PARALLEL XTP-IMPLEMENTATION ON TRANSPUTERS

Torsten Braun

Institute of Telematics, University of Karlsruhe
P.O. Box 6980, W-7500 Karlsruhe, Germany
E-Mail: braun%i70vca@iravcl.ira.uka.de

Martina Zitterbart

IBM Research Division, Thomas J. Watson Research Center
P.O. Box 704, Yorktown Heights, NY 10598, U.S.A.
E-Mail: zit@watson.ibm.com

Abstract

High performance communication systems have to provide very high data rates of more than 100 Mbit/s to new applications such as multimedia applications. The bottleneck in communication systems is the protocol processing of the layers above media access control layer. Especially the transport and the network layer are very time consuming components of the protocol stack. To overcome this bottleneck light weight protocols such as XTP (Xpress Transfer Protocol) can be implemented on multiprocessor systems. XTP is a protocol with functionality of layers 3 and 4 and was designed for protocol processing in reliable high speed networks such as FDDI. XTP is based on several finite state machines to support an efficient implementation in VLSI. This paper presents a software implementation of XTP for a multiprocessor system. The implementation is based on these finite state machines, which are designed to run in parallel. The finite state machines are mapped onto processes, which can run in parallel on the processors of a multiprocessor architecture. The processes communicate with each other by message passing.

1. Introduction

In communication systems during the last few years, transmission rates of communication networks have evolved faster than the performance of protocol processing. Often nowadays, the bottleneck in communication systems is the protocol processing of the layers above the media access layer.

Recently, applications such as multi-media-applications are coming up, which require very high performance communication systems. High data rates of more than 100 Mbit/s as well as low delay and delay restrictions according to isochronous data traffic are required. To provide the performance requirements of those applications there are several approaches:

- Existing protocols, e.g. OSI protocols or TCP/IP, are implemented efficiently by the use of special implementation techniques [1], hardware [2], or multiprocessor architectures [3].
- So called light weight protocols such as the Xpress Transfer Protocol (XTP) [4] and the Versatile Message Transaction Protocol (VMTP) [5] or special purpose protocols have been developed. Several mechanisms of conventional protocols are often considered as unsuitable for high speed networks. Moreover, conventional protocols often restrain efficient protocol implementations in hardware or on multiprocessor architectures.

This paper combines the two approaches of light weight protocols and protocol implementation on a multiprocessor architecture: a software implementation of the XTP protocol on a parallel architecture is presented.

Section 2 focuses on features of the XTP protocol and the several finite state machines, which are designed to run in parallel. In section 3 the implementation environment is described, and the XTP implementation for the multiprocessor architecture is presented. First performance results are discussed also in this section. Finally conclusions in section 4 will summarize the paper.

2. The XTP-Protocol and its Finite State Machines

2.1. XTP Overview

XTP is a transfer protocol designed for the use in high speed networks from 100 Mbit/s to 1 Gbit/s. It comprises functions of the transport and the network layer according to the OSI reference model. XTP offers a connection oriented service, but it is also appropriate for connectionless data transfer. Error, rate, and flow control are realized. Moreover it supports several addressing mechanisms, e.g. OSI, Internet, etc.

The design of XTP was influenced by the following ideas:

- The receiver part often is the bottleneck on data transfer in conventional protocols. Several functions moved from the receiver to the sender part to make the receiver simpler.
- XTP offers a connection oriented service. Datagrams may also be transferred efficiently and with low overhead supported by short-lived connections. A virtual connection consists of two endpoints and is identified by an identifier called *key*. The set of state variables at one endpoint is called context.
- XTP was designed to execute protocol processing in parallel. Several finite state machines (FSMs) perform different tasks, e.g. input or output of data, rate control, context managing etc. The FSMs interoperate by events and by operating on some common data. A FSM of a special type is created for every context processed by the instance.
- XTP uses two packet formats, an information packet and a control packet. The information packet contains user data. The control packet contains the receiver's error, flow and rate control parameters. Both packet types consist of a header, a data segment (user or control data), and a trailer. Header and trailer have fixed lengths of 24 and 16 bytes.
- XTP was not designed for special kinds of networks or for special application types. XTP is intended for an universal protocol. To minimize unnecessary functions XTP offers different modes, options, and formats for several requirements.

