Parallelism and Performance in Communication Subsystems

Shu-Ping Chang, Ahmed Tantawy and Hanafy Meleis

IBM Research Division
Thomas J. Watson Research Center
P.O. Box 704, Yorktown Heights, NY 10598, USA

Abstract

This paper discusses the impact of various design and implementation factors on the performance of a communication subsystem. Measurements obtained from two different software implementations of the same protocol (ISO 8802-2 Logical Link Control Protocol Type 2) over three different experimental hardware platforms are used as a basis for comparison. The impact of exploiting parallelism is especially emphasized. The effects of software structure and optimized hand coding are also discussed. Although some results pertain to only one specific protocol, similarities with other connection-oriented protocols can be drawn.

1. Introduction

The major challenge in high speed networking has shifted from the invention of faster Media Access Control (MAC) protocols to solving the more crucial problems related to the design and implementation of high performance communication subsystems, capable of handling the ever increasing bandwidth of the physical networks. This issue is being addressed by many researchers, having different perception of the real problem. Some view the protocols themselves as the real bottlenecks and suggest the use of simpler "lightweight" protocol stacks [1]. The majority, however, believes that the problem lies in the architecture and implementation of the communication subsystems in general. We do not fully adhere to the first group because many laboratory measurements (e.g., [2,3,4]) have shown that the processing of the protocol state machine represents only a small fraction of the total pathlength needed to process a data packet in the communication subsystem. The "weight" is mainly in the interface between the host system and the communication subsystem. The architecture of the hardware platform and the software structure used in the core communication subsystem are the most important factors affecting the performance of such systems.

Most current work stresses the importance of parallelism in the implementation of high performance communication systems [5,6]. This is viewed as the only way to hide the growing mismatch in speed between processors and transmission links. There is, however, a number of approaches to exploit parallelism in the design of more efficient communication subsystems. These include:

1. using a pipeline architecture to implement the sequential stages of packet processing (e.g., one processor per layer),
2. using a multiprocessing platform to execute several functions simultaneously on the same packet, and
3. using a set of coprocessors (or hardware assists) to implement time critical and/or time consuming functions and, therefore, off-loading the main protocol processor.

In section 2 of this paper, we present three experimental prototype FDDI network adapters for Microchannel-based systems (e.g., IBM PS/2). The first prototype has a simple uniprocessor architecture while the two others use a dual-processor architecture. In section 3, software architectures are described. Clearly, each hardware platform requires a different software "kernel" to support the implementation of protocol state machines. We also briefly

0-8186-2755-7/92 $03.00 © 1992 IEEE

describe two different implementations of the same protocol, i.e., ISO 8802-2 Logical Link Control (LLC) [7]. In section 4, we present the effect of some major design and implementation elements on the overall performance of the system. Namely, we discuss the effect of data movement, software coding and the multiprocessing approach used.

2. Experimental Hardware Platforms

2.1 Platform 1: Simple Uniprocessor Architecture

Figure 1 shows Platform 1. The adapter has two interfaces: one to the FDDI network and the other to the Microchannel bus of the host system. A general purpose microprocessor (e.g., an Intel 80386) is used to run all the necessary software. A buffer memory is used to store all data packets, i.e., packets awaiting transmission on the network, unacknowledged transmitted packets, and packets received and not fully processed and transmitted to the host system. All components communicate through a 32-bit wide adapter bus. The processor receives data packets and other control messages (e.g., for flow control) from either interface and processes them to take proper action (i.e., transmits a packet or a control message to either interface or, simply, updates the state of the protocol state machine). Only then, the processor becomes ready to receive the next packet.

2.2 Platform 2: Dual-processor (Master-Slave) Architecture

Figure 2 represents a variation of Platform 1, in which the main adapter processor handles all data movement operations and controls another processor dedicated to protocol processing. This protocol coprocessor uses a separate dual-port header memory. This modification enables the coprocessor to execute protocol entities as soon as the packet header is received, without having to wait for the complete reception of the entire packet in the buffer memory. This allows the processing of a packet header concurrently with the reception of its data field. The main adapter processor receives all packets from either the host system or the network interface. It routes the proper header to the protocol coprocessor and triggers its operation. When the coprocessor finishes processing the appropriate protocol functions, it notifies the main processor, which then enqueues the packet to its proper destination (i.e., either the network interface or the host system).

