Reprint from „Proceedings of SBT/IEEE International Telecommunications Symposium;
August 22-26, 1994, Rio de Janeiro, Brazil“

IMPLEMENTATION OF TRANSPORT PROTOCOLS USING PARALLELISM AND VLSI COMPONENTS

Torsten Braun, Jochen Schiller, Martina Zitterbart
University of Karlsruhe, Institute of Telematics, Zirkel 2, 76128 Karlsruhe
Phone: +49 721 608-[3982,4021,4026], Fax: +49 721 388097
Email: [braun,schiller,zit]@telematik.informatik.uni-karlsruhe.de

Abstract

Service requirements and performance needs are increasingly demanding in emerging high performance communication subsystems. Well-suited protocols and efficient protocol implementation techniques form the core of such systems. Applying parallelism and introducing VLSI-components to overcome system bottlenecks represent promising approaches towards highly efficient implementations. This paper discusses experiments with parallel protocol implementations and introduces a protocol especially designed to support parallelism. Moreover, a parallel VLSI architecture is introduced enabling fine-grained parallelism among protocol functions as well as coarse-grain parallelism among connections. Dedicated VLSI components are used for potential bottleneck functions, such as timer and memory management or retranmission support.

1 Introduction

During the last few years the communication world has seen remarkable changes. Applications are becoming increasingly complex and require higher performance. A wider variety of communication services steadily increases the demands on communication subsystems. The transmission technology has evolved from data rates of several megabits per second to data rates exceeding a gigabit per second. However, current communication subsystems cannot deliver the available network performance to the applications.

Several research efforts on implementing high performance communication subsystems have been undertaken during the last few years covering aspects, such as software optimization, applying parallelism, using hardware support and dedicated VLSI components. A comprehensive overview on parallelism in communication subsystems can be found in [1]. Some of the approaches deal with efficient implementations of standard protocols, such as OSI TP4 or TCP (e.g., [2], [3]). Others developed protocols specially suited for parallel implementations, such as XTP [4], TP++ [5], MSP [6], [7], AXON [8] or PATROCLOS [9]. Moreover, [10] and [11] specifically deal with the VLSI implementation of simple protocols. The parallel VLSI architecture presented in this paper is especially targeted towards more complex communication protocols, such as connection-oriented protocols. Moreover, the architecture is highly independent of the specific protocol to be implemented and, thus, serves as a sound basis for the emerging environment of communication protocols. In addition, the presented architecture forms a key part of a framework that eventually should cover efficient and flexible automated protocol implementations from protocol specifications.

This paper is structured as follows. Section 2 reports on experiments with parallel protocol implementations and summarizes design requirements for protocols suitable for parallel implementations. In section 3, a parallel VLSI implementation architecture is presented. A summary and an outlook on future work is given in section 4.

2 Parallel Protocol Implementation and Design

2.1 Implementation of Standard Protocols

The use of parallelism at the point of bottleneck generally is a suitable approach to improve the performance of computing systems [12]. Considering processing speeds and memory bandwidth as major bottlenecks for high performance communication subsystems, the use of multiprocessor platforms forms an adequate approach to increase their performance.

Implementations of standard protocols like OSI and TCP/IP on parallel architectures lead to performance gains [1], [3]. However, mainly pipelined parallelism can be achieved [13] because of the subdivision of OSI systems into hierarchical layers and the high data dependencies among the protocol functions. Most of the approaches provide a rather coarse-grained level of parallelism using protocol entities or protocol stacks as basic parallel building blocks [14].

Due to the highly different complexity of protocol layers, the mapping on multiprocessors often leads to implementations which are not very efficient because of the unbalanced processor load (e.g., TCP and IP, OSI-TP4 and OSI-CLNP) [15]. Protocol functions as atomic units for the parallelization are more promising. However, to extract parallelism at the functional level, detailed protocol analysis is required [16]. Generally, standard protocols which are based on single extended finite state machines (FSMs) are not designed to support parallelism.

2.1.1 Parallel Protocol Implementation

SNR and XTP have been the first protocols based on parallel FSMs. SNR [17] is an end-to-end protocol for high speed networks with a relatively simple functionality. It is decomposed into seven processes, which have been implemented on a multiprocessor system based on M68030 processors [17]. In order to decouple control and user data processing, connection state parameters are exchanged periodically by control packets independent of user data transfer.

