IFIP Transactions C:
Communication Systems

International Federation for Information Processing

Technical Committee 6

Communication Systems

HIGH PERFORMANCE NETWORKING, IV

Proceedings of the IFIP TC6/WG6.4 Fourth International Conference on
High Performance Networking
Liège, Belgium, 14-18 December, 1992

Edited by

A. DANTHINE

Institut d'Electricité B28
Université de Liège
Liège, Belgium

O. SPANIOL

RWTH Aachen
Informatik IV
Aachen, Germany

N·H

1993

Keywords are chosen from the ACM Computing Reviews Classification System, ©1991, with permission.
Details of the full classification system are available from
ACM, 11 West 42nd St., New York, NY 10036, USA.

This book is printed on acid-free paper.

Printed in The Netherlands

Table of Contents

hpn 92

TCP/IP on the Parallel Protocol Engine

Erich Rütsche and Matthias Kaiserswerth

IBM Research Division, Zurich Research Laboratory
Säumerstrasse 4, 8803 Rüschlikon, Switzerland

Abstract

In this paper, a parallel implementation of the TCP/IP protocol suite on the Parallel Protocol Engine (PPE), a multiprocessor-based communication subsystem, is described. The execution times of the various protocol functions are used to analyze the system's performance in two scenarios. In the first scenario we execute the test application on the PPE; in the second we evaluate the potential performance of our TCP/IP implementation when it is driven by an application on the workstation. For the second scenario, the end-to-end performance of our implementation on a four-processor PPE system is more than 3300 TCP segments per second.

Keyword Codes: C.1.2; C.2.2; D.1.3
Keywords: Multiple Data Stream Architectures (Multiprocessors); Network Protocols;
            Concurrent Programming

1. INTRODUCTION

Progress in high-speed networking technologies such as fiber optics have shifted the bottleneck in communications from the limited bandwidth of the transmission media to protocol processing and the operating system overhead in the workstation. So-called *lightweight* protocols and protocol offload to programmable adapters are two approaches proposed to cope with this problem. Protocols such as the Xpress Transfer Protocol (XTP)[1] [PEI 92] and VMTP [Cheriton 88] try to simplify the control mechanisms and packet structures such that the protocol implementation becomes less complex and can possibly be done in hardware. We took the second approach in building the Parallel Protocol Engine (PPE) [Kaiserswerth 92], a multiprocessor-based communication adapter, upon which protocol processing can be offloaded from a host system. The Nectar CAB [Arnould 89] and the VMP Network Adapter Board [Kanakia 88] are other programmable adapters, each based on a single protocol processor. The XTP chipset [Chesson 87] is a very specialized set of RISC processors designed to execute the XTP protocol. Our objective was to investigate and exploit parallelism in many different protocols. Therefore we decided to develop a general-purpose communication subsystem capable of supporting standard protocols efficiently in software.

---

[1] Xpress Transfer Protocol and XTP are registered trademarks of XTP Forum

In this paper our goal is to demonstrate that a careful implementation of a standard transport protocol stack on a general-purpose multiprocessor architecture allows efficient use of the bandwidth available in today's high-speed networks. As an example, we chose to implement the TCP/IP protocol suite on our 4-processor prototype of the PPE.

We implemented the socket interface and a test application directly on the PPE to facilitate our performance measurements. In this test scenario we analyze the performance of TCP/IP and the socket layer. We also examined a second scenario to understand how our implementation would perform when integrated into a workstation, where protocol processing up to the transport layer is performed on the PPE and applications can access the transport service via the socket interface on the workstation.

In Section 2 our hardware platform, the PPE, is presented. Section 3 introduces TCP/IP. In the following section we explain our approach to parallel protocol implementation. Section 5 presents the results and discusses the impact of the hardware and software architecture on performance. The last section gives the conclusion and an outlook on our future work.

2. THE PARALLEL PROTOCOL ENGINE

The PPE is to be presented only briefly here. It is described in greater detail in [Wicki 90] and [Kaiserswerth 91, 92]. We will first concentrate on the hardware and then present the programming environment.

The PPE is a hybrid shared-memory/message-passing multiprocessor. Message passing is used for synchronization, whereas shared memory is used to store service primitives and protocol frames. Figure 1 shows the architecture of the PPE and its use as a communication subsystem.

The PPE uses two separate memories, one for transmitting, one for receiving data. Both of these memories are mapped into the address space of the workstation. In our implementation, four T425 transputers [INMOS 89] are used as protocol processors. On each side of the adapter, two T425s have access to the shared memory. Each processor uses private memory to store its program and local data. We decided against using a single shared memory for storing both inbound and outbound protocol data, although this would make the adapter more flexible and facilitate programming, for the following reason. High-speed network interfaces work in a synchronous fashion, with data being clocked in and out of memory, possibly at the same time, at the transmission speed of the physical network. Splitting the adapter into separate receive and transmit parts accommodates simultaneous transmission and reception and only requires memory with half the speed of that required for a single-memory solution. This architecture results in significant cost savings, especially when transmission speeds exceed 100 Mb/s.

The network interface has read access to the transmit side and write access to the receive side of the adapter. We emulate a physical network by means of an 8-bit wide parallel interface, which allows a point-to-point connection between two PPE systems operating with a bidirectional transmission rate of up to 120 Mb/s. The transputer links are used exclusively for signalling and control message transfer within the PPE and to and from the host system.

