Flexible Design of Hardware-supported High-performance Protocol Processing Units

Georg Carle, Günter Schäfer, Jochen Schiller

High Performance Networking Group
Institute of Telematics, University of Karlsruhe, Germany
[carle, schaefer, schiller]@telematik.informatik.uni-karlsruhe.de

Abstract
Emerging applications mostly require both, high performance as well as support of a wide variety of communication services. For example, audio, video, and data transmission may require highly different services, e.g., guaranteed delay, jitter, or bandwidth. An additional challenge arises by the growing demand for multipoint communication services. ATM networks are capable of satisfying the basic application requirements by providing multipoint bearer services with data rates exceeding a gigabit per second. However, current communication subsystems (including higher layer protocols) that provide reliable services are not able to deliver the available network performance to the applications.
In order to provide the required high performance services to the applications, new protocols as well as high-performance implementation architectures for the communication subsystems need to be designed. Dedicated VLSI components should be used in flexible implementation platforms used for time-critical processing tasks in order to provide high performance.
This paper presents a new approach for the flexible design of hardware-supported high-performance communication subsystems. The design process allows mapping of a formal protocol specification onto a parallel, hardware-based implementation architecture. The highly modular VLSI implementation architecture designed with parametrizable and programmable components allows for service flexibility. The architecture is not limited to a certain protocol, but allows the implementation of a variety of high-speed protocols. We validated our approach with a design example using a formal specification of the protocol RMC-AAL (Reliable Multicast ATM Adaptation Layer, RMC-AAL [CaZi95])

Design Flow

One global goal of research is the automatic derivation of a high-performance communication subsystem from a formal specification [KrKS87, Kris92, Schi95]. Figure 1 shows the overall design process of our approach based on individual design steps that are customized for the design of hardware-supported high-performance ATM protocol processing units. The *specification* mentioned here consists of the *protocol* itself and a set of *configuration parameters*. The standardized language SDL (Specification and Description Language, [ITU88]) is used for specification. Therefore, we can use different tools for formal verification of the protocols to be implemented.
The configuration parameters describe the desired performance, the existing software environment, and the maximum costs of a system. Costs may be expressed in terms of processing costs, or hardware complexity. For determination of the required number of processing units, simulation results and measurements at results of previous design cycles can be used.
From these parameters, an *implementation framework* is composed from a set of predefined functional units. This framework consists of the interfaces to an environment, static memory for protocol data and a central crossbar to connect all components (c.f. Figure 2). We describe all hardware components shown in Figure 2 using the standardized hardware description language VHDL (VHSIC Hardware Description Language, [IEEE87]).