A more sophisticated protocol supporting parallelism is XTP [4]. Parallelism has been considered during the design process and has resulted in the specification of multiple FSMs [18]. Moreover, a fixed header format is used to simplify header generation and analysis. Furthermore, XTP clearly separates between send and receive parts which can be implemented concurrently without significant interactions and almost independently of each other. Thus, handling of duplex traffic can be supported efficiently [19]. Moreover, the distinction of information packets, which mainly contain user data, and control packets, which contain control information (e.g., for flow control or acknowledgment) simplifies parallel processing of user and control data.

Such features are extremely helpful in designing and implementing a high performance parallel XTP on multiprocessor architectures. However, our experiences have also shown that the XTP specification still includes some ***drawbacks*** hindering more efficient parallel implementations [20]. For example, the use of different types for information and control packets is not really reflected in FSM processing. Several FSMs are involved in user data and control processing. Another disadvantage is the tight co-operation among FSMs performed by signalling events and by sharing common data bases. Communication overhead caused by event signalling among the FSMs and by context information updates is a major limiting factor. Most of the FSMs need to access the connection state information which forms the most important data base. However, the concurrent access of different FSMs to this data base sequentializes their processing.

2.2 Parallel Protocol Design Guidelines

Based on the analysis of XTP FSMs and the performance results of a corresponding multiprocessor implementation [19], we derived the following guidelines for protocol architectures appropriate for parallel processing [20]:

- Data dependencies among protocol functions hinder parallel processing considerably. Some protocol functions are directly data dependent, i.e., their output data forms the input data needed by another function. Additionally, semantical dependencies can occur if multiple outputs of different functions have to be combined to calculate the final result (e.g., different concurrent header analyzing functions). The different results have to be combined for a decision about the correctness and about subsequent processing steps.
- A more loosely structuring of control and data processing by separating them permits a higher degree of parallelism. Independent FSMs for every packet type can allow for highly concurrent processing. The exchange of control packets, which is often triggered by data transfer, can be decoupled from data transfer by requesting control packets explicitly, or by periodic state exchange.
- An orthogonal design of control functions with minimized interactions can provide high degrees of parallelism. This can be achieved by a strict subdivision into independent protocol functions (e.g., separation of acknowledgment from rate control). Such an ap-

proach allows for a modular design, which also supports a flexible protocol configuration as introduced in [21].

- In order to minimize communication overhead among protocol functions, event signalling should be reduced to a minimum. Infrequent periodical signalling may also be sufficient in contrast to signalling events for every received packet.
- Local resources should be used instead of global resources in order to avoid consistency problems and access conflicts. The control functions should operate on independent state variables and use separate control packets.

2.3 PATROCLOS: A Highly Parallel Protocol Architecture

Based on the protocol design guidelines presented above, a highly parallel protocol architecture (named PATROCLOS) has been designed [9]. It uses a fine granularity based on a protocol function oriented decomposition into basic modular building blocks to simplify parallel implementation. Parallel FSMs are the atomic building blocks of the PATROCLOS architecture. FSMs belonging to the same PATROCLOS entity exchange messages for co-operation. Periodical information exchange among FSMs is used to reduce the communication overhead.

PATROCLOS consists of two types of FSMs: interface and protocol FSMs. *Interface FSMs* are located at the interfaces to the application and the network. They are only involved in local communications within a single protocol entity. *Protocol FSMs* communicate directly by separate so-called FSM protocols with the corresponding FSMs at the peer entity (cf. Figure 1). They are designed to allow for parallelism between send and receive part. Moreover, they decouple connection state information exchange from user data exchange. For every FSM protocol a separate PDU is defined as well as individual error recovery mechanisms and timers are used. The PDUs contain only the absolutely necessary information for their dedicated protocol function implemented by the FSM. Multiplexing of PDUs by different FSMs is avoided to support a higher degree of parallelism. Every protocol function is mapped onto a dedicated FSM.

