method could probably be adapted to our needs we have chosen to base ourselves on object-oriented (OO) methods. One of the main advantages that an object-oriented method gives us is that since it is focused

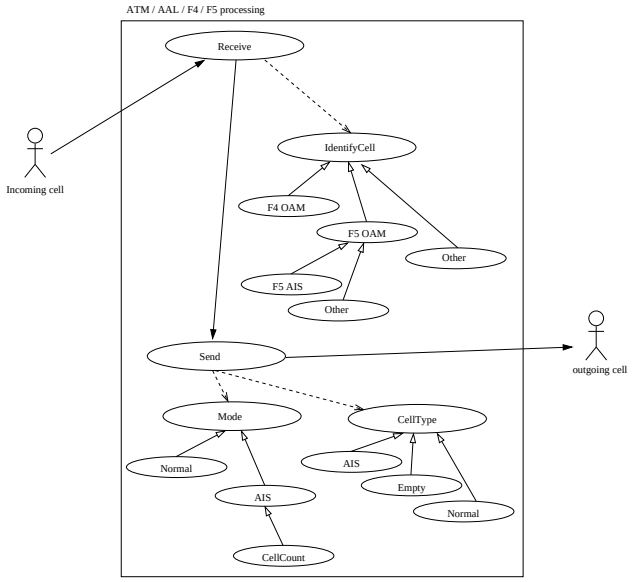

Figure 3: Use Case diagram for processing ATM F5 AIS cells from the network.

on identifying objects and implementing the functionality of the system using objects, the method naturally adopts to the identification of functional units.

As most modern OO methods, we start the description of our protocol by drawing up a use case diagram. The use case diagram shows the main functionality that needs to be implemented. In our context use cases are essentially data transports between service access points (SAPs) of different protocol layers. Such a use case could be for example sending a user protocol data unit (PDU), receiving one, or perhaps receiving an operation and maintenance (OAM) type PDU. As an example, figure 3 shows use cases for ATM F5 AIS cell processing [3] drawn using the UML [2] notation. The use case diagram can be used to guide the development process in an iterative way such that in each iteration the implementation of one use case is added to the prototype (c.f. the ROPES process of [2]).

From the use cases we can obtain our first object diagrams of the system. We then continue to refine the object diagrams until we identify operations that can naturally be implemented in hardware. This is the process of *domain analysis*. Using these functional units we can now build a simulator for our TACO processor, by instantiating suitable modules within the simulator framework (c.f. below). At this stage we have also obtained the instruction set of the processor.

The next step is to implement the control structures using the given functional units. This is done by building collaboration diagrams [2] showing the interactions between the different functional units. Each interaction corresponds to a move instruction and is indicated by an arrow. Figure 4 shows such a diagram.

The collaboration diagram in figure 4 shows a part of the interactions needed for processing ATM cells with the kind

of TACO processor shown in figure 1 (a processor with three data buses and dual functional units). The dotted arrows indicate information generated by the interconnection network controller (e.g. immediate integer generation and conditional execution control), and the solid arrows indicate transfers between functional units. Two arrows pointing to a single FU indicate that both of the input registers of the FU are loaded with data (see section 2 for more information concerning architectural details). Dashed horizontal lines indicate clock cycles.

The functional units in TACO processors are designed to produce their result in one clock cycle. This is possible by specifying the internal functionality of each FU so that it is concise and simple enough to be completed in one cycle. We have made this decision to keep the process simple at this stage. The TACO architecture allows FU's that need more than one cycle for completing their operations, and we plan to include such more complex FU's in the design process in the future.

From the collaboration diagram it is possible to draw estimates on the bus utilization and clock cycle requirement for a certain algorithm implemented on a certain processor architecture: e.g. the interactions shown in figure 4 (a part of ATM AIS cell processing) require 8 clock cycles and use 21 out of 24 possible data transfers.