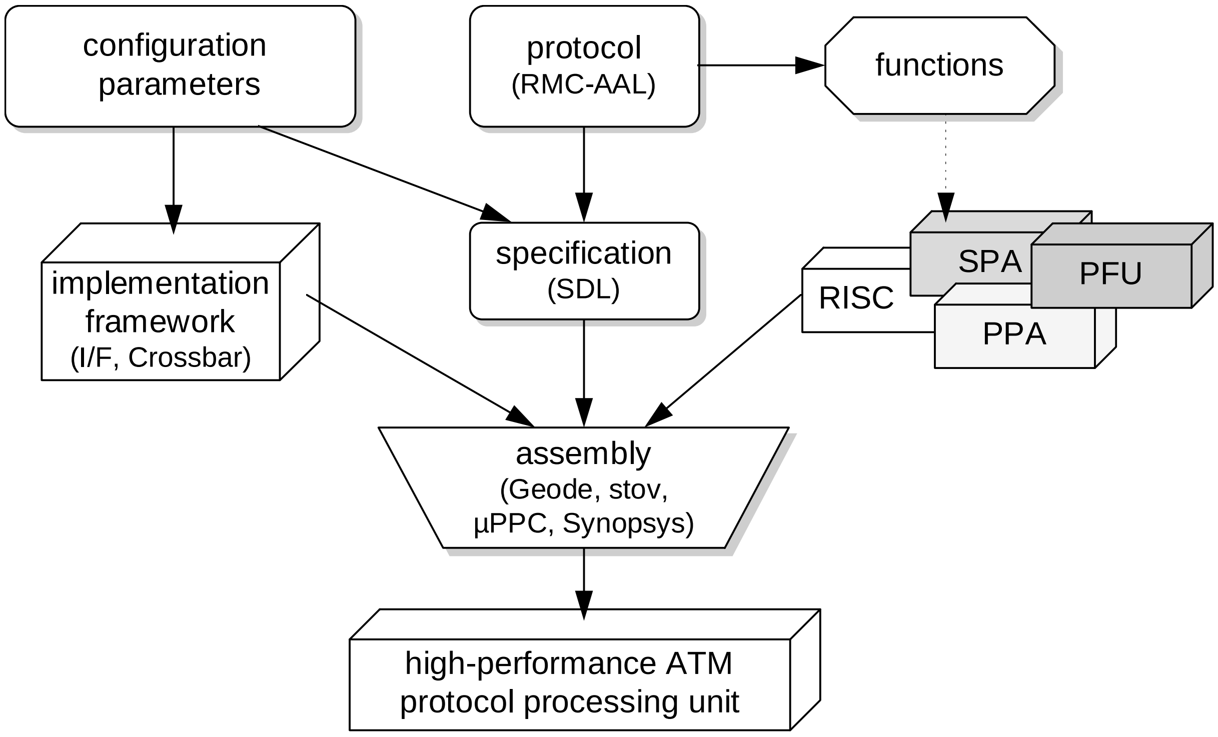
configuration parameters
protocol (RMC-AAL)
functions
implementation framework (I/F, Crossbar)
specification (SDL)
SPA
PFU
RISC
PPA
assembly (Geode, stov, µPPC, Synopsys)
high-performance ATM protocol processing unit

Figure 1: Customized design flow for high-performance ATM protocol processing units

From *protocols* like RMC-AAL we have extracted several *functions*, e.g., timer, CRC, FEC, transmit, or acknowledgement processing. These functions can be implemented on four different alternative architectures depending on the desired performance:

- *RISC-processors*: The tool Geode [Veri95] can be used to generate C-code from an SDL-specification. A RISC-processor can execute this code after compilation. Descriptions of different RISC-processors are available for example as VHDL-code or gate-level schematic.
- *Synthesisable Protocol Automata (SPA)*: With the help of a customized SDL-to-VHDL-compiler we can automatically map an SDL-description of a protocol automaton onto a VHDL-description of a hardware unit. After synthesis onto real hardware (e.g., with the tool Synopsys) this unit acts as a protocol automaton. Due to the dedicated hardware for protocol processing, the performance of such a unit may be significantly higher compared to a general purpose RISC-processor. However, existing hardware synthesis tools do not achieve optimal performance when synthesising netlists from high-level VHDL descriptions.