The programming language which best describes the transputer's programming model is OCCAM [Pountain 88]. It is based on the theory of *Communicating Sequential Processes* (CSP) developed by Hoare [Hoare 78]. The structuring elements are processes that communicate and synchronize via messages. Message transfer is unbuffered; communicating processes must reach

Application | **Protocol Layers** | **Physical Layer**
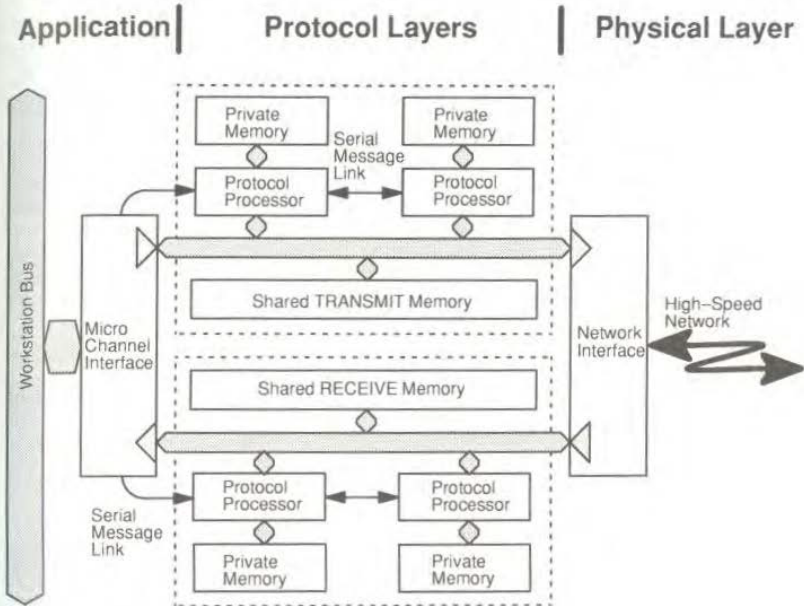
Figure 1. Architecture of the PPE

a *rendezvous* before the message is copied directly from the sender's to the receiver's address space. This behavior maps directly to the transputer's register model and microcode, which support efficient context switches and transparent message passing via four external links and any number of internal *soft* channels. However because OCCAM discourages the use of pointers and shared memory between different processes and offers very little support of user-defined structured data types, we chose to do our implementation in the C programming language [LS-C 89]. Access to the transputer specific facilities, such as synchronous message passing and process control, is provided through library functions, which can, in part, also be generated as more efficient inline code by the compiler.

3. THE TCP/IP PROTOCOL STACK

We implemented the full TCP/IP protocol stack on the PPE. It consists of the *Internet Protocol* (IP), the *Internet Control Message Protocol* (ICMP), and the *Transmission Control Protocol* (TCP). Applications interface to the protocol implementation via sockets, similar to the BSD version of Unix[2].

---

[2] Unix is a registered trademark of AT&T in the United States and other countries.

IP is a datagram protocol that implements functions similar to those of the OSI Connectionless Network Protocol (CLNP). ICMP, which is an integral part of IP, is used to exchange control messages between internet clients, e.g., it generates a destination unreachable message when the addressing information in a received datagram does not allow forwarding or local delivery. TCP, which roughly implements the ISO Transport Layer functions, provides an error- and flow-controlled end-to-end transport connection between applications. TCP thus builds reliable data transmission services on top of the unreliable IP datagram service. A TCP connection is specified through the pair of Internet addresses and the TCP port identifiers of the two communicating partners. The socket is the local end point of a TCP/IP connection. The application program accesses sockets through local identifiers, similar to file descriptors in Unix.

As we did not want to implement TCP/IP from scratch, we based our work on a version of TCP/IP for MS/DOS from the University of Maryland [UM 90].

4. PARALLEL IMPLEMENTATION OF TCP/IP

To develop a parallel solution one needs to partition the problem into a set of subproblems that can be executed in parallel. The algorithms solving these subproblems are typically encapsulated in cooperating processes which are mapped to the parallel-processor hardware. Depending on the underlying hardware and the implementation model chosen, these processes communicate and synchronize via shared memory or message passing.
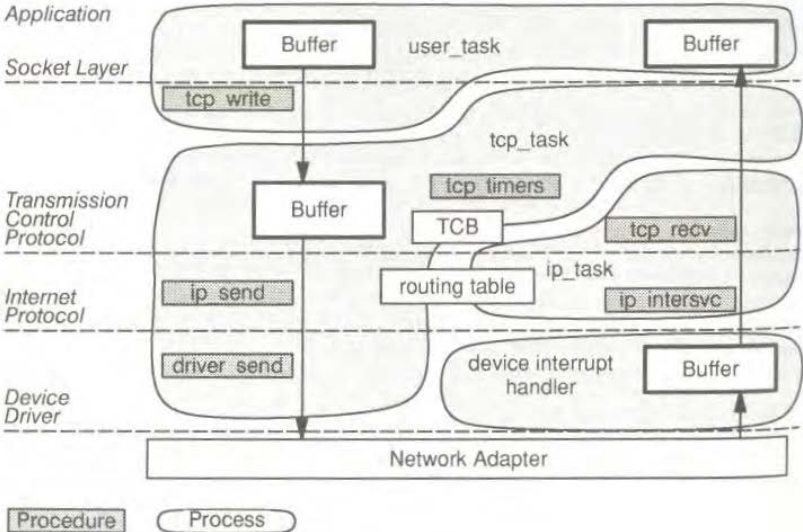
Figure 2. MS/DOS_IP Process Structure

The source code, which served as a basis for this implementation, was already structured into multiple processes that run on top of a simple, non-preemptive multitasking kernel. Figure 2 shows the original split into three processes and one interrupt service routine. Having such a process structure allowed us to stay fairly close to the original source.