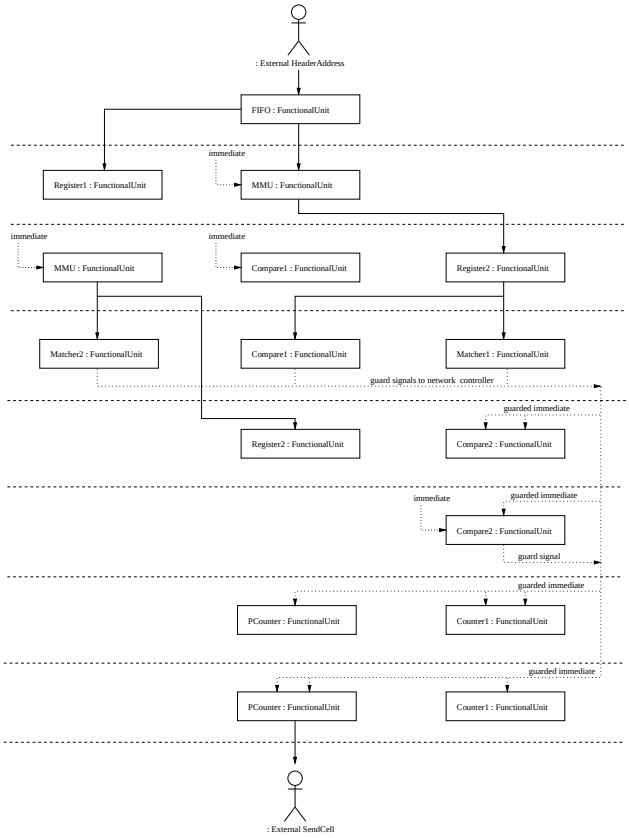

Figure 4: Operations per cycle in AIS processing for a TACO processor with dual FU's and three data buses. Arrows represent data transfers. Immediate value generation and control signals are managed by the interconnection network controller (not shown).

4. THE PROCESSOR SIMULATOR

At the moment the simulation environment consists of a library of components implemented in SystemC [5, 9] on an x86 Linux system using GNU compilation utilities. SystemC is a C++ [10] application framework for simulating hardware. It provides a set of classes for describing common entities in hardware design, e.g. signals, clocks etc. SystemC is distributed under an open license and is supported by several of the major EDA companies.

The TACO component library contains implementations of functional units, interconnection buses, and the program dispatch logic. Using this library it is possible to build a cycle-level accurate simulator of a TACO processor.

To test the fundamental assumptions of the TACO framework we have prototyped a processor for processing ATM AIS cells. This instance of the TACO architecture features three 32-bit buses in the interconnection network. This makes it possible to have three parallel data transports in one machine cycle.

The purpose of the TACO class library is to provide an environment for simulating and evaluating protocol processor designs. The goal is to be able to quickly simulate different instances of our protocol processor architecture and by means of these simulations and their results to find an optimal configuration of functional units and buses for a certain application to be performed on a protocol processor.

We have used inheritance and object oriented (OO) concepts extensively in the simulator. The class hierarchy of the simulator is given in figure 5. The classes that are used for simulating hardware are derived from the class `sc_module` provided by SystemC. This class provides among other things macros for simulating signals and ports. The `SocketManager` class is not a hardware simulation module: it is used by objects from the functional unit classes during simulator initialization for generating, connecting and maintaining sockets, socket ID's and signals dynamically. During simulations `SocketManager` is used for providing pointers to dynamically created subclasses of `sc_module`.

There are three different types of sockets needed in TACO processors (see section 2 for architectural details). InSockets are used for writing data into operand registers of FU's, OutSockets for reading data from result registers, and trigger sockets for writing data into trigger registers and simultaneously triggering FU operations. In the simulator all sockets are derived from the base class `Socket` that provides most of the socket interfacing and a state machine for each socket. The subclasses add their own internal functionality and interface requirements to the base class description.

The three level hierarchy for functional units was needed to overcome certain SystemC limitations discussed in more detail later in this paper. The base class `FunctionalUnit` provides most of the interfacing and a state machine for each FU. The TACO class library contains, but is not limited to the FU classes shown in figure 5. The functional

unit subclasses generally feature one operand register, one trigger register and one result register, but some, e.g. the `Matcher` class, require a second operand register. The internal functionality in each functional unit class is programmed to provide the result of a triggered FU operation by the beginning of the next simulation clock cycle. The functionality of the FU's is discussed in more detail in e.g. [12] and [13].

The interconnection network controller (class `NetControl`) does not have any subclasses, since there is always only one such module in a TACO processor. The controller manages breaking the long TACO instruction words into bus instructions (see section 2), immediate value generation and conditional execution management along with monitoring and issuing control signals. The interconnection network controller functionality is implemented as a state machine.

OO techniques ensure that similar objects have compatible external interfaces (e.g. FU's have compatible registers). They also make easy addition of new objects of the same kind into the system possible (e.g. two matchers). Functional units that are not needed for a certain protocol processing application can be left out of the simulation to speed up simulator compilation and the simulation itself.