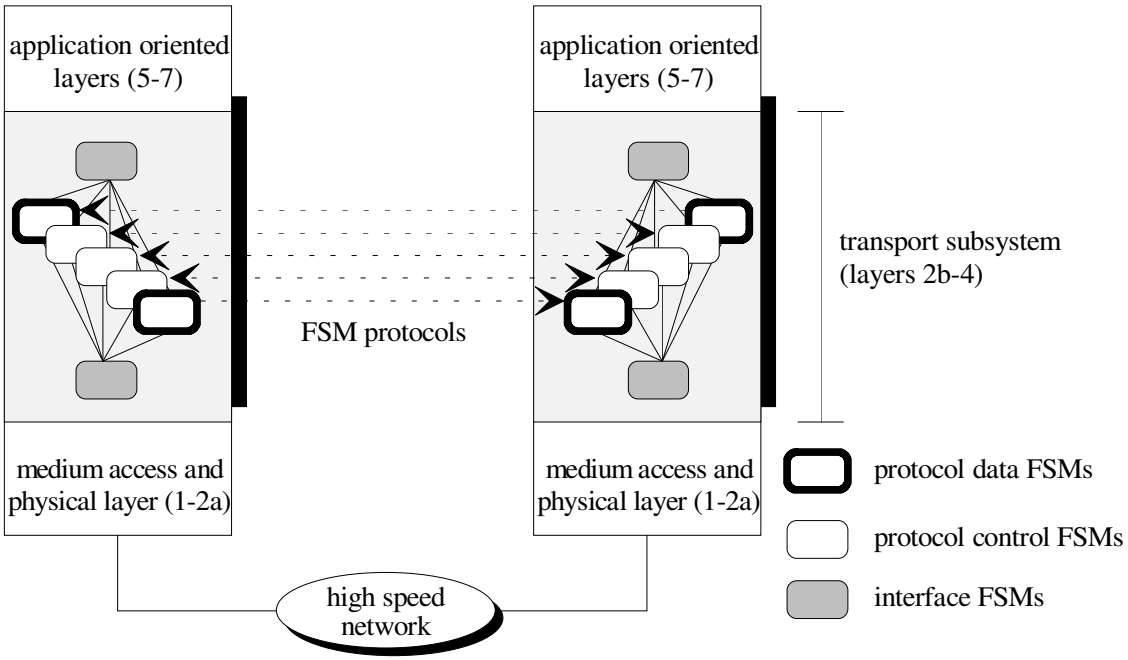

Figure 1: Architecture of PATROCLOS

For the implementation of PATROCLOS a specially suited hybrid multiprocessor architecture has been designed [20]. Performance evaluations show significant improvements compared to similar implementations of other protocols. The receive throughput is limited to 6300 packets/s by the time of the data receive process to analyze a data PDU. The achievable send throughput is limited to 8000 packets/s by the time to format a packet. A TCP/IP implementation [3] running on the same platform as used for performance

evaluation of PATROCLOS achieves less than 3000 packets/s.

The bottleneck processes of that TCP/IP implementation require approximately the double processing time as the PATROCLOS bottleneck processes. One reason therefore is that a lot of control functions within the TCP/IP bottleneck processes such as acknowledgment processing or flow control have been moved to dedicated protocol control FSMs of the PATROCLOS architecture. The performance results indicate that, generally, parallel protocol processing and, especially, parallel processing of control and user data functions is a successful approach towards high performance communication subsystems.

3 Parallel VLSI Implementation Architecture

The inherent parallelism of protocols, such as PATROCLOS, can be applied to parallel VLSI implementation architectures. The VLSI architecture presented in this section additionally implements parallelism among

different connections. The architecture generally distinguishes protocol independent and protocol dependent components.

3.1 General Architecture

The architecture (cf. Figure *2*, [22]) consists of two data memories, for the receive side (*receive RAM*) and for the send side (*send RAM*), respectively. They are managed by specialized extended memory management units (*EMMU*s). The network unit, the applications, and the connection processors (*CP*s) can access the data memories via the *EMMU*s. The use of pointers avoid data copies during protocol processing. Operations on the memory, such as segmentation / reassembly and allocation / deallocation, are completely handled by the *EMMU*s. Applications may read and write data via DMA.