Several mechanisms are realized to achieve the generality mentioned above. Connections can be established and closed implicitely and explicitely, e.g. by service primitives. The implicit connection establishment and closing is supported by FIRST and END flags in the packet header or trailer. If a packet with a FIRST flag arrives, a connection is created. The connection is closed, when a packet with an END flag has been received.

When a connection between end systems has to be established through one or more routers, these routers have to do the routing decision. Each connection through a router is identified by a value called *route*. A route is generated with a FIRST packet. Subsequent packets of this connection contain the same route values.

XTP uses a byte oriented flow control mechanism such as in TCP. The receiver's control parameters are requested by the sender. They contain information about its buffer state. The receiver delivers them within control packets to the sender.

Besides the window based flow control mechanism there is also a rate control mechanism that restricts the size and the time spacing of bursts of data from the sender.

Similar to TCP and OSI TP4 checksum calculation is used to detect bit errors and to initiate retransmission. Checksum is placed in the trailer, so it can be calculated while a packet is being sent. Corrupted packets are discarded by the receiver without any other actions. Only when a packet with a sequence number out of order arrives, reactions are triggered at the receiver side. For error recovery the protocol allows go-back-n or selective retransmission.

XTP uses a few timers only. The most important ones are the connection, the path, the wait, and the rate control timer. The two former ones expire when connections or routes have been inactive for a certain time. The wait timer is used to monitor the receiver's reaction after a request for a control packet by the sender. When the wait timer expires, the sender attempts re-synchronization. The rate control timer is explained below.

As already mentioned above, XTP is designed to be implemented in VLSI. The protocol consists of a couple of finite state machines (FSMs) to support efficient parallel execution. The implementation presented in this paper is based on these FSMs. The FSMs are described in the next subsection.

2.2. Finite State Machines

There are several types of FSMs that perform different tasks. Some FSMs are multiplied for every context and some for every route. The FSMs cooperate on common data, e.g. context data, and by signalling events to other FSMs. FSM state transitions are triggered by those events and are dependent on the contents of current context data.

There are different types of events: network, timer, internal, and external events. Network events occur when packets arrive from the network. External events are triggered by the layer above XTP (XTP user). Internal events are created by the XTP FSMs and are signalled to other XTP FSMs. Timer events are triggered by expired timers. Timers are used by several FSMs, examples are given below.

The central FSM is the ***context manager state machine*** (CMSM) that wakes up other FSMs of a certain context. CMSM becomes active when a new connection has to be created. Then, CMSM creates a new context, signals events to the other FSMs to activate them, and monitors the corresponding connection with the connection timer. After a connection has been closed, the corresponding context is released. The CMSM and the other FSMs return to idle state.

Ten further FSMs are specified in [6]: output, sync, rate control, control send, input, reader, write close, read close, router state machine, and the route manager.

The ***output state machine*** (OSM) has to decide whether, which, and how much data has to be sent to the network. The OSM contains the sender's flow control activities, checks the flow control parameters, and decides whether control packets must be requested from the receiver.

The ***sync state machine*** (SSM) supports the synchronizing handshake that is used to synchronize control communications between two hosts. A sender increments a packet counter whenever a data packet has been transmitted. The current counter value is sent in the *sync* field of control packets. The receiver returns this value in the *echo* field of its control fields. The sender matches sync and echo fields and can decide to retransmit packets sent prior to the control packet with this sync value. Sync/echo packets are also used to estimate current round trip time.

Rate Control is supported by the ***rate control state machine*** (RCSM). The RCSM is periodically interrupted by the rate control timer to reset the *credit* value. Credit indicates how many bytes the sender can transmit to the according receiver in a certain time period. OPSM decrements the credit value whenever data has been sent. By this mechanism the amount of data sent during a certain time period is restricted.

The ***control send state machine*** (CSSM) creates and formats control packets. It is triggered by various other FSMs, when those demand to send control packets. CSSM competes with OSM for network access.