As the algorithms needed for internal FU functionality are well known and hardware (gate level or schematic level) specifications with excellent performance characteristics exist for them, the emphasis in the simulator is to define the processor control structure, internal signaling and the number of FU's in a way that ensures maximal protocol processing throughput for a certain application.

The simulator is initialized by instantiating each bus in the interconnection network and then instantiating all the required functional units. The functional units are connected to the interconnection network by calling connect routines in bus objects. The creation of sockets and the signals required by them is done automatically and dynamically from within each functional unit as they are connected to buses as shown in the code excerpt below (line numbers have been added to the code in this excerpt for commenting purposes).

```
0:  NetControl* nc = new NetControl("NC");
1:  Bus* bus1 = new Bus("Bus1");
2:  Bus* bus2 = new Bus("Bus2");
3:  Matcher* m1 = new Matcher("M1",clk);
4:  bus1.insertOperand(m1);
5:  bus1.insertData(m1);
6:  bus2.insertData(m1);
```

Line 0: *Create the interconnection network controller "nc".*
Lines 1 and 2: *Create two new buses in the interconnection network and connect the buses to the network controller.*
Line 3: *Create a matcher functional unit "m1".*
Line 4: *Create an insocket for the operand register of m1, create signals for connecting the socket to m1, and connect the socket to m1 and the interconnection network controller.*
Line 5: *Create an insocket for the data register of m1, create signals for connecting this socket with m1, and connect the socket to m1 and the interconnection network controller.*
Line 6: *The data insocket already exists, so just connect this socket to the interconnection network.*

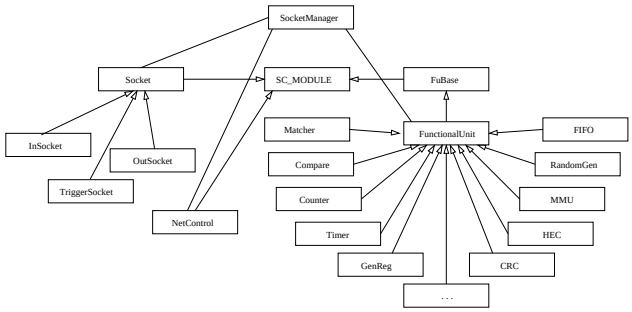

Figure 5: Class hierarchy of the TACO SystemC simulator framework.

As we wanted our simulator to be easily expandable, we early on decided to use inheritance as a structuring mechanism. The simulator (as seen in figure 5) has parent classes that encompass all the mutual features of the subclasses, and thus the subclasses are made reasonably simple by utilizing inheritance. Essentially the leaf classes only contain the implementation of the functionality, and all functionality dealing with the socket interface is implemented in the base class `FunctionalUnit`.

However, early on we noticed that this kind of implementation technique seems to be actively discouraged in SystemC version 1.0.1. A simple use of inheritance caused a core dump. After much debugging we were finally able to isolate the cause of the problem into the constructors of our functional unit classes. The constructor for an object of class `module` is declared as `SC_CTOR(module)` in SystemC. This is expanded to

```
typedef module SC_CURRENT_USER_MODULE;
        module(sc_module_name);
```

The first line defines a type name alias, that we will return to below. The second line is interesting. It declares a constructor for objects of class `module` that takes an anonymous `sc_module_name` object as a parameter. The life-time of the object is the scope of the constructor and it is used to push the current module onto the simulation context stack. This makes sure that all ports declared in the class are attached to the correct instance of class `module` in the simulator. So when building the inheritance hierarchy with `sc_module` as the base class, we must make sure that all the parent classes are instantiated within the scope of the correct `sc_module_name` object. This can be achieved by adhering to the following convention: Each non-leaf class must have a default constructor (a constructor with empty argument list). However a second problem still remains. This has to do with the first line with the `typedef`, which declares the type `SC_CURRENT_MODULE` to be an alias for the name of the current module. This type is needed in macros that are used to connect the execution of a method to the correct simulation context. However since non-leaf classes are created with the default constructor the constructor will not define `SC_CURRENT_MODULE` and thus the program will fail to compile. We have resorted to C++ templates to overcome this particular problem.