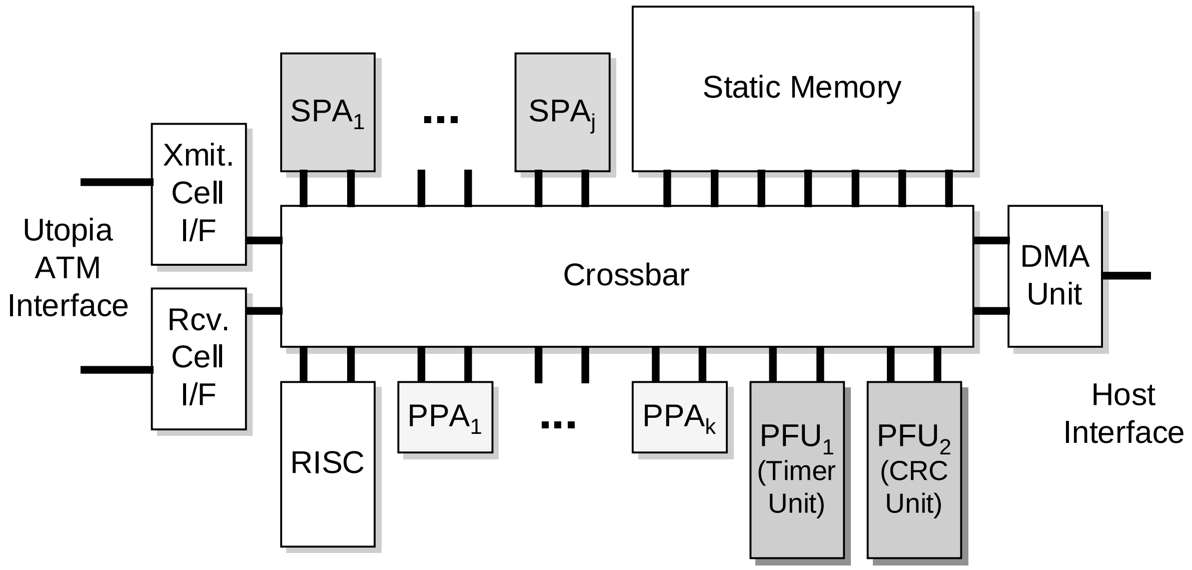

Figure 2: Flexible architecture for high-performance ATM protocol processing

- *Programmable Protocol Automata (PPA)*: For even higher performance we have designed microprogrammable automata. These automata consist of only 2895 standard cells in CMOS-technology and can run with 100 MHz. Up to now, we program these units directly with microcode using a custom microcode compiler µPPC (microprogram protocol compiler).
- *Protocol Function Units (PFU)*: To achieve highest performance, we implement time-critical protocol functions as hardware macros. Examples are timer, CRC, and FEC (Forward Error Correction) units. Gate-level VHDL is used for implementation of PFUs.

Depending on the specification, the different units are chosen and configured to assemble the *high-performance protocol processing unit*. Currently, we are using 0.7µm standard-cell technology for layout synthesis and, alternatively, FPGA-boards inserted into workstations for rapid prototyping.

Design Tools

Our design flow comprises several tools as shown in Figure 1. Up to now it is not possible to use a single tool for the whole design flow that is flexible enough for the different requirements and produces communication components with the required performance. The following gives a short overview of the tools used in our approach, and summarizes their advantages and disadvantages.

- *SDL-to-C compiler (GEODE code generator)*: GEODE [Veri95] is a commercial tool set for the design of event-driven real time systems, using the language SDL'88 [ITU88], and Message Sequence Charts (MSC, [ITU93]) for formal specification. The tool set provides support for graphical editing, simulation, debugging, and C code generation. Both the graphical form called SDL/GR and the textual phrase form called SDL/PR are supported. SDL specifications are logically composed of a hierarchy of structural objects. It can be selected how the GEODE code generator performs mapping of the SDL objects process, process instance, block, and system onto operating system processes. Specific functionalities which are specified as abstract data types can be mapped onto separately specified C functions. In our flexible design approach, we also perform mapping of abstract data types onto specific hardware functions implemented in PFUs.
- *SDL-TO-VHDL compiler (stov)*: In order to facilitate the process of hardware implementation of SDL specifications we developed a dedicated SDL-TO-VHDL compiler called *stov*. The compiler generates VHDL code that is specially suited to the flexible architecture shown in Figure 2. The generated code makes use of the existing VHDL libraries that describe the architecture. This allows for rapid prototyping of protocol processing units after successful simulation of the SDL specification. As there are some SDL constructs that cannot be translated into hardware descriptions, an appropriate subset of SDL is supported by the compiler.
- *VHDL compiler (Synopsys)*: Based on VHDL-descriptions of hardware on the register-transfer level, this commercial tool synthesises netlists for different technologies [Syno95]. These netlists can be used for further synthesis on ASICs or FPGAs. Compared to hand-coded netlists, this tool does not achieve the optimum speed and size of the hardware due to the complexity of the synthesis. On the other hand, using such a powerful synthesis system is the only way to manage the complexity of large hardware systems. In addition to synthesis, this tool also allows for simulation and debugging of VHDL-descriptions.
- *Microcode compiler (µPPC)*: Our custom microcode compiler allows for easy programming of the PPAs using a simple assembly level language. The language comprises 19 operations, comments, labels, and macros. Figure 3 gives a short microcode example for illustration. This microcode inserts the parameters of `i.gap` into a queue if the event `i.gap` arrives at the component. Afterwards, it gets the value of `arr_gap_no` from the memory, increments it by 1 and stores it back. The new state of the component is now `ARR_ACTIVE`.

```
@if event i.gap arrives
*/ARR_ACTIVE_i.gap
@insert i.gap into queue
/:PutQueue(ARRQueue, [inp,inp])
@store gap_no + 1
arr_gap_no mv mem a_r           xx   CNT  MOVE S,A
inc a_r                         xx   CNT  MOVE S,A
arr_gap_no mv a_r mem           xx   CNT  MOVE S,A
ARR_ACTIVE                      xx   CNT  SAVE
/*LabelARR                      xx   JMP  SLEEP
/*END
```