2.3 Platform 3: Asynchronous Dual-processor Architecture

Figure 3 shows Platform 3, which is a slight variation of Platform 2, characterized by the use of two communicating processors, operating asynchronously. The first processor, called the Packet Handling Processor, performs all the operations of the main processor of Platform 2 while the second processor, called the Protocol Processor, performs the functions of the protocol coprocessor used in Platform 2. The difference is in the way these two processors interact. In Platform 2, the operation is single-threaded and controlled entirely by the main processor. The coprocessor is used to perform protocol processing while the rest of a packet is received. This is particularly useful when long packets are handled. In case of short packets, the main processor is slowed down because it has to wait for the coprocessor to finish its job before accepting a new packet. There is a fork and a "rendez-vous" point for each packet. In Platform 3, the Packet Handling Processor continues to move packets in and out of the adapter, regardless of the status of the Protocol Processor. When the latter finishes a job, it passes any modified or new header to the Buffer Memory and then takes the next header, if any. All queues of packets and control messages flowing in and out of the adapter through either interface are handled by the Packet Handling Processor.

3. Experimental Software Implementations

3.1 Generalized Software Architecture

In general, the software running on each system comprises four components:

1) The Kernel, which is a simple and customized operating system for the adapter. It provides certain basic functions, such as memory management, timer management, task