We thus implemented a multi-level class hierarchy, with the concrete functional unit implementation classes as leaf-classes and two intermediate levels before the SystemC level (the `sc_module` class). The top-level class (`FuBase` in figure 5) is an abstract class[1] that defines the interface to the functional unit classes. The second-level class `FunctionalUnit` is a template class that contains the common functionality of the functional units. It is also used to set `SC_CURRENT_MODULE` to the correct type. Conceptually the template class FunctionalUnit together with the class FuBase forms the interface that all Functional units support.

The following are the class declarations of the three-level class hierarchy (see figure 5):

```
class FuBase:  public sc_module
  {...};

template <class CB> class FunctionalUnit:
  public FuBase{
     typedef CB SC_CURRENT_USER_MODULE; ...};

class Matcher:  public FunctionalUnit<Matcher>
  {...};
```

5. CONCLUSIONS

In this paper we have proposed a C++ simulation framework for TTA protocol processor architectures based on SystemC. As mentioned in the introduction most of the commercial protocol processor architecture offerings are multiprocessors with general purpose processors as the computing elements. Our approach has more in common with application specific processors (ASIP) in that we try to provide hardware implementations of frequently occurring operations. However, in contrast to ASIP, our hardware operations encapsulate much more functionality than the 2-3 assembler instructions typically found in ASIP operations.

Two proposals from academia should be mentioned. In [7] a protocol processor architecture optimized for internet protocols is proposed. The emphasis in the optimization is on the handling of the state table. The processor contains a special unit to handle jumps efficiently. In contrast the TACO processor is more like a pipelined processor. The programmer has to schedule the jumps. In [6] another programmable protocol processor architecture is proposed. Here the idea is that each layer of the protocol is processed in a separate stage. The data flow is thus organized according to layers. In our approach something similar could be achieved by having one bus for each layer in the protocol stack.

Our next goal is to further improve our development method, sketched in section 3. This includes both making the design steps more precise and including estimates of physical parameters into the process. First steps for estimating physical parameters and maximum clock speed for the architecture from the system level have been discussed in [8].

6. ACKNOWLEDGMENTS

Financial support for this work from the Nokia foundation and the HPY research foundation is gratefully acknowledged by the first author.

[1] A class that cannot be instantiated.

7. REFERENCES

[1] H. Corporaal. *Microprocessor Architectures - from VLIW to TTA*. John Wiley and Sons Ltd., Chichester, West Sussex, England, 1998.

[2] B. P. Douglass. *Real-Time UML: Developing Efficient Objects for Embedded Systems*. Addison-Wesley, 2000.

[3] International Telecommunication Union, Telecommunication Standardization Sector. *ITU-T Recommendation I.610: B-ISDN Operation and Maintenance Principles and Functions*, 1993.

[4] M. Kohler. NP complete. *Embedded Systems Programming*, 13(12):45–60, November 2000.

[5] S. Y. Liao. Towards a new standard for system-level design. In *Proceedings of the Eighth International Workshop on Hardware/Software Codesign*, San Diego, CA, USA, May 2000.

[6] D. Liu, U. Nordqvist, and C. Svensson. Configuration based architecture for high speed and general purpose protocol processing. In *Proceedings of the IEEE Workshop on Signal Processing Systems (SiPS'99)*, Taiper, Taiwan, October 1999.

[7] Y. Ma, A. Jantsch, and H. Tenhunen. A programmable protocol processor architecture for high speed internet protocol processing. In *Proceedings of the 18th IEEE NORCHIP Conference*, pages 212–216, Turku, Finland, November 2000.

[8] T. Nurmi, S. Virtanen, J. Isoaho, and H. Tenhunen. Physical modeling and system level performance characterization of a protocol processor architecture. In *Proceedings of the 18th IEEE NORCHIP Conference*, pages 294–301, Turku, Finland, November 2000.

[9] Open SystemC Initiative. *http://www.systemc.org*.

[10] B. Stroustrup. *The C++ Programming Language*. Addison-Wesley Publishing Company, Reading, Massachusetts, USA, 3rd edition, 1997.

[11] D. Tabak and G. J. Lipovski. MOVE architecture in digital controllers. *IEEE Transactions on Computers*, 29(2):180–190, February 1980.

[12] S. Virtanen. On communications protocols and their characteristics relevant to designing protocol processing hardware. Technical Report 305, Turku Centre for Computer Science, Turku, Finland, September 1999.

[13] S. Virtanen, J. Lilius, and T. Westerlund. A processor architecture for the TACO protocol processor development framework. In *Proceedings of the 18th IEEE NORCHIP Conference*, pages 204–211, Turku, Finland, November 2000.