The data and control packets on the receiving data path are processed by the ***input state machine*** (ISM) and the ***reader state machine*** (RSM). ISM checks addresses, the sequence of data packets, updates context data, causes sending of control packets to the associated sender, if necessary, and delivers data packets to the user via RSM.

The ***write*** (WCSM) and ***read close state machine*** (RCSM) perform closing of connections. It is checked whether data sent earlier arrived correctly at the associated receiver before closing the writer (sender). A receiver has to acknowledge received data before closing.

Two FSMs exist for routing activities. The ***host route manager*** (HRM) monitors the routes being active supported by a path timer. The router state machine (RSM) performs routing and route exchange.

3. The XTP-Implementation on Transputers

3.1. The Implementation Environment

The Transputer [7] is a family of high performance microcomputers with a processor, a timer, on-chip memory, an external memory interface, and four communication links on a single chip.

One of its most significant features is the ability to perform multi-tasking efficiently in hardware. The Transputer uses processes as fundamental standard software building blocks. Processes are able to communicate using the synchronized and unbuffered rendez-vous mechanism.

Transputer networks are loosely coupled multiprocessor systems with distributed memory. They are built by connecting the links of several Transputers.

The Transputer was developed concurrently with the programming language OCCAM, but it is also well suited to other languages, e.g. the C language. During the project described in this paper a parallel TCC (Transputer C Compiler) [8] is being used, which provides the most significant features of the OCCAM language including the datatype "channel", the "alt"-construct, and the "timer" variable. The "alt"-construct allows an alternative waiting for inputs on different channels and for timer events.

A so called configurator was developed for flexible mapping of processes to Transputers. By this tool the various processes can be distributed to the available processors and also the processes' communication channels are connected together.

3.2. Design Considerations

For the design of the XTP implementation different aspects have been considered. First, it should be possible to insert the XTP implementation into an implementation of a protocol stack on a multiprocessor architecture according to the concepts also presented in [3]:

Communication systems are typically structured in hierarchical protocol layers, as suggested by the OSI reference model. The protocol layers operate in a pipelined principle: each packet moves through the protocol layers step by step. Thereby several packets may be concurrently in different protocol layers. Some of the layers can even be divided into sublayers. Such a layer pipeline performs overlapped computation of sublayers to exploit parallelism. Additional to hierarchical structuring of sublayers the protocol entities of those layers can be divided internally into a send and a receive part resulting in a send and a receive pipeline for each network node (cf. figure 1).

Usually, the send and the receive pipeline are not completely independent of each other. For example, the sender and the receiver part have to exchange control information such as connection state data in a connection oriented protocol or data units in a gateway system.

To avoid movements of data packets between different layers in a protocol stack, a global data memory must be provided, to which all layers must have access. Inside this data memory all data packets remain at a fixed place while being processed by the layers of the protocol stack. The data packets are not moved when passing from one layer to another. The different layers only add (send side) or remove (receive side) the packet header or trailer to or from the data unit.

Because all the protocols of a stack are intended to be implemented on a multiprocessor architecture, there is the question how to organize a global data area on multiprocessor systems. There are two possibilities:

- The global data can be stored in a shared memory to which all processors have access.
- The global data can be stored in a local memory of a dedicated processor. The other processors, which want to read or write data, have to send requests to a memory manager process on the dedicated processor. The memory manager delivers the requested data or writes the received data in his local memory.

Because Transputers do not support shared memory very well the latter possibility is implemented. The global memory is divided into two parts, one part for data of the sender functions and one part for data of the receiving ones of the protocol stack. For each global memory part manager processes are implemented to manage the access to the global memory.

This concept reduces copying of data significantly. Only references to data packets stored in the global memory are delivered between adjacent layers. One disadvantage of the concept may be the overhead caused by the communication between the protocol processes and the memory management processes. In gateway systems data may have to be copied, if there is always a memory part for each network.