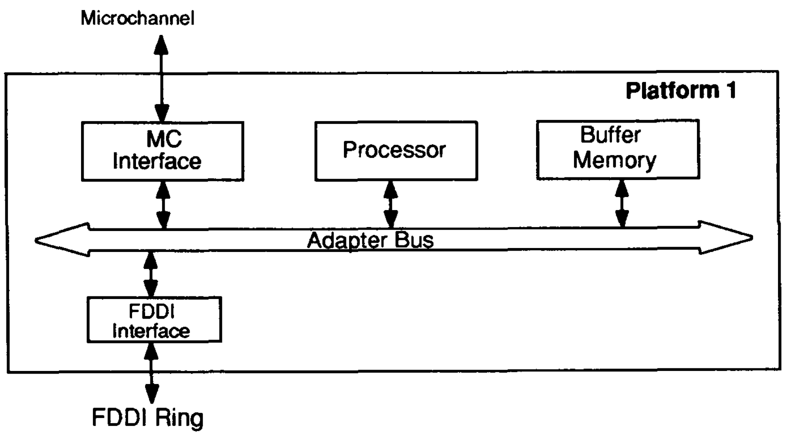

Fig. 1: Experimental Platform 1

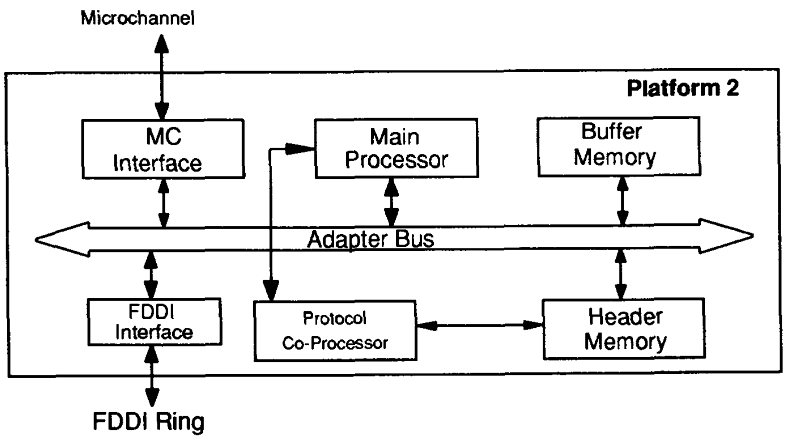

Fig. 2: Experimental Platform 2

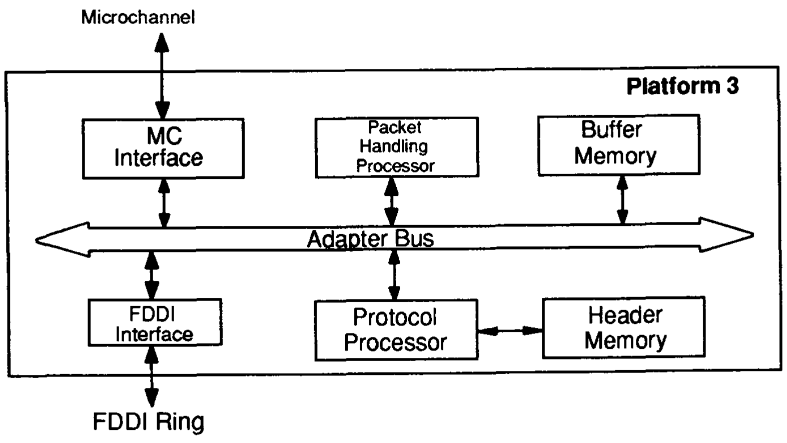

Fig. 3: Experimental Platform 3

switching and data transfer among the various components of the platform.

2) The Protocol Processing Components, which are the embodiments of the protocol state machines, as described in the protocol specification documents.

3) The Interlayer Communication Component, which is responsible for handling interlayer data units as well as messages, such as flow control between adjacent layers. This component is not specified by protocol standards and its implementation depends solely on the implementors specifications and choices. For example, one might choose to restrict this function to the exchange of messages and data units between adjacent layer components. The additional function of interlayer flow control is, however, useful if the implementation is to be robust, so that no messages or data units are to be lost due to saturation in the buffer space of a given layer. Furthermore, the mechanism used for interlayer flow control can significantly affect the performance of the communication subsystem. For these reasons, this component is not considered in the performance comparisons given in this paper.

4) The Auxiliary Components, which include testing and monitoring facilities, such as trace, debugging tools and error logging. This component will be ignored in the rest of this paper because its effect on the performance depends on its functionality and its status (enabled or disabled).

3.2 Implementation of the Software Kernel

The kernel can be either integrated or layered. In a first implementation, our layered kernel consists of a full-size operating system for embedded control applications augmented with a Run Time Environment (RTE) that offers the interface and functionality required by the protocol processing and interlayer communication components. In a second implementation (for Platform 3), we developed an integrated kernel providing the basic functional interfaces required by the other entities, in a much more efficient and simpler way.

3.3 Implementation of the Protocol State Machine

The protocol state machine corresponding to ISO-LLC Type 2 (connection oriented) is implemented in two different ways. First it is developed using PASS [8], which is a set of automated tools that generate C code directly from a specification of state transitions, using a special formalism. The second version is an optimized hand coded C program with a few assembly language routines.

4. Performance Effect of Some Design and Implementation Factors

The effect of some design and implementation factors is investigated. The performance measure used is the total pathlength required to process the data packets received from and transmitted to the network. The inverse of this value gives the number of packets that can be handled per time unit. The pathlength considered for measurements is the "optimal" one, i.e., the pathlength corresponding to the normal case of operation, thus discarding all the effect of transmission and buffering errors. This does not mean that tests for error conditions are discarded, it simply means that tests are considered to lead always to positive (regular) results. This is a reasonable assumption, given the realizability of large buffer sizes, the reliability of current physical networks and the robustness of protocols and their implementations.

The performance of the system is measured in machine cycles to exclude, as much as possible, the effect of the power of the processor used. It should be noted, however, that the same C language program may yield different pathlengths on different processors. Moreover, the processor chosen for a particular implementation may considerably affect the pathlength because any given machine operation might require different numbers of execution cycles on different processors. For these reasons, we believe that the measurements given here should be considered as trend indicators and only their relative values have a reasonably general significance.

4.1 Software Coding and Structure

To study the effect of software coding, we compared the pathlength of the two implementations of the protocol state machines mentioned in section 3.3. The hand coded version yielded pathlengths of 1500 for received packets and 750 for outgoing packets, in comparison with 2560 and 1400, respectively, for the automatically generated version. The advantage of quick code generation is outbalanced by a significant decrease of performance (surprisingly, the ratio is the same: approximately 1:2).

Given that the procedure call overheads may typically reach 15% of the total pathlength [4], proper programming style is greatly welcome. Also, since more than one protocol entity may typically be processed on the same processor, priority and interrupt mechanisms as well as efficient switching from one protocol entity to another are highly required.