As we wanted to execute the IP layer on different processors than the TCP layer, we first isolated the IP relevant functions from both **tcp_task** and **ip_task** into separate processes. Because of the functional division of the PPE into a transmit and receive side, we then split the remainder, i.e. the core of the TCP protocol, of **tcp_task** and **ip_task** *vertically* into three processes (**rtask**, **tcp_recv** running on the receive side and **xtask** running on the transmit side). We will describe the functions of the various protocol processes, that implement IP, TCP and the socket layer in turn. Figure 3 shows the high-level process structure we derived for our implementation.
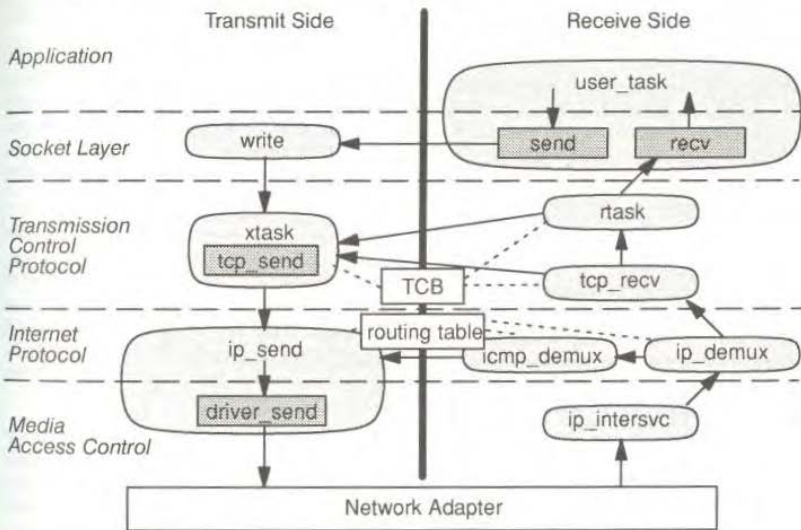
Figure 3. High-Level Process Structure

In the following we present our parallel solution in a top-down approach, first showing the high-level process graph of the main processes in our implementation. These processes have access to data shared between the transmit and receive side and can interact with one another via high-level primitives such as remote procedure calls (RPC) and queues. In a second step, we will then show how these services, in particular shared data between the receive and transmit side as well as RPCs from the receive to the transmit side, have been realized on the PPE.

4.1 IP and ICMP

Because IP is a datagram protocol, the normal flow of data through IP in an end-system requires no interaction between the receiving and transmitting part. Routing information and exception handling, however, require a data exchange. The handling of exception and control messages is the function of ICMP. We therefore partitioned IP into two independent processes **icmp_demux** and **ip_demux**. To guarantee the timely handling of incoming packets, we dedicated a separate process on the receive side of the PPE to the handling of the physical network interface.

The routing table is shared between both processes on the transmit and receive side of the PPE. An RPC is used if **icmp_demux** needs to send out an ICMP message.

4.2 TCP

Splitting the PPE hardware into a separate send and receive side had more impact on how we had to deal with TCP, the socket layer, and application layer, than it had on IP.

We decided to split the finite state machine (FSM) responsible for implementing a TCP connection into two separate FSMs once the connection is in the data phase. The actions of these FSMs are implemented on the receive side through two processes, **rtask** and **tcp_recv**. On the transmit side one process **xtask** implements the FSM. Owing to the duplex nature of TCP and the piggybacking of control information in data packets, these processes need to share the protocol's send and receive state variables maintained in the *transmission control block* (TCB).

tcp_recv demultiplexes incoming TCP segments, locates the appropriate TCB and executes the required action for the FSM state. Header prediction is used to speed up packet handling for packets arriving consecutively on the same connection. Correctly received segments are appended to the receive queue and the application process waiting on this connection is then woken up to move the data to its own buffers. When the received data exceeds the *acknowledgement threshold*, which is specified as a percentage of the advertised receive window, **tcp_recv** makes an RPC to the transmit side to generate an acknowledgement. The acknowledgement is sent as a separate packet, unless this information can be piggybacked onto an outgoing data segment.

rtask is driven by two timers, one responsible for *delayed* acknowledgements, the other for *keep-alive* messages. In steady state data transmission, **rtask** should never generate an acknowledgement, as **tcp_recv** already generates acknowledgements while data are received. Only when the timer runs out and new unacknowledged data have been received since the last acknowledgement will **rtask** generate an acknowledgement. Similarly, keep-alive messages are also sent only when no activity has taken place on a connection for some time. Again, both acknowledgements and keep-alive messages are generated via RPCs to the transmit side.

On the transmit side the process **xtask** manages the transmit queue and the retransmission timers. To send data, **xtask** creates the TCP header and fills in the necessary information from the TCB, such as addresses and sequence numbers for the data and acknowledgements. The header and a pointer to the data are then passed to the IP process (procedure **ip_send**), which embeds this information into an IP datagram.

4.3 Socket Layer and Application

To facilitate our experiments with TCP/IP, we decided as a first step to implement the entire socket layer as well as the test application on the PPE. A detailed description of the interactions be-

tween an application on the host system and a protocol on the PPE can be found in [Kaiserswerth 92].