Figure 3: Microcode example

The compiler converts this microcode into a binary format which can be downloaded to the PPA. The disadvantage of this microcode is its low level language. Therefore, an additional SDL-to-Microcode-Compiler is under development.

Design Example: Timer PFU

To present not only high-level configuration of the architecture, we discuss one unit in more detail. The *timer PFU* manages a dynamic list of timers. Several commands exist to manipulate the list (c.f. Table 1). The parameters are defined as follows: *conn_id* indicates the unique identification of the connection the timer belongs to, and *time_out* is a 32 bit value indicating the time-out value. The appropriate receiver(s) of an alarm are listed in the *receiver* vector. The *timer_id*s are managed by the global accessible *timer component* itself. The resolution of this component implemented using 0.7µm CMOS technology is 100 ns, the maximum time-out value is, therefore more than 7 minutes. Larger time-out values can be handled completely by software in case they are not time-critical and do not require high precision. In other cases, the timer component can be used in combination with additional software functions.

command	input parameter	output parameter	comment
`create`	`conn id, time out, receiver`	`timer_id`	create a new timer entry in the timer list; return the timer identification
`set`	`conn id, timer id, time_out`		set an existing timer to a new time-out value
`reset`	`conn id, timer_id`		reset an existing timer to its original time-out value
`delete`	`conn id, timer_id`		delete an existing timer
`alarm`		`timer_id`	time-out has occurred

Table 1: Example commands of the timer PFU

Figure 4 shows the internal hardware architecture of the timer PFU. Key components are a queue, a timer control unit, ALU, and RAM. The queue decouples this component from other components connected to the central crossbar. Therefore, the execution speed of this component can differ from other components. The same mechanism is applied to all components inside our architecture.

All timer values are stored in the RAM. The ALU performs all necessary operations like data movement, comparison, and incrementing. The execution unit inside the timer control performs the signalling to other components via the crossbar and moves data inside the component. The execution of the commands listed in Table 1 is controlled by the control unit.