A key feature of the architecture is the replication of identical *CP*s similar to [11] for different connections. They include registers, arithmetic logical units (ALUs), timers, and other components required in a protocol implementation. The main purpose of the remaining components (*management*, *A_MUX*, *N_MUX*, *N_DMX*) is the distribution and collection of relevant data. Received data is divided into user data and a protocol header. User data is written into the *Receive RAM*. The protocol header together with a reference to the user data is delivered to the *N_DMX* that forwards it to an appropriate *CP*.

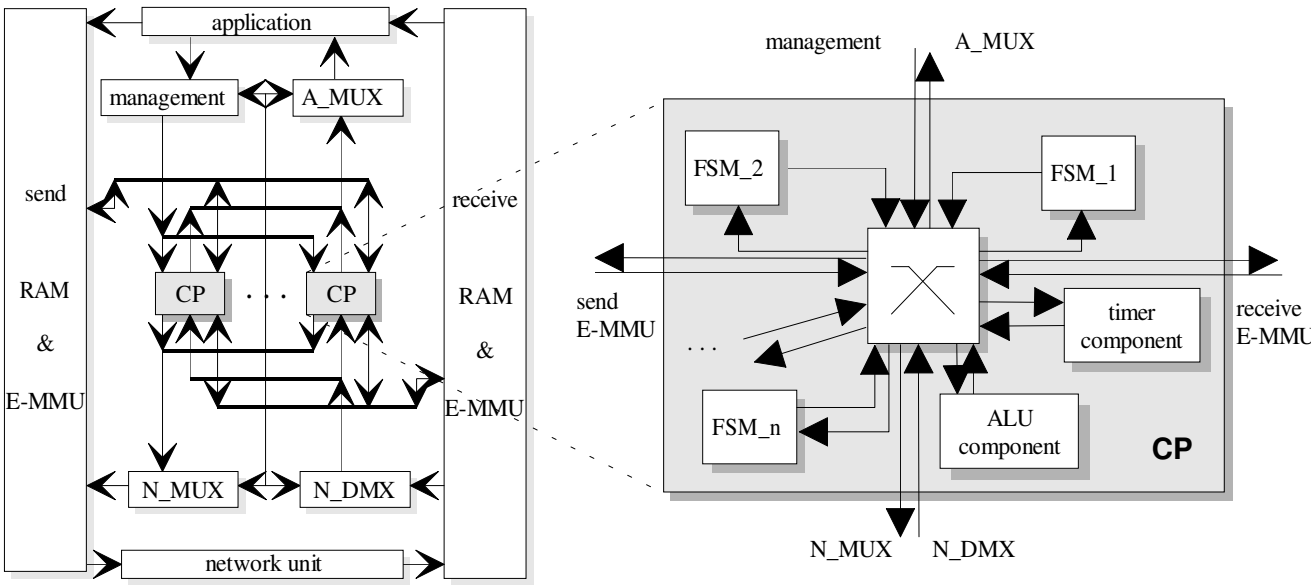

Figure 2: Overview of the VLSI implementation architecture

The *N_MUX* collects protocol headers and references to user data to be sent to the network. The *CP*s cooperate with an application component via *management* and *A_MUX*. The manager maps a request for connection establishment onto a *CP* depending of its actual load, which then is responsible for handling that connection properly. Therefore, this architecture exploits the interconnection parallelism. The mapping information is distributed to the *A_MUX*, *N_MUX* and *N_DMX*.

The four components *management*, *A_MUX*, *N_MUX* and *N_DMX* are connected to the *CP*s

via dedicated busses. The connections between *CP*s and *EMMU*s are used for issuing commands to the *EMMU*s. For example, for segmentation support a pointer to the user data and the length of a segment is given to the *send EMMU* that returns a list of pointers to the segments.