In our implementation, the socket layer, although tightly coupled with TCP, is part of the application process. It is accessed via a procedural interface, used to create a socket, bind an address to it and establish a TCP connection with a remote socket. As the FSM logic to establish connections is also part of **tcp_recv**, we decided to place the socket and the application code on the receive side. Because we wanted to avoid moving data to be sent from the receive side to the transmit side via a transputer link[3], we also allow the application to use buffers on the transmit side of the PPE. When data is to be transmitted, the **send** procedure simply makes an RPC with the buffer address on the transmit side, thus causing the **write** process to copy the data from this application buffer to the TCP send queue. When the application wants to receive data, the **receive** procedure checks the receive queue for this connection and blocks the application process if the queue is empty.

4.4 Low-Level Primitives

Before giving an example of how TCP data segments are sent and received, we describe how we maintain shared TCBs and routing tables on the PPE, which does not have shared memory between its transmit and receive side, and how we realize RPCs from the receive to the transmit side. Figure 4 shows the process graph of the additional processes required to implement these functions.

Transmit Side   Receive Side

◀━━━ dedicated Transputer Link ◁━━━ internal Channel
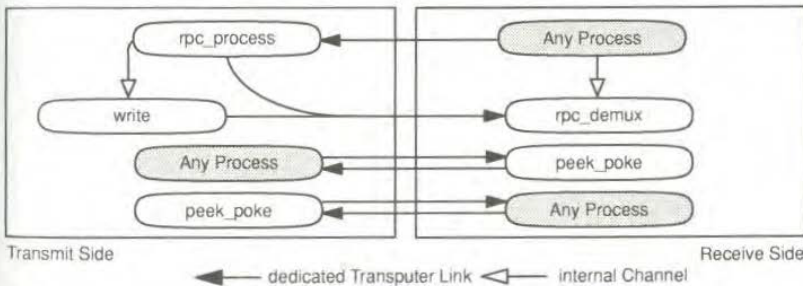
Figure 4. Low-Level Primitives

We implement *distributed shared* memory between the transmit and receive side by placing the data structures that are to be shared at identical physical addresses in the local memories of the two processors which access the data structure. Whenever a value is written onto the local copy of the data structure, the address of the variable and its value are sent via a dedicated transputer link to a server process, **peek_poke**, on the remote side. This process then updates the memory area identified through the address with the accompanying value. The **peek_poke** processes run at high priority to ensure that the exchange of a message with a remote process takes place immediately and is not delayed by scheduling overhead, which would then also delay the remote process because of the transputer's synchronous message passing. Serializing write accesses to the shared data structures is not necessary in our case. Each replicated data structure falls into two parts, one

---

[3] We measured an effective throughput rate of approximately 14 Mb/s across a transputer link, clearly much lower than via our high-speed parallel interface.

written only from the receive side (e.g., the updated transmit window), and the other written only from the transmit side (e.g., the last send sequence number).

Since we do not have a locking protocol for accessing shared data structures, it is possible that for a brief period after the local update and before the remote update has been propagated, the same field in the shared data structure contains two different values. Because of the properties of TCP and the way we have split the protocol onto the transmit and receive side of the PPE, this inconsistency will only be of importance if it is the reason for the protocol state to change. As an example consider the following: assume the retransmission timer (it is also maintained in the TCB) in **xtask** expires and, because the acknowledgement field in the TCB does not indicate reception of an acknowledgement, **xtask** decides to retransmit the unacknowledged TCP segments. On the receive side, however, an acknowledgement has been received in the meantime which makes this retransmission unnecessary[4]. To avoid this problem, before actually going to a retransmit state, **xtask** will reread the acknowledgement field, now however with the value on the receive side, to make sure that a retransmission is warranted. Reading a remote field is similar to writing; a message with the address and size of the variable is sent to the remote **peek_poke** process, which then returns the value of that field.

RPCs from the receive to the transmit side have been implemented as follows: any process on the receive side can format an RPC message, which is then sent via a dedicated transputer link to the **rpc_process**. This process will then execute the remote procedure, or in the case of transmission requests, pass the request via a local (internal) channel to the appropriate **write** process, one of which exists for each TCP connection. Return values are sent, again via a dedicated transputer link, back to the receive side to **rpc_demux**, which forwards these values over a local channel to the process that had initiated the RPC. Upon receiving the return value, the caller becomes ready again and can continue its execution.

4.5 Example

Sending a TCP data segment: The normal data flow is shown in Figure 3. The send data are in a remotely allocated buffer on the transmit side. The application creates a socket and establishes a TCP connection. The socket send call causes an RPC to the remote **write** process which in turn copies the data into the TCP send buffer. **xtask** then controls the transmission and eventual retransmissions of the data. The send procedure builds the TCP segment and forwards the pointer to the segment and the associated control block to **ip_send**. Here the IP header is placed in front of the TCP segment and then the packet is sent to the network. The data is copied twice: first from the application buffer to the send queue in shared memory and from there to the network.

Receiving a TCP data segment: Upon receipt the data is also copied twice: first from the network to the receive queue and from there to the application buffer. The interrupt handler process serves the physical interface and forwards pointers to received datagrams to **ip_demux**, which checks the header and forwards the packet depending on its type to **tcp_recv** or **icmp_demux**.

tcp_recv analyzes the TCP header and calls the appropriate handler function for a given protocol state. To send an acknowledgement or a control packet, **tcp_recv** uses RPCs to the transmit side. Correctly received segments are appended to the receive queue. **rtask** wakes up the application process which is blocked in the socket receive procedure. This procedure then fills the user buffer with data from the receive queue.