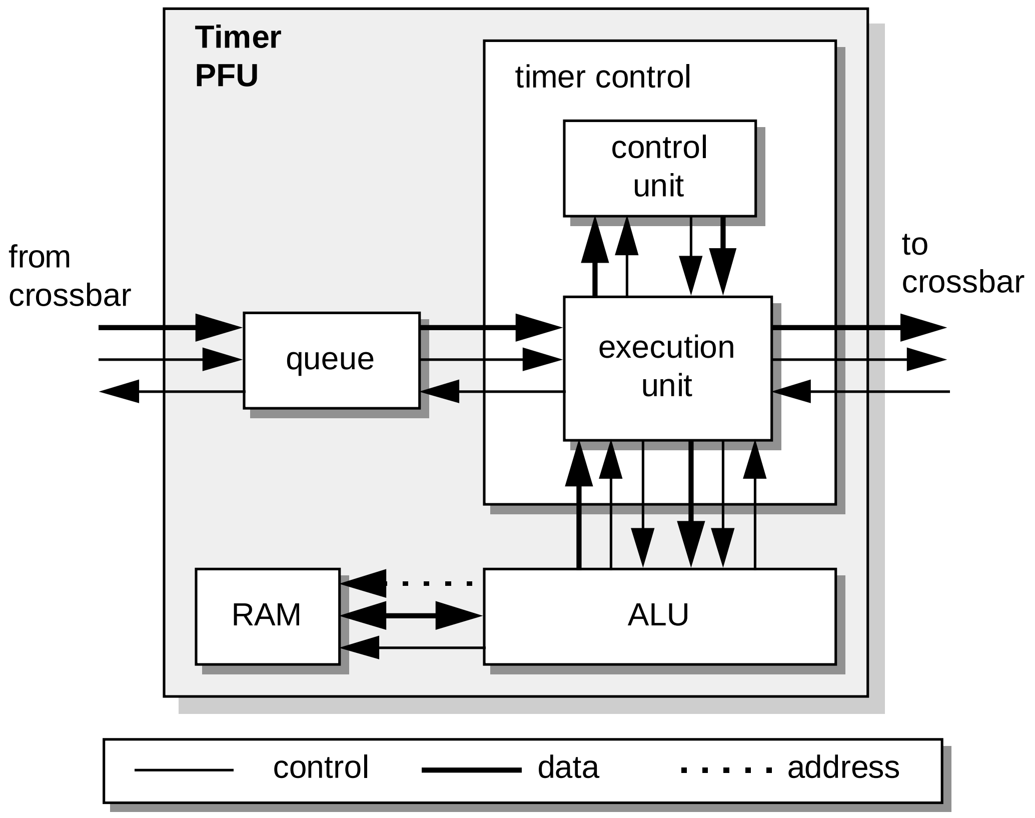

Figure 4: Internal hardware architecture of timer PFU

Figure 5 shows some performance results of the component implemented on programmable hardware (FPGA). For rapid prototyping we use a FPGA-board inside a workstation that is accessible via a C-programming interface. The disadvantage of this approach is the lower speed of FPGAs compared to ASICs and the need for accessing the FPGA-board via the system bus. After synthesis of the hardware, the FPGA can be configured and then used as a co-processor to speed-up execution time of time-critical functions. Therefore, we are able to compare pure software solutions with this first step of hardware support.