The main components inside a connection processor are the *FSM* processing units that perform all protocol functions. The FSMs of a formal protocol specification can be mapped onto these processing units (e.g., protocol and interface FSMs of PATROCLOS or single FSMs of standard protocols). All *FSM* processing units work independently; they communicate via asynchronous signals, thus, enabling concurrency among different protocol functions. *FSM*s use global or local *Timer*s. In case of global timers, multiple *CP*s can issue commands (e.g., start, restart) to such *timer component*s. In addition, there is a global *ALU* to manipulate global registers that store all variables used by more than one *FSM* (e.g., in XTP, cf. 2.1.1). *FSM*s issue a command to the *ALU* that performs the required operation on the data and may return a result (e.g., in case of comparisons). To guarantee consistency, *FSM*s can access the registers through the *ALU* only. Components within the *CP* are interconnected via a connecting element. To provide a maximum of flexibility they are designed with identical interfaces.

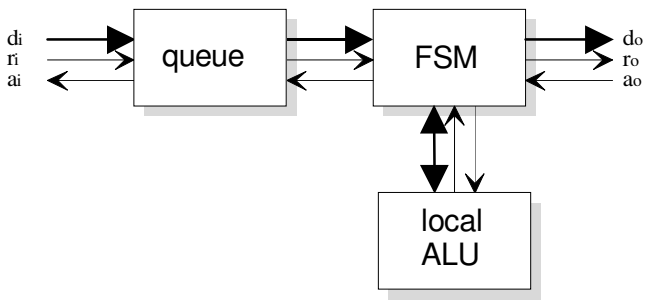

Figure 3: FSM processing unit

The *FSM* processing unit (cf. Figure *3*) is the basis for implementation of parallel FSMs described in a protocol specification. Each of the processing units consists of local registers, a local ALU, and a control unit. Local ALUs are customized for the individual requirements of the corresponding *FSM*. An ALU can be as simple as an adder or as complex as a management unit for a dynamic list. To decouple *FSM*s from each other and from other units inside a *CP*, each *FSM* utilizes a separate input queue. Signals to an *FSM* will be inserted into this queue according to the implicit mechanisms of the protocol specification language SDL (asynchronous communication). Furthermore, an FSM has an interface to issue signals to other components.

Due to its modular design, this architecture can be adapted to different protocols. For example, XTP needs global variables and global timers and, therefore, a global ALU with registers and global timers are required. PATROCLOS needs local variables and local timers only and, thus, the global components are not required.

3.2 Protocol Independent Components

Protocol independent components can be used for the implementation of different protocols. They are fixed in their general structure but flexible enough to be adapted to different requirements. They comprise so-called system functions as well as support functions for protocol functions.

3.2.1 System Functions

System functions perform work related to, e.g., timer management or memory management. They are independent of the protocols and, thus, can be used by different protocols. Two components for system functions are already implemented: an extended memory management unit (*EMMU*) and a list manager for a dynamic list of timers.

The *EMMU* receives commands from the applications, the network, and the *CP*s. It allocates and deallocates memory for user data and protocol headers and it manages the required pointers and data structures. It is assumed, that protocol headers are of a fixed format, i.e., a fixed size and a fixed position for each field in order to allow for fast header parsing. User data may have flexible length.

The timer management unit can receive commands to insert, delete and reset a timer value as well as to report the actual value of a specific timer. If a time-out has occurred, the unit independently issues an alarm signal to the appropriate receiver. Every entry in the timer data structure contains an *InitiatorID*, a *TimerName*, and a time-out value.

3.2.2 Support Functions

Time critical parts of a protocol function are not the state transitions itself, but the management of data structures, such as context data. Examples

are segmentation / reassembly and retransmission of data.

One of the main arguments against hardware architectures is their lack of flexibility, e.g., for handling dynamic lists. However, for some protocol functions dynamic lists are essential. One example is the list that holds all data used for retransmission. Therefore, some protocols need a list of gaps representing spans of bytes not yet correctly transmitted. The sender has to retransmit those bytes if it guarantees correct and complete transmission. A retransmission manager has been developed that manages gaps and can be used for gaps in received data to support the acknowledgment function or for gaps in acknowledgments of transmitted data to support the retransmission mechanism. Every entry contains the following fields: *conn_id, gap_start, length, next*. *Conn_id* indicates the appropriate connection, *gap_start* and *length* hold the lowest unacknowledged sequence number and the length of the gap, *next* points to the next entry of the list. The functions of the retransmission manager are: *set_gap(conn_id, sequence_number, length)*, to insert a new gap, *delete_gap(conn_id, sequence_number, length)*, to delete (parts of) a gap, and *get_gap(conn_id, ptr, sequence_number, length, next)*, to read the content of the register identified by *ptr*. Moreover, there are commands to update registers holding the highest received sequence number and the highest acknowledged sequence number per connection. Besides range checking of all fields in a command, the retransmission manager has to join gaps in case of a register overflow. Therefore, the closest gaps are combined to a single gap.