---

[4] Note: the logic of the protocol would allow for a retransmission in any case.

4.6 Configuration

On each side of the PPE only one of the two processors is physically able to control the interface to the network. Thus we placed the device driver and the IP layer processes on those two processors. TCP, the socket layer, and the application execute on the second processor on each side of the PPE.

4.7 Memory Management

The buffer memory in the protocols and the socket is managed in an *mbuf*-like linked list. There is only one buffer size to simplify these functions. The buffer size determines the maximum TCP segment size. Provided there is sufficient physical memory (up to 4 MB on each side of the PPE) large fixed-sized buffers help avoid costly memory management functions. Buffer queues and the free buffer list are protected by semaphores to serialize access to the data structures from different processes on the same processor.

The data and control flow in the PPE is organized such that only one processor requests buffers and the other only releases buffers. We ensure that one buffer element always remains in the queue, thus one processor can always append to the end of the buffer queue and the other processor can consume the first element without requiring any additional queue access protocol between the two processors.

5. PERFORMANCE

5.1 Test Setup

To measure the performance of the TCP/IP implementation we used a simple test driver running on the PPE: a source process on one system that sends data over a socket and TCP/IP to a sink process on the other system which receives the data. The setup is shown in Figure 5.
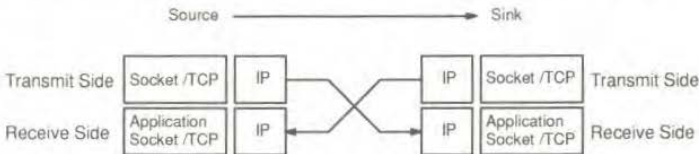
Figure 5. Test Environment

As the final goal of this work is to offload protocol processing from the workstation to the adapter, we examined the following two scenarios:

Scenario 1: The complete socket layer is implemented on the subsystem. Upon receipt, a contiguous block of data is copied from the socket layer to the test application. It is gathered from the linked list of buffers that holds the received TCP segments. For sending, the write process copies the data from the application's buffer to the send queue. The application allocates and controls its buffers on the receive and the transmit side of the PPE. In this scenario we can measure the throughput between sockets on two PPEs.

Scenario 2: In the socket interface all copy calls are replaced by a null call. The test application is merely used to drive the socket and the TCP/IP implementation. This scenario is used to evaluate

the possible performance in case the socket-based application programming interface (API) were implemented on the workstation. The socket layer would then be split into two parts. The upper half resides in the workstation. Calls to the API result in control flows to and from the lower half of the socket layer, which runs on the PPE. Copying data to and from the TCP layer must be done by the workstation processor, because the current PPE only functions as a bus slave. Therefore the copy operations in the socket layer can be combined with the copy between the workstation and the PPE. In this scenario we measure the throughput between the lower half of the socket layer on two PPEs. The results of scenario 2 provide an upper bound for the expected performance of such an integrated system. As such they are valid if one manages – as shown for our implementation of the ISO 8802.2 Logical Link Control protocol [Kaiserswerth 91] – to fully overlap the copy operations and the exchange of control between the workstation and the PPE with the protocol execution on the PPE.

We did not implement TCP checksumming, because it should really be done in hardware [Lumley 92]. To do software checksum calculation on the transputer would cost 3 µs per 16-bit word. We did, however, implement IP header checksumming

The Zählmonitor 4 (ZM4) [Dauphin 91] monitoring and tracing system was used to record execution traces of the PPE subsystem. ZM4 allows gathering of trace events from multiple processors. These events are timestamped with a global clock operating with a resolution of 100 ns. A powerful toolset [Mohr 91] provides trace analysis and visualization.

5.2 Measurements

Because we wanted to see the effects of pipelining and parallel execution of the protocol, we measured the time spent in the various parts of the device driver, IP, TCP and the socket layer. To judge the performance of our implementation we measured the number of TCP segments the implementation can handle per second. Given the segment size, the expected maximum throughput can easily be calculated.

	µs/Segment	µs/32-bit word
Process (Procedure) on Receiver		
tcp_recv	235	
user_task (socket_recv/copy)	31	0.545
ip_intrsvc	9	
ip_demux	23	
Process (Procedure) on Transmitter		
write	30	0.545
tcp_snd_data	147	
ip_send	23	
driver_send	17	0.27
Access to *Shared* Memory (poke call)	18.6	2.4

Table 1. Measured Execution Times

Table 1 lists the execution times of the major processes of our implementation. We used segments of 4096 bytes in these measurements. The times are reported for the first test scenario. The execution times per segment are approximately 4% lower for the second scenario because of reduced contention for accesses to the shared memory. The times per 32-bit word for **user_task** and **write**

are the duration of a 32-bit copy operation from shared memory to shared memory. If we assume that for every n segments an acknowledgement must be sent, we can use the following formula to calculate the execution time per TCP segment in a single processor implementation:

$$T = \max\left[\left[\sum_{i \in [receive\ procs]} t_i + \frac{1}{n}\sum_{i \in [transmit\ procs]} t_i\right], \left[\frac{1}{n}\sum_{i \in [receive\ procs]} t_i + \sum_{i \in [transmit\ procs]} t_i\right]\right]$$

The calculated worst case throughput for $n=2$ is 2150 segments/s in the second scenario.