4.2 Memory Organization

The management of the data structures used in packet handling may consume a few hundred cycles per packet. Special consideration must be given to the way packets are stored in memory. Consequently, the tradeoff between memory utilization and performance becomes an important design decision.

Storing a packet in a buffer that has exactly its size optimally utilizes the memory but requires an almost unfeasible memory management procedure. Packets need to be removed and the space that they utilize must be reused. Obviously, one cannot guarantee that a new packet will perfectly fit in the same space used by a previous packet.

On the other extreme, one might consider partitioning the memory into fixed size buffers. That size should accommodate the largest possible packet expected from the physical network (e.g., 4500 bytes in case of FDDI). This makes packet buffering and buffer release a very simple job but the utilization of memory space will be far from optimal.

Intermediate solutions are also possible. Instead of using the maximal packet size as the basis for partitioning the memory, one may choose smaller partitions (e.g., 256 bytes). Each packet would then use (or release) a number of partitions. Memory management here is much simpler than the "exact fit" allocation scheme but more elaborate than the "maximum size" allocation scheme. However, the processor has to maintain a linked list for every packet, in addition to the linked list (queue) of packets. Every time a partition is needed to store a segment of a packet, about 10 machine instructions are needed to maintain all the lists and their attributes, thus adding an extra 15% to the data movement load.

Given the bimodal distribution of packet sizes in typical networks [9,10], it is reasonable to consider a limited number of (e.g., two) distinct partition sizes (e.g., small: 256 bytes and large: 4 Kbytes). An FDDI packet is therefore given a small partition, followed by a large one, followed by another small one, if needed. This considerably limits the number of times a partition is needed, especially for large packets. Therefore, memory management overhead is reduced, while using a statistically sound memory saving approach.

Another related issue is the possibility that packet lengths may be modified after their processing, due to the addition or deletion of headers and trailers. To avoid crossing the partition boundary when this highly probable event occurs, pre-reservation of expansion space is required in, at least, the first partition.

4.3 Data Movement

To study the effect of data movement among the adapter components, let us compare Platforms 1 and 2. The total pathlength for Platform 1 (excluding the management of packet buffering) is
$C_1 = 1600 + 2L + P$,
where L is the packet length in bytes and P is the protocol processing time. The value of P depends on the way the protocol is implemented, as discussed in section 4.1. It is important to mention here that, according to the formula given above, the P/C ratio varies between 45% for short packets and 15% for

long FDDI packets. This illustrates the primary importance of data movements as opposed to the processing of the protocol state machine.

For Platform 2, the value of C_2 is approximately calculated by:
$C_2 = 1060 + L + \max [(350 + L), (250 + P)]$

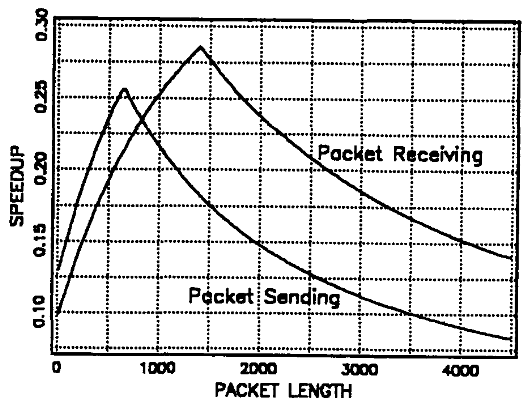

Fig. 4 Speedup due to parallelism between data moves and protocol processing

Figure 4 depicts the relative speedup in total pathlength between Platforms 2 and 1 for the hand coded version of the protocol state machine in case of received as well as transmitted packets.

4.4 Interaction between Processors

The overhead resulting from interprocessor communication is highly dependent on the mechanism used for exchanging data and control information between the processors. This involves the way interrupts are used and handled. It also involves the mechanism used to transfer packet headers between the header memory and the buffer memory.