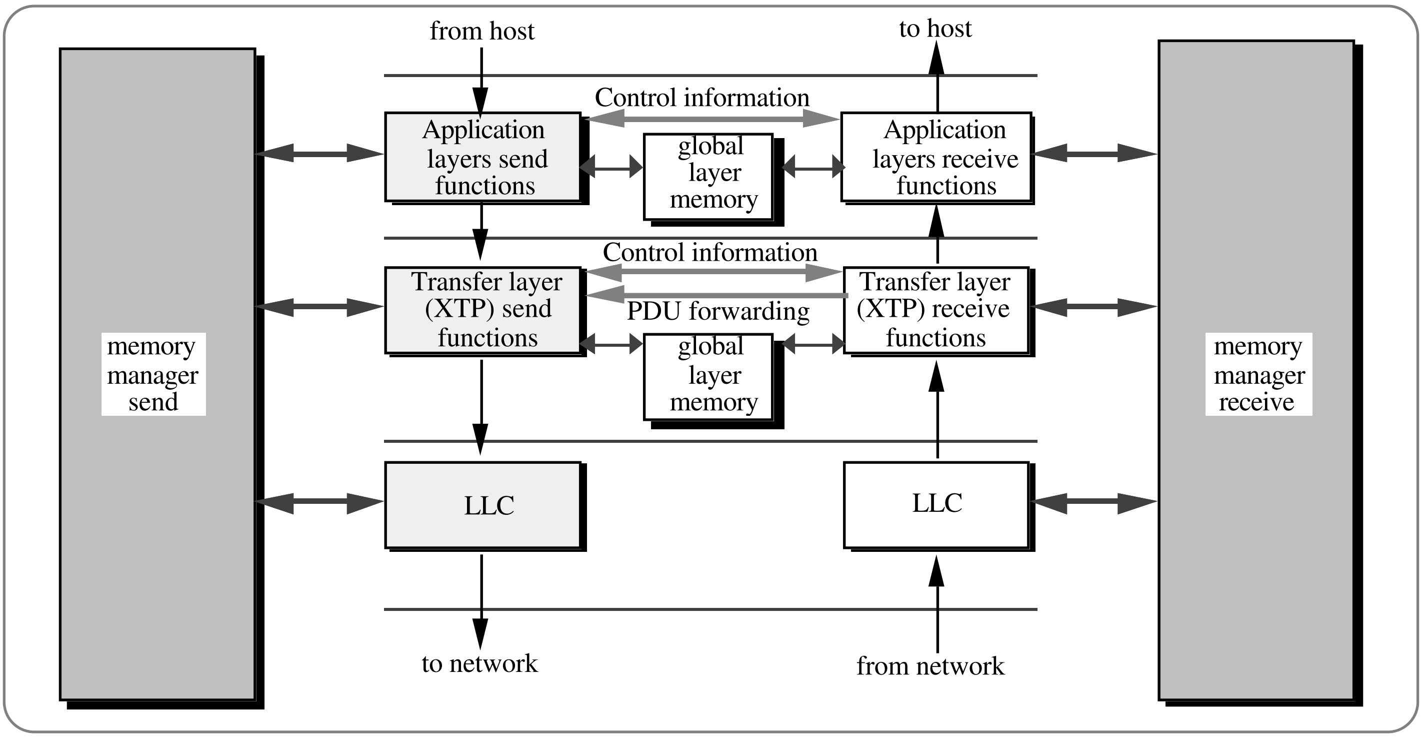

Figure 1: Parallel Protocol Stack Implementation

For test purposes a special environment consisting of test user processes, a network simulation process, two memory manager processes and the XTP protocol implementation were used (cf. figure 2).

Another design issue is how to implement the XTP FSMs. It is impossible to create a process for every FSM because for each connection or route and for each FSM type there exists always one FSM. The number of FSMs could become very high. In consequence, performance would decrease by process switching. Also creation and release of processes are very time consuming.

The presented implementation creates a separate process for every FSM type. This process is responsible for every created context or route. FSM state transitions are triggered by events and predicates. These predicates are mainly context data, which are common for several FSMs. Event signalling between processes is done by passing messages across channels. For two processes, i.e. for two FSM types, there exists exactly one channel, if the two processes are signalling events between each other. Various event types can be signalled across this channel. A message identifier indicates the event type and the corresponding context. The different protocol processes operate on common context data. To allow all processes to run on different Transputers, the context data is distributed as well as possible to those processes that mainly operate on it. Context data needed by several processes have to be exchanged whenever any context values change.