3.3 Protocol Dependent Components

Protocol dependent components are units that must be (partially) changed if another protocol is implemented. The control unit of an FSM that controls the state transitions and all other operations of the FSM is protocol dependent. Using many protocol independent functions only the FSM state transition tables and the registers have to be adapted to protocol changes. These tables consist not only of (state, nextstate)-pairs, but also of low-level calls of system functions, protocol support functions, and general ALU functions. Up to now changes in the protocol description have to be manually mapped onto changes in the state transition tables.

3.4 Implementation and Simulation Environment

The architecture with its components and interconnections is currently being described with the hardware description language VHDL to allow simulations and synthesis. Based on the description in VHDL, parts of the architecture have already been simulated with a VHDL simulator for validation purposes (discrete event simulation). The protocol used for simulation includes connection management, acknowledgment, retransmission, and other typical functions of transport oriented layers. To retrieve better estimates considering space and time requirements, parts of the design have been synthesized with a high level synthesis tool (Synopsys). An implementation of a list manager for timers using 20 MHz standard cell technique has a performance of more than 3 million insertion operations in the average case.

4 Summary and Future Work

The demand for high performance communication subsystems is steadily increasing during the last few years. Although there have been considerable research efforts, there is still a lack of high performance solutions. The parallel VLSI architecture presented in this paper enables the implementation of such systems by applying parallelism at different levels of granularity and by using the support of dedicated VLSI for potential bottleneck components.

However, the pure design and implementation of a system capable of serving demanding applications cannot be considered as the ultimate goal. Rather, once such a system is designed, there should be some focus on how the implementation productivity can be improved and, even more importantly, how the methods and concepts can be applied to different system environments and communication protocols. Therefore, two main issues form the target of our future work on the presented architecture: providing flexible communication support and enabling automatic or semi-automatic implementation of such advanced communication subsystems from high level protocol specifications.

Flexible communication support can more easily be implemented by pure software solutions. However, by properly designing the protocols and the implementation environment flexibility may also be achieved using hardware-oriented

approaches or dedicated VLSI architectures. In our case, the design principle of desinging a protocol out of a set of almost autonomous protocol functions (specified as FSMs) enables a high degree of flexibility. These building blocks are mapped onto dedicated VLSI components that may be parametrized according to individual application needs.

Automated protocol implementations from high level specifications increase the productivity of protocol implementations. They are usually applied to pure software solutions only. Mostly, even the system environment (such as operating system or workstation architecture) are not considered. We are targeting towards an approach that facilitates the mapping of, e.g., SDL specifications onto VHDL descriptions for the different components of a communication subsystem. Such an approach includes a high potential for efficient implementations suitable for the emerging gigabit networking environment and the increasing variety of application requirements.

References

[1] Zitterbart, M.; *Parallelism in Communication Subsystems;* in: Tantawy, A.N.(ed.): High Performance Networks - Frontiers and Experiences, Kluwer Academic Publishers, 1994

[2] Ito, M.; Takeuchi, L.; Neufeld, G.; E*valuation of a Multiprocessing Approach for OSI Protocol Processing;* Proceedings of the First International Conference on Computer Communications and Networks, San Diego, CA, USA, June 8-10, 1992

[3] Rütsche, E.; Kaiserswerth, M.; *TCP/IP on the Parallel Protocol Engine*, in: Danthine, A.; Spaniol, O. (eds.): High Performance Networking, IV, IFIP, North-Holland, 1993, pp. 119-134

[4] Strayer, W.T.; Dempsey, B.J.; Weaver, A.C.; *XTP: The Xpress Transfer Protocol;* Addison-Wesley Publishing Company, 1992