To calculate the expected performance of our implementation in TCP segments/s from the numbers reported in Table 1 for scenario 1, we need to consider the sequence in which the processes execute on the various processors of the PPE.
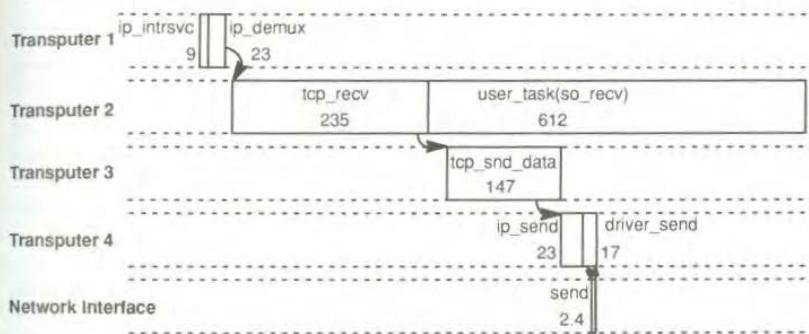
Figure 6. Pipelined Protocol Execution on Sink (Scenario 1) , Times in μs

Figure 6, for example, shows the pipelined handling of an incoming packet and the sending of the acknowledgement which occurs because the acknowledgement threshold has been reached. The time spent in **tcp_recv** is 235 μs. The processing in **user_task** (612 μs) dominates the sending of the acknowledgement. These two processes sum up to 847 μs and form the most expensive pipeline stage, which limits the expected throughput to 1180 TCP segments/s. On the sender **xtask**, **tcp_snd_data**, and **write** add up to 751 μs[5] in the limiting stage of the execution pipeline. The transmitter is therefore expected to be able to send 1331 segments/s. The receiver is thus the bottleneck in our implementation.

We verified these performance predictions by measuring the actual throughput for a transfer of several MB between our two test systems. We obtained a somewhat lower throughput of 1100 segments/s. The difference between the measurement and the expected 1180 segments/s is due to the slow-start algorithm and the eventual retransmission of lost packets, and to process scheduling overhead not captured in our measurements.

---

[5] 16 μs (**xtask**) + 147 μs (**tcp_snd_data**) + 30 μs (**write**) + 1024 × 0.545 μs (copying 4096 bytes in **write**) = 751 μs

For scenario 2 one can again apply the same model to analyze the expected performance. For example, in Figure 7 we show the process execution on the sending PPE.

Transputer 3 | xtask 16 | tcp_snd_data 147 | write 30 | xtask 16 | tcp_snd_data 147

Transputer 4 | ip_send 23 | driver_send 17 + 274

Network Interface | send (4096 bytes) 274

Figure 7. Send Pipelined on Source (Scenario 2), Times in μs

Depending on the segment size, either the TCP layer, which costs 163 μs, or the IP layer[6] is the limiting factor of our execution pipeline. In this example the segment size is 4096 bytes. Therefore IP and the network interface are the limiting pipeline stage. **ip_send** and **driver_send** make up 314 μs, which leads to an expected throughput of 3185 TCP segments/s. The time spent in **driver_send** is the sum of the time to control the transmission process, i.e., to set up the DMA transfer and wait for its completion (17 μs) and the actual transmission time, which depends on the network speed. At 120 Mb/s the transmission of a 32-bit word takes 0.27 μs. When sending 4096 bytes the total time spent in **driver_send** adds up to 291 μs.

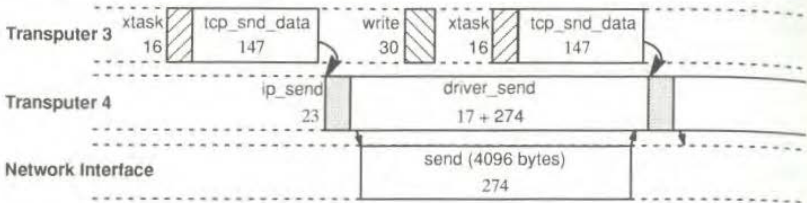For the receiver, TCP handling remains the same, but the execution time of **user_task** drops to 31 μs because we do not copy the received data on the PPE. **tcp_recv** and **user_task** together add up to 267 μs, which means an expected performance of 3745 segments/s. Given the transmission rate of 120 Mb/s the sender is the bottleneck in this case. The actual performance we measured was 3089 segments/s. The difference from the predicted performance is due again to process scheduling overhead not captured in our measurements.

	Scenario 1 Segments/s	Scenario 2 Segments/s
Calculated Throughput Of Receiver	1180	3745
Calculated Throughput Of Sender	1331	3185
Measured Throughput	1100	3089

Table 2. Measured and Calculated Throughput for Both Scenarios Assuming a Segment Size of 4096 Bytes

Table 2 summarizes the predicted and measured performance of our implementation and Figure 8 compares the performance of the two scenarios for different segment sizes.

For unidirectional transmission one observes a speedup of 1.4 when running on four processors instead of one. The reason is uneven load balancing. IP and TCP run on two processors each. As the processing requirements of TCP are much higher than those of IP (8.3 to 1 for reception and 4.4 to 1 for transmission) two of the four processors are only very lightly loaded. The parallelism between the sending and the receiving side, however, can be fully utilized in duplex transmission.