In Platform 2, the processing load per packet (C_2) is distributed among the two processors as follows. The main processor spends M_2 cycles for every packet of length L and the coprocessor spends T_2 cycles, where:

$M2 = 1410 + 2L$
$T_2 = 250 + P.$

In Platform 3, The packet handling processor carries a load M_3 for every packet and the protocol processor carries a load T_3, where
$M3 = 1420 + 2L$
$T_3 = 410 + P.$
One should also add to these values the additional load that each processor incurs for requesting control over the shared adapter bus in order to transfer data and messages. This competition between processors makes it impossible to calculate a deterministic value for C_3. The bus arbitration mechanism (i.e., priorities, preemption, and maximum bus holding time) as well as the actual traffic across the bus become crucial factors in determining the overall performance of the system.

A quick comparison of the loads on the different processors in Platforms 2 and 3 shows that the load on the secondary processor (protocol processing) has increased in Platform 3. This is due to its involvement in moving the header from the buffer memory to its own header memory before processing it and possibly sending back a new header or control data unit. These moves consume approximately 100 cycles, during which the adapter bus is also used by the protocol processor.

The other additional 60 cycles are spent in handling interrupts from the packet handling processor to indicate the status of the queue of packets awaiting processing. The issue of using interrupt versus polling deserves more discussion because it affects the overall performance of the adapter. In Platform 2, we use polling because there is a main processor that controls the entire operation. It can poll a component whenever it needs to know information about it or get data from it. In Platform 3, we use interrupts because the communicating processors run asynchronously, i.e., no one has to frequently wait for the other. They only need to communicate when there is a change in the status of jobs queued for the other. This means that, for example, an interrupt is issued to the protocol processor whenever the queue of packets awaiting processing becomes non-empty after it has been empty or vice versa.

This "toggle" interrupt policy avoids having to interrupt the protocol processor every time a new packet arrives. On the other side, the protocol processor does not have to respond to the interrupt by performing a context switch to receive the new header and, thus, wasting many cycles (about 200 in an i80386). The interrupt handling routine can simply set a flag to indicate the status of the job queue, leaving the main task stream continue its processing in a sequential manner.

Finally, a fair comparison of the two dual processor architectures, i.e., Platforms 2 and 3, must also consider some qualitative criteria, in addition to the quantitative measures. Due to its asynchronous operation, Platform 3 is less prone to losing packets, it allows a generally better utilization of the available processing power and, in return, it is more complex.

5. Conclusion

We have used a few experimental hardware and software prototypes of a high performance communication subsystem to show the effect of some design and implementation choices. The protocol state machine itself is not a crucial factor affecting the overall performance of the subsystem. Instead, data movement and memory organization have the largest effect. Obviously, multiprocessing improves the performance but the degree of improvement highly depends on the allocation of tasks among processors and, more importantly, on the way processors communicate and handle interrupts.

References

[1] W. Doeringer, D. Dykeman, M. Kaiserswerth, B. Meister, H. Rudin and R. Williamson, A Survey of Lightweight Transport Protocols for High Speed Networks, IEEE Transactions on Communications, Vol.38, No.11, Nov. 1990.

[2] L. Svobodova, Measured Performance of Transport Services in LANs, Computer Networks and ISDN Systems, Vol.18, 1989.

[3] D. Clark, V. Jacobson, J. Romkey and H. Salwen, An Analysis of TCP Processing Overhead, IEEE Communications Magazine, June 1989.

[4] A. Tantawy, T. Schuett, H. Meleis, R. LaMaire and R. Auerbach, High Performance Protocol Implementations: LLC Case Study, Protocols for High Speed Networks II, North Holland, 1991.

[5] M. Zitterbart, High Speed Transport Components, IEEE Network Magazine, pp.54-63, Jan. 1991.

[6] H. Meleis and A. Tantawy, High Performance Networking: the Modular Communication Machine (MCM) Approach, IEEE Workshop on FTDCS, Cairo, Egypt, Oct. 1990.

[7] ISO, Standard 8802-2 Logical Link Control (LLC), Dec. 1989.

[8] A. Fleishman, PASS - The Parallel Activity Specification Scheme, IBM Tech. Report No. 43.8715, 1988.

[9] K.M. Khalil, K.Q. Luc, D.V. Wilson, LAN Traffic Analysis and Workload Characterization, 15th Conference on Local Computer Networks, Minneapolis, MN, Oct. 1990.

[10] R. Gusella, A Measurement Study of Diskless Workstation Traffic on an Ethernet, IEEE Transactions on Communications, Vol.38, No.9, 1990.