[5] Feldmeier, D.C.; *An Overview of the TP++ Transport Protocol; in:* Tantawy A.N. (ed.): High Performance Communication, Kluwer Academic Publishers, 1994

[6] La Porta, T.F.; Schwartz, M.; *A High-Speed Protocol Parallel Implementation: Design and Analysis;* in: Danthine, A.; Spaniol, O. (eds.): High Performance Networking, IV, IFIP, North-Holland, 1993, pp. 135-150

[7] Haas, Z.; *A Protocol Structure for High-Speed Communication over Broadband-ISDN;* IEEE Network Magazine, Vol. 5, No. 1, January 1991, pp. 64-70;

[8] Sterbenz, J.P.G.; Parulkar, G.M.; *AXON Host-Network Interface Architecture for Gigabit Communications;* in: Johnson, M. J. (ed.): Protocols for High-Speed Networks, II, North-Holland, 1991, pp. 211-236

[9] Braun, T.; *A Parallel Transport Subsystem for Cell-Based High-Speed Networks;* Ph.D. Thesis (in German), University of Karlsruhe, Germany, VDI-Verlag, Düsseldorf, 1993

[10] Krishnakumar, A.S.; Kneuer, J.G.; Shaw, A.J.; *HIPOD: An Architecture for High-Speed Protocol Implementations*; in: Danthine, A.; Spaniol, O. (eds.): High Performance Networking, IV, IFIP, North-Holland, 1993, pp. 383-396

[11] Balraj, T.; Yemini, Y.; *Putting the Transport Layer on VLSI - the PROMPT Protocol Chip;* in: Pehrson, B.; Gunningberg, P.; Pink, S. (eds.): Protocols for High-Speed Networks, III, 1992, North-Holland, pp. 19-34

[12] Stone, H.S.; *High Performance Computer Architecture;* Addison-Wesley Publishing Company, 1987

[13] Zitterbart, M.; *Parallel Protocol Implementations on Transputers - Experiences with OSI TP4, OSI CLNP, and XTP;* IEEE Workshop on the Architecture and Implementation of High Performance Communication Subsystems, Tucson, AZ, USA., February 17-19, 1992

[14] Jain, N.; Schwartz, M.; Bashkow, T.R.; *Transport Protocol Processing at Gbps Rates;* Proceedings of the ACM SIGCOMM ´90, September 24-27, 1990, Philadelphia, USA, pp. 188-199

[15] Braun, T.; Rütsche, E.; Kaiserswerth, M., Zitterbart, M.; *Implementation of Communication Protocols for High-Speed Networks on the Parallel Protocol Engine;* internal report

[16] Koufopavlou, O.G.;Tantawy, A.N.; Zitterbart; M.; *Parallelization of TCP/IP for Very High Speed Networks;* 17th Annual IEEE Conference on Local Computer Networks, Minneapolis, MN, USA., 13.-16. September 1992, pp. 576-585

[17] Sabnani, K.; Netravali, A.; Roome, W.; *Design and Implementation of a High Speed Transport Protocol;* IEEE Transactions on Communications, Vol. 38, No. 11, November 1990, pp. 2010-2024

[18] Protocol Engines Inc.; *XTP Protocol Definition;* Revision 3.4, July 17, 1989;

[19] Braun, T.; Zitterbart, M.; *A Parallel Implementation of XTP on Transputers;* 16th Annual IEEE Conference on Local Computer Networks, October 14-17, 1991, Minneapolis, MN, pp. 172-179

[20] Braun, T.; Zitterbart, M.; *Parallel Transport System Design;* in: Danthine, A.; Spaniol, O. (eds.): High Performance Networking, IV, IFIP, North-Holland, 1993, pp. 397-412

[21] Zitterbart, M.; Stiller, B.; Tantawy, A.; *A Model for Flexible High-Performance Communication Subsystems;* IEEE-JSAC, May 1993, pp. 507-518

[22] Schiller, J.; Braun, T.; *VLSI-Implementation Architecture for Parallel Transport Protocols;* IEEE Workshop on VLSI in Communications, Stanford Sierra Camp, Lake Tahoe, CA, USA, September 15-17, 1993