---

[6] The time spent in the IP layer is 23 μs + 17 μs + 0.27 μs × packet size (in 32-bit words)

The speedup is 2.15, because the sending and receiving sides of the PPE are equally loaded. An optimal speedup of 1.7 could be expected for the TCP/IP implementation when only two processors, one for the transmit and one for the receive side of the PPE, were used in full duplex transmission.
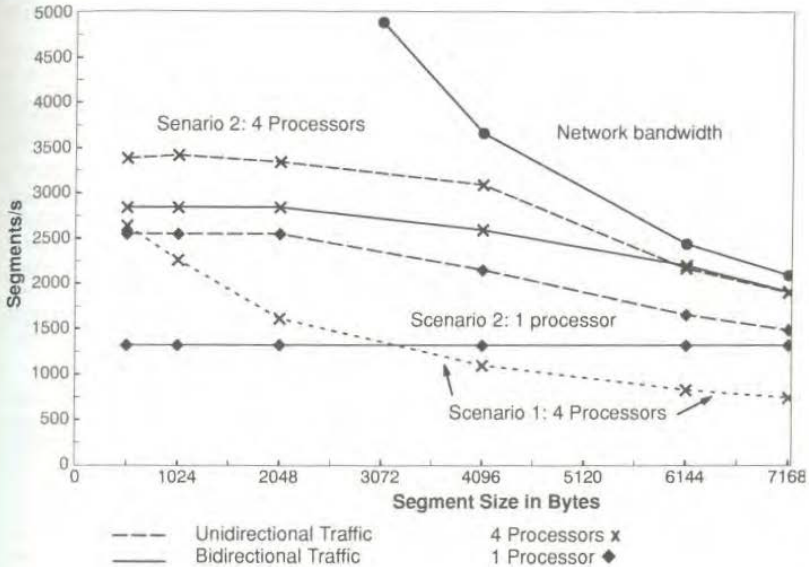
Figure 8. Performance in Segments/s of the Two Scenarios

The impact of having implemented our distributed shared memory via the **peek_poke** process is the same in absolute numbers in both scenarios, but different compared to the limiting pipeline stage. The **peek_poke** call costs 0.6 µs per byte plus a fixed overhead of 18.6 µs in an unloaded system. A 20-byte call from the receiver only takes 30 µs because the transmitter is immediately ready to handle the call. Handling the distributed shared memory costs about 12% of the 235 µs spent in **tcp_recv**. This overhead on the receiver is important in the second scenario, but it is small compared to the copying costs in the first scenario. For the transmitter the **poke** call induces an elapsed time of 57 µs because of the system load and the context switches involved on the receive processor. Although this is 46% of the **tcp_snd_data**, it has no impact on the throughput, because either **ip_send** and **driver_send** or **tcp_recv** and **user_task** will be the limiting stages.

The prototype PPE interface to the workstation (IBM Risc System/6000) allows a copy throughput of only 33 Mb/s[7]. If the application were to be executed on the workstation, all copying would be done from the workstation's processor and if we assume code similar to the second test scenario running on the PPE, then the limited copy throughput rather than the protocol processing will be the bottleneck and we should expect the performance of the integrated system to be around 30 Mb/s.

6. CONCLUSIONS

Our measurements show that a full implementation of TCP/IP on the PPE can cope with data rates in the range of 100 Mb/s. The throughput is much higher than the bandwidth of our hardware interface to the workstation.

It turns out, however, that using a total of four processors, two for IP and two for TCP offers only very little improvement over a two-processor solution, because of the vastly different processing requirements in the two protocol layers. For full duplex traffic, however, the split onto a receiver and a transmitter processor improves protocol performance by a factor of 1.7. Partitioning protocols to obtain even load and linear speedup is a hard problem, in particular for protocols which clearly were not designed with parallel execution in mind. [Zitterbart 91], for example, reports even poorer speedup factors. With an 8 transputer implementation of the OSI CLNP she only achieves a performance increase 3.73 over the single processor version.

Having used a DOS implementation of TCP/IP as the basis for our parallel implementation was a sound decision. Our implementation runs efficiently, when one compares it with other transport protocol implementations. For example, Zitterbart describes a parallel implementation of OSI TP4 written for a system of 8 transputers which was able to process 460 PDUs/s [Zitterbart 91]. In [Braun 91] a parallel implementation of XTP is described, there the performance is 1330 PDUs/s.

Once new faster processors, such as the 100 MIPS T9000 transputer, become available, the gains for pipelined execution of protocols will have to be reevaluated. While the T9000 will be 10 times as fast as the T425, the delays for interprocessor communication will not have shrunk by the same factor. Therefore the relative overhead for pipelining the protocol execution within a layer and even between layers will grow. We claim, however, that the parallel execution of transmit and receive functions is still a suitable form of parallelism to increase protocol throughput. Distributed shared memory, implemented with transputer links easily allows protocol state information to be shared between the two sides of the adapter and impacts the performance of the transport protocol much less than expected. First evaluations of a new architecture, which is based on two T9000s supported by dedicated hardware for checksumming and extraction of header information, indicate a performance of over 30000 TCP segments/s.

---

[7] The reason why this interface is so slow, is that the clocks on the workstation and the PPE run asynchronously. When arbitrating an access from the Micro Channel to the shared memory on the PPE we are forced to use the Micro Channel's *Asynchronous Extended* cycle [IBM 90] of at least 300 ns. This cycle then may even need to be extended by up to 487 ns to match it with the appropriate access cycle of the PPE shared memory. In a new design for the Micro Channel interface this problem would be addressed by buffering in the interface which would allow write-behind and read-ahead. For consecutive accesses, the arbitration cycle for the next word access to the shared memory could then be overlapped with the current word access cycle, thus being able to use regular Micro Channel cycles of 200 ns, and consequently increasing the throughput to more than 50 Mb/s. A busmaster interface using the Micro Channel's streaming mode would allow give higher throughput.

