Optimization Techniques for Parallel Protocol Implementation

Stefan Leue
Institute for Informatics
University of Berne
Länggassstrasse 51
CH-3012 Berne, Switzerland
leue@iam.unibe.ch

Philippe Oechslin
Computer Network Lab LTI
Swiss Federal Institute of Technology
DI-LTI EPFL
CH-1015 Lausanne, Switzerland
oechslin@ltisun.epfl.ch

Abstract

We propose a method for deriving parallel, scheduling optimized protocol implementations from sequential protocol specifications. We start with an SDL specification, identify a common path for optimization and perform a data dependency analysis. The resulting common path graph is parallelized as far as permitted by the data dependency graph. The degree of parallelism is extended even further by deferring data operations to the exit nodes of the common path graph. The resulting parallel operation model is then submitted to a scheduling algorithm yielding an optimized compile-time schedule. An IP based protocol stack with TCP and FTP as upper layers serves as an example.

1 Introduction

The typical quality of service requirements (e. g. transfer delay, throughput rates) for high speed protocols impose strong performance requirements on high speed protocol implementations. As the throughput of networks has increased much faster than the processing power of processors these requirements can only be satisfied by efficient processing of protocol data by the involved protocol machines. Different approaches to improve the performance of communication protocols have been proposed, so f. e. improvements by changes to the protocol mechanisms ([CT90] and [TM92]), by *hardware implementation* of protocol functions ([KS89]), and by *parallelizing* the implementation of communication protocols (BraZit92, [Hei92], [PS92], [RK92] and [TZ93]). These latter papers suggest distributing protocol functions over multiple processors with either dedicated or general purpose functionality, thus an MIMD parallelization. We will focus on this parallelization approach in this paper.

The method we propose takes as input an SDL specification of a protocol stack, some statistical information about the behavior of the protocol and a description of the hardware it will be executed on. Starting with a control dependence graph derived from the specification we determine a subset of this graph called the common path graph in which we next reduce the control dependencies to allow parallelism as far as allowed by data dependencies. Then we group and defer the data manipulation operations to allow Lazy Message Processing. The resulting dependence graph will be mapped on the hardware using a classical scheduling algorithm which will yield a compile-time schedule. The example will be given in Section 2, our method will then be explained in Sections 3 and 4, and we conclude in section 5.

2 The example SDL specification

We base our optimization example on the SDL (see [BHS91]) specification of an IP based protocol stack (see Tables 1 and 2[1]). The system consists of one block `STACK` representing the example protocol stack. `STACK` is substructured into three blocks, each containing the functionality of one protocol layer. The environment represents the lower layer medium, which hands `IP_packet` data units to the stack, and the upper layer FTP-user to which `FTP_data` data units are delivered. The `IP` block mimics the IP protocol entity behavior. After performing certain checks on the `IP_packet.header` field the upper layer protocol is tested. For reasons of conciseness we only consider the case where TCP is the upper layer protocol to which IP conveys the data in the form of an `IP_TCP_SDU`, an alternative upper layer protocol is

[1] The labels at the left margin are not part of the SDL specification, their meaning will be explained later.

```
        SYSTEM STACK
        ... /* type definitions
        BLOCK STACK_block;
        CHANNEL TCP_FTP
          FROM TCP TO FTP WITH TCP_FTP_SDU;
        ENDCHANNEL TCP_FTP;
        CHANNEL IP_TCP
          FROM IP TO TCP WITH IP_TCP_SDU;
        ENDCHANNEL IP_TCP;
        CHANNEL ENV_IP
          FROM ENV TO IP WITH IP_packet;
        ENDCHANNEL ENV_IP;
        CHANNEL FTP_ENV
          FROM FTP TO ENV WITH FTP_data;
        ENDCHANNEL FTP_ENV;

        BLOCK IP; ...
          PROCESS IP_process; ...
          STATE waiting;
*   S1    INPUT(IP_packet);
*   D1    DECISION IP_check_sum(IP_packet.header);
            (FALSE) : CALL error_handling;
               NEXTSTATE waiting;
CP          (TRUE)  :
*   D2      DECISION
              IP_header_check(IP_packet.header);
              (FALSE) : CALL error_handling;
                 NEXTSTATE waiting;
CP            (TRUE)  :
*   D3        DECISION
                IP_test_if_options(IP_packet.header);
CP              (FALSE) :;
        (TRUE) : CALL IP_options_processing;
              ENDDECISION;
BP  D4        DECISION
                IP_test_upper_protocol(IP_packet.header);
                ('TCP') :
*   S2            TASK IP_TCP_SDU :=
                    TCP_SDU_compile(IP_packet);
*   S3            OUTPUT(IP_TCP_SDU);
                  NEXTSTATE waiting;
                ('UDP') : ...; /* UDP handling */
                (ELSE ) : ...;
                         /* other protocol handling */
              ENDDECISION;
            ENDDECISION;
          ENDDECISION;
          ENDPROCESS IP_process;
        ENDBLOCK IP;
```

Table 1: Part 1 of the SDL-PR specification

```
         BLOCK TCP; ...
          PROCESS TCP_process; ...
          STATE waiting ;
*   S4    INPUT(IP_TCP_SDU);
*   D5    DECISION TCP_check_sum(IP_TCP_SDU);
CP          (TRUE)  :;
            (FALSE) : NEXTSTATE waiting;
          ENDDECISION;
*   D6    DECISION
            connection_state[
              IP_TCP_SDU.TCP_packet.header.TCP_destination]
            ('established') :
*   D7        DECISION
                TCP_test_flags(
                  IP_TCP_SDU.TCP_packet.header);
CP              ('normal')  :
*   S5            call TCP_normal_operations(
                    IP_TCP_SDU.TCP_packet.header);
                (ELSE) : call TCP_exception_handling;
                         NEXTSTATE waiting;
              ENDDECISION;
*   D8        DECISION TCP_seqno_ok(
                         IP_TCP_SDU.TCP_packet.header);
CP              (TRUE)  :
*   D9            DECISION TCP_test_upper_application(
                    IP_TCP_SDU.TCP_packet.header);
                    ('FTP') :
*   S6                 TCP_FTP_SDU :=
                         IP_TCP_SDU.TCP_packet.data;
*   S7                 OUTPUT(TCP_FTP_SDU);
                       NEXTSTATE waiting;
                    ('TELNET') :