The primary goals of this XTP implementation are efficiency and flexibility. For that reason, mainly small FSMs that exchange a lot of events with other FSMs and that share a considerable amount of common data with other FSMs are combined to a product FSM. Too fine process granularity causes high communication overhead because of the communication concepts realized on Transputers. The resulting FSMs are implemented by a single process that can run on a single processor or that can run concurrently with other processes on a processor.

3.3. Realization

The process net of the implementation [9] is depicted in figure 2. There are two FSMs not realized in the current implementation, which are only needed, if the communication system must have gateway functionalities.

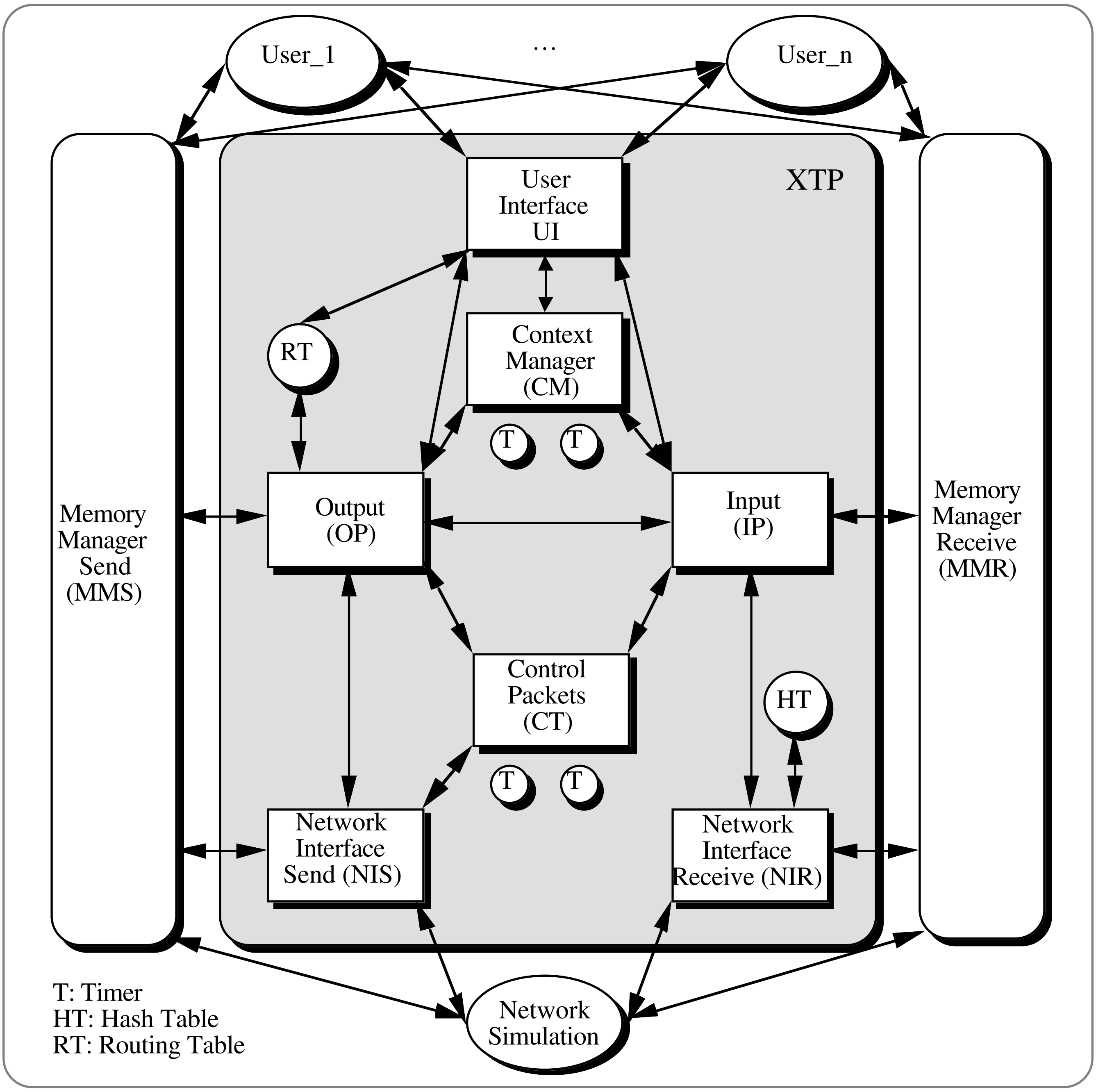

Figure 2: Process Net of the Parallel XTP Implementation

The implemented XTP FSMs are mapped to seven processes: user interface (UI), input (IP), output (OP), control packet (CT), context manager process (CM), and two network interface processes for send (NIS), and receive (NIR) part. Additionally, there are timer processes and processes to manage data structures such as hash and routing tables. The routing table process (RT) determines a route for a connection to another instance. The hash table process (HT) maps incoming keys to a local context. HT is used by NIR when packets arrive.

The Protocol Processes in Detail

The ***user interface process*** (UI) is the interface between send and receive part of XTP and the users. Only UI represents the XTP instance to the users. Different users can use UI at the same time by sending a source identifier with each message, i.e. service primitive. RSM, RCSM and WCSM are implemented inside UI. UI has also the task to deliver data to the user and to control closing of connections. A TSAP (transport service access point) is assigned to every connection. A TSAP consists of an entity address and the connection endpoint identifier (here: context).

The sender part of XTP consists of the processes OP, RT, CT and NIS. The ***Output process*** (OP) comprises functions of OSM and RSM. OP has to send data packets to the network and to perform functions such as flow control, rate control, and retransmission of lost or corrupted packets. OP uses the ***routing table process*** (RT) to determine the route values.

The ***control packet process*** (CT) has to react on requests of other processes that trigger sending of control packets (cf. CSSM). The control packets are necessary for acknowledgement or synchronization procedures, i.e. CT performs SSM.

The ***network interface send process*** (NIS) is the interface between the XTP send part and the lower layers. NIS supports OP to perform OSM. NIS controls access to the underlying network and delivers control or data packets to the network. Another task is to complete the packet trailer and to send it to the memory manager. Headers are generated by CT or OP dependent on sending a control packet or a data packet. Keeping context data necessary for header or trailer generation is distributed among the three processes CT, OP, and NIS.