Our measurements are in line with Clark's observation [Clark 89] that the actual protocol processing is not the reason for poor protocol performance. In the PPE, buffer copying and management cost twice as much as the protocol processing. The second scenario shows how throughput can be tripled if the user data were copied by the workstation processor overlapped to the protocol execution on the PPE. In a future design of the PPE, we will concentrate on improving the interface to the shared memory for the protocol processor[8] and the workstation.

We also plan to work on the design of efficient software interfaces between our subsystem and the host system. As can be seen from results published for the Nectar CAB and our own work, crossing the software interface between the host processor and the communication subsystem is a costly operation. Many researchers who advocate the offloading of protocol functions into a dedicated subsystem ignore this issue. For our TCP/IP implementation only a host API based on sockets will be acceptable, as this interface has become the de-facto standard. These sockets must be lightweight enough to provide efficient pipelined execution between the communication subsystem and the host processor to exploit the full power of the PPE.

7. REFERENCES

[Arnould 89]   Arnould, E. A., Bitz, F. J., Cooper, E. C., Kung, H. T., Sansom, R. D., Steenkiste, P. A., The Design of Nectar: A Network Backplane for Heterogeneous Multicomputers, Proceedings of ASPLOS-III, pp 205-216, April 1989.

[Braun 91]   Braun, T., Zitterbart, M., A Parallel Implementation of XTP on Transputers, Proc. 16th Annual Conf. on Local Computer Networks, Minneapolis, Oct. 1991.

[Chesson 87]   Chesson, G., The Protocol Engine Project, Unix Review, Vol.5 No.9, Sept. 1987, pp.70-77.

[Cheriton 88]   Cheriton, D.R., VMTP: Versatile Message Transaction Protocol – Protocol Specification. Network Working Group, Request For Comments, RFC 1045, February 1988.

[Clark 89]   Clark,D. Lambert, M.L., Romkey, J., Salwen, H., An Analysis of the TCP Processing Overhead. IEEE Communications Magazine, Vol. 27, No. 6 (June 1989), pp. 23-29.

[Clark 90]   Clark, D., Tennenhouse, D., Architectural Considerations for a New Generation of Protocols. Proceedings of the SIGCOMM'10 Symposium, Sept 1990, pp. 200-208.

[Dauphin 91]   Dauphin, P., Hofmann, R., Klar, R., Mohr, B., Quick, A.,Siegle, M., Soetz, F., ZM4/SIMPLE: A General Approach to Performance-Measurement and -Evaluation of Distributed Systems. Technical Report 1/91, Erlangen, January 1991.

[Hoare 78]   Hoare, C.A.R., Communicating Sequential Processes. Communications of the ACM,Vol.21, No 8, August 1978, pp. 666-677.

---

[8] In the PPE a shared memory cycle of the transputer is twice a local memory cycle

[IBM 90]   IBM RISC System/6000 POWERstation and POWERserver Hardware
                    Technical Reference – Micro Channel Architecture, 1990.

[INMOS 89]   Inmos Limited, The Transputer Databook. First Ed. 1989, Document
                    No. 72 TRN 20300, pp. 23-43 and 113-179.

[Kaiserswerth 91]   Kaiserswerth, M., A Parallel Implementation of the ISO 8802.2-2 LLC
                    Protocol, IEEE Tricomm '91 – Communications for Distributed
                    Applications and Systems, Chapel Hill NC, April 17-19, 1991.

[Kaiserswerth 92]   Kaiserswerth, M., The Parallel Protocol Engine, IBM Research Report,
                    RZ 2298 (#77818), March 1992.

[Kanakia 88]   Kanakia, H., Cheriton, D.R., The VMP Network Adapter Board (NAB):
                    High Performance Network Communication on Multiprocessors, ACM
                    SIGCOMM 88, pp. 175-187.

[Lumley 92]   Lumley, J., A High-Throughput Network Interface to a RISC
                    Workstation, Proceedings of the IEEE Workshop on the Architecture
                    and Implementation of High Performances Communication Subsystems,
                    Tucson, AZ, Feb. 17-19, 1992.

[LS-C 89]   Logical Systems, Transputer Toolset, Version 88.4 Feb. 1989.

[Mohr 91]   Mohr, B., SIMPLE: A Performance Evaluation Tool Environment for
                    Parallel and Distributed Systems, in A. Bode, Editor, Distributed
                    Memory Computing, 2nd European Conference, EDMCC2, pp. 80-89,
                    Munich, Germany, April 1991, Springer Verlag Berlin LNCS 487.

[PEI 92]   Protocol Engines Incorporated, XTP Protocol Definition, Revision 3.6.,
                    Edited by Protocol Engines Mountain View, CA, January 11, 1992.

[Pountain 88]   Pountain, D., May, D., A Tutorial on OCCAM2, BSP Professional
                    Books London 1988.

[UM 90]   IBM Corporation, University of Maryland. Network Communications
                    Package. Milford 1990.

[Wicki 90]   Wicki, T., A Multiprocessor -Based Controller Architecture for
                    High-Speed Communication Protocol Processing, Doctoral Thesis, IBM
                    Research Report, RZ 2053 (#72078), Vol 6, 1990.

[Zitterbart 91]   Zitterbart, M., Funktionsbezogene Parallelität in transportorientierten
                    Kommunikationsprotokollen, Dissertation, VDI-Reihe 10 Nr. 183,
                    Düsseldorf: VDI-Verlag 1991.