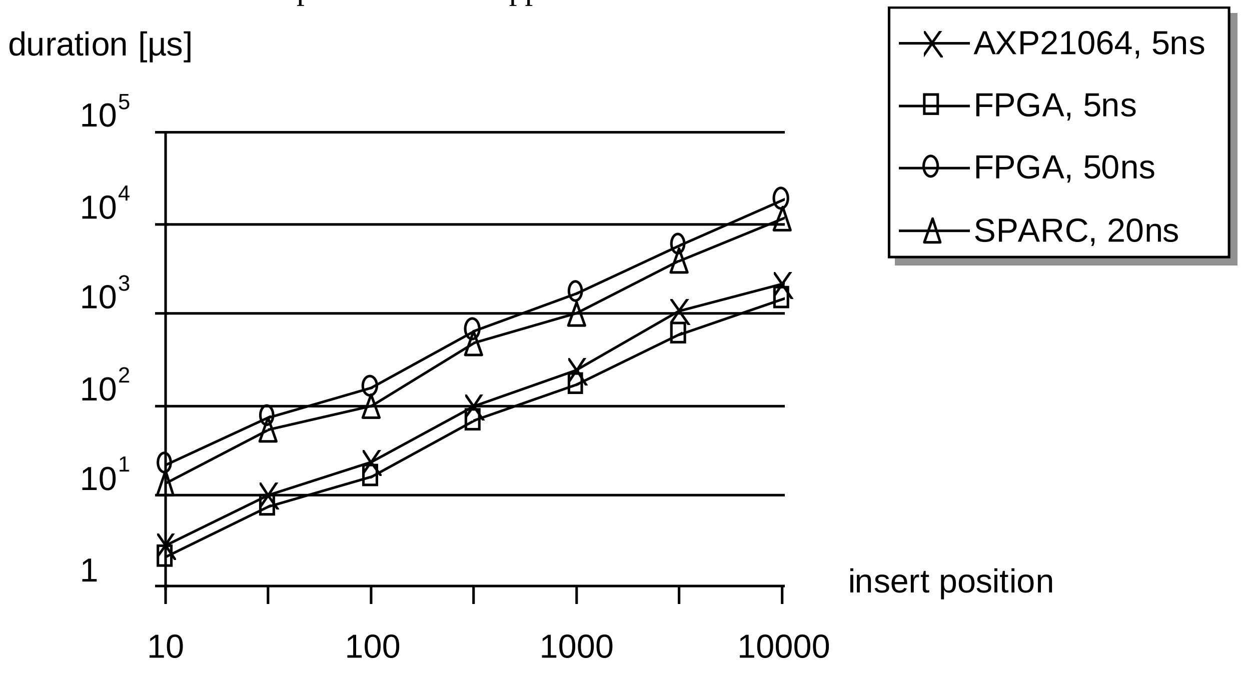

Figure 5: Performance comparison of timer implementations

Our current FPGA implementation runs with a clock-speed of 20 MHz (50 ns cycle time) on a Xilinx XC4013-4 FPGA. Figure 5 compares this performance with a software solution using the Digital Alpha Processor AXP21064 running with 200 MHz and a SPARC processor with 50 MHz. The fourth line shows the theoretical performance of our timer component using faster technology allowing the same clock-speed as the Alpha processor. The figure shows clearly that our solution is as powerful as workstation processors, although we are using only a small FPGA. Furthermore, the software solutions suffer from heavy system load, the operating system and, therefore, cannot guarantee a certain execution time.
Using the same hardware description of the timer PFU we also synthesised the component based on 0.7µm CMOS technology. This results in a critical path of 13 ns, allowing a clock-speed of more than 70 MHz. The chip area needed for the timer PFU is 1.8 mm².

Conclusion

The flexible design process presented in this paper allows for the efficient development of high-performance protocol processing units. Using a single, formal protocol specification allows to derive a number of alternative implementations on different architectures. This method is highly suitable for the development of protocol-specific solutions. Simulation and instrumentation of designs allow to gain valuable insight into protocol-specific and architecture-specific bottle-necks that may occur under specific scenarios. The alternative target architectures for the protocol functions allow to achieve highest performance, as well as a suitable compromise between costs and performance.

References

[CaSc95] Carle, G., Schiller, J.: *Enabling High-Bandwidth Applications by High-Performance Multicast Transfer Protocol Processing*, 6th IFIP Conference on Performance of Computer Networks, Istanbul, Turkey, October 23-26, 1995; in S. Fdida, R. Onvural (Eds.): "Data Communications and their Performance"; Chapman&Hall 1996, pp. 82-96

[CaZi95] Carle, G., Zitterbart, M.: *ATM Adaptation Layer and Group Communication Servers for High-Performance Multipoint Services*, 7th IEEE Workshop on Local and Metropolitan Area Networks, pp. 98-106, March 26-29, 1995, Duck Key, Marathon, Florida, U.S.A.

[IEEE87] IEEE: *IEEE Standard VHDL Language Reference Manual*, IEEE Std. 1076-1987

[ITU88] ITU-T Recommendation Z.100: *Functional Specification and Description Language (SDL)*, Telecommunication Standardization Sector of ITU, Geneva, 1988

[ITU93] ITU-T Recommendation Z.120: *Message Sequence Chart (MSC)*, Telecommunication Standardization Sector of ITU, Geneva, 1993

[Kris92] Krishnakumar, A.S.: *A Synthesis System for Communication Protocols*, Proceedings of the 5th Annual IEEE International ASIC Conference and Exhibit, Rochester, New York, September 1992

[KrKS87] Krishnakumar, A.S.; Krishnamurthy B.; Sabnani, K.: *Translation of Formal Protocol Specifications to VLSI Designs*, Protocol Specification, Testing and Verification, VII, Elsevier Science Publishers B.V., North-Holland, May 1987, pp. 375-390

[Schi95] Schiller, J.: *CHIMPSY - a Modular Processor-System for High-Performance Communication*, 1. GI/SI Jahrestagung, Zurich, September 18-20, 1995

[Syno95] Synopsys Inc.: *Documentation of Simulator, Design Compiler, and Design Analyzer,* Version 3.2a, Synopsys, Inc., Mountain View, California, USA, 1995

[Veri95] Verilog SA: *Technical documentation of the GEODE toolset*, Verilog SA, Toulouse, France