Similar to NIS, the ***network interface receive process*** (NIR) is the interface between the XTP receive part and the network. NIR uses HT to get the local context of an incoming XTP header. Then, NIR delivers the packet to the ***input process*** (IP). IP and NIR perform ISM. IP distributes new context values to those processes, where context data is stored. Morever, IP triggers reactions as control packet requests or connection releases dependent on the header and the trailer information.

The central process of the XTP protocol implementation is the ***context manager process*** (CM) that is executing CMSM. CM triggers initializing context data and computes keys for the created connections. After connection establishment CM monitors the connections by means of the connection timer. After closing a connection CM triggers the release of the context data.

There are other processes in the test environment besides the XTP protocol processes. These processes can be classified as test processes and memory manager processes. The network simulation process substitutes the underlying network. This process transfers XTP packets between XTP instances. Test user processes are implemented instead of higher layers.

The tasks of ***memory manager processes*** for ***send part*** (MMS) and ***receive part*** (MMR) have been already described above. Cooperation between memory manager processes and protocol processes is organized in a client/server principle. Protocol processes (clients) send requests to the memory manager processes (servers) to read or write data packet information and to allocate or release buffer space. Memory manager processes perform the desired actions and return the results to the clients. For example, a protocol process must send a command to MMR, if the process wants to read the received XTP header fields. MMR returns the header data fields.

The Multi-Transputer Implementation of XTP

The presented XTP implementation can easily be implemented on various Transputers by mapping processes to processors and interprocess communication connections to software or hardware channels (links) dependent on two processes being mapped to the same Transputer.

If there are more than four interprocessor channels required for one processor, so called multiplex-processes are inserted. The multiplex-processes multiplex several logical interprocess channels to a reduced number of interprocessor channels. (cf. figure 3). These processes decrease the number of required links significantly. Additional overhead caused by the multiplex-processes is mostly very low because interprocessor communications can run in parallel to other computations.

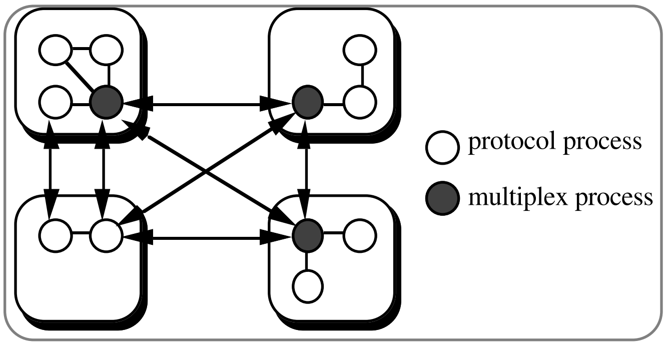

Figure 3: Multiplex-Process

The processes need not to be interconnected directly, if they have to communicate with each other. There is a mechanism implemented in every process to route messages not belonging to the process to the right destination process. Routing is done by the use of routing tables that are mapping incoming messages to an output channel by means of interpreting the destination process identifier. This mechanism is only used if necessary because it causes additional overhead.

Every process also starts a subprocess, the so called communication-process, that is doing interprocess communication across channels (cf. figure 4). A communication-process receives incoming messages from another main process and stores the messages in a queue. The communication-process cooperates with the according main process by this queue. The communication-process wakes up the main process whenever a message arrives. Then, the main process performs computations dependent on the incoming message.

If interprocessor communication is necessary, the main process can do the computations while the communication-process performs interprocessor communications. Additonally, these communication-processes provide asynchronous interprocess communications, deadlocks are avoided.

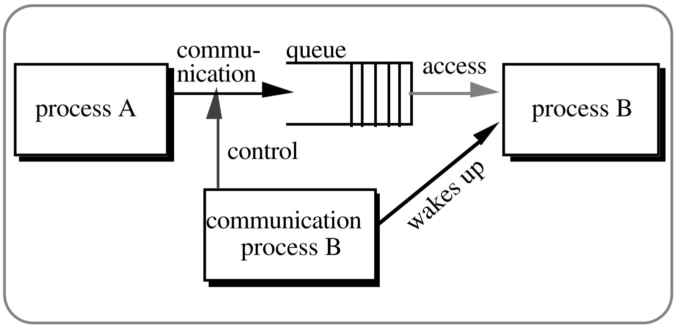

Figure 4: Communication-Process

The design of the multiprocessor implementation is supported by the mentioned configurator program. The implementor has to describe the configuration in a description file only. The file must contain the mapping of processes to processors and the connections of the processes' input and output channels. Generation of boot programs to initialize the Transputer network and generation of routing tables mentioned above is based on the implementor's description and is done automatically by the configurator.

3.4. Performance Results

Figure 5 presents first performance results of the current implementation [10]. The results show how much time the processes spend for sending and receiving a packet. After establishing a connection nine packets are sent. Processing times for UI, CM, and RT are independent of the number of packets.

The network interface processes, NIR and NIS, are the bottlenecks with about 750 µs (NIR) and 840 µs (NIS) processing time per packet including communications. These processing times reduce the maximum achievable performance on a multiprocessor configuration with one processor for a single process to 1330 and 1190 packets/second.

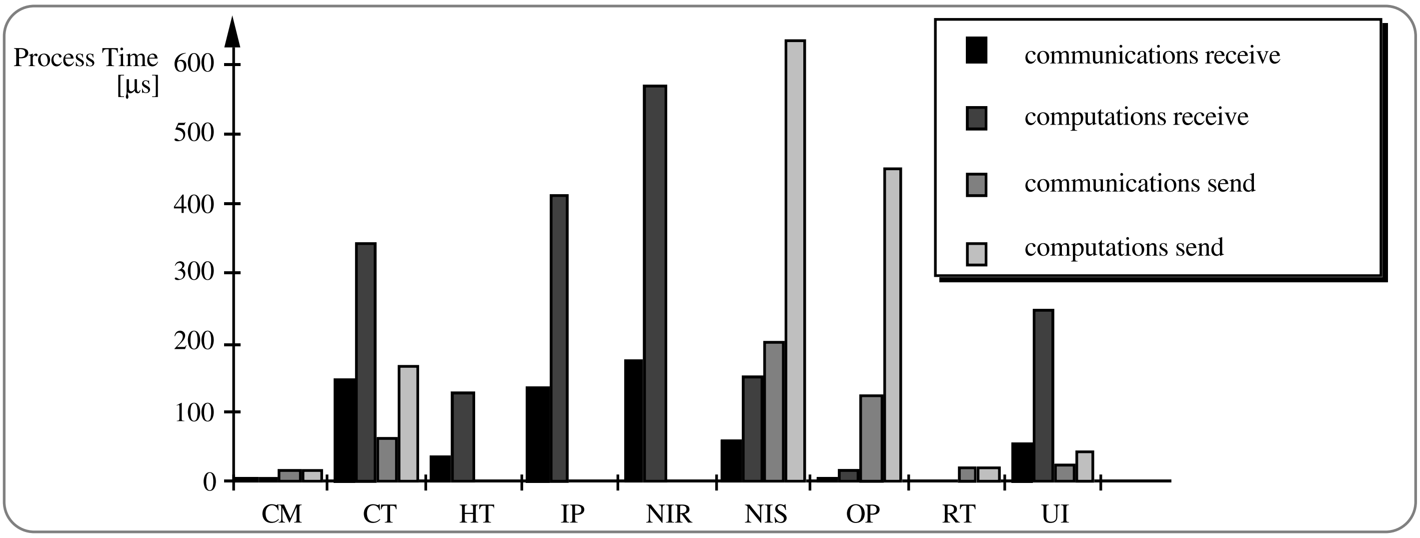

Figure 5: Performance Results

Summation of process times without communications comes up to 1900 µs on receiving and to 1350 µs on sending. Parallel execution will have the most advantages on receiving. The communication overhead is still high: it comes up to 25 %. Efforts to reduce communication overhead will be necessary. Several Transputer network configurations will be examined and evaluated.

4. Conclusions

In this paper an implementation of XTP for a communication system based on a multiprocessor architecture is presented. The XTP FSMs and the mapping to the processors are discussed. Resulting processes are atomic units for distributing to processors of a multiprocessor system.

First performance results show that higher performance values are achievable by the use of a multiprocessor system. XTP seems to be appropriate for parallel processing. Improvements, completion and more detailed performance evaluations of the implementation are performed currently.

References

[1] C. Woodside, K. Ravindram, R. Franks
The Protocol Bypass Concept for High Speed OSI Data Transfer
2nd IFIP WG6.1/WG6.4 International Workshop on Protocols for High Speed Networks, November 1990, Palo Alto, USA.

[2] M.A. Sidenius
Hardware Support for Implementation of Transport Layer Protocols
2nd IFIP WG6.1/WG6.4 International Workshop on Protocols for High Speed Networks, November 1990, Palo Alto, USA.

[3] M. Zitterbart
High Speed Transport Components
IEEE Network Magazine, January 1991.

[4] R.M. Sanders, A.C. Weaver
The Xpress Transfer Protocol (XTP) - A Tutorial
Computer Communications Review, October 1990.

[5] D. Cheriton, C. Williamson
VMTP as the Transport Layer for High-Performance Distributed Systems
IEEE Communications Magazine, Vol. 27, No.6, June 1989.

[6] Protocol Engines Inc.
XTP Protocol Definition
Revision 3.4, 17 July 1989.

[7] Inmos Ltd.
Transputer Reference Manual
Prentice Hall, 1988

[8] Definicon Systems Inc.
The Parallel Transputer System: Operation and Installation Manual
Release 2.0, 1988

[9] A. Schörk
Parallele Realisierung von XTP auf Transputern
Diploma thesis, University of Karlsruhe, October 1990.

[10] Th. Weber
Untersuchung und Adaption des XTP-Protokolls auf Transputer-Netzwerken
Diploma thesis, University of Karlsruhe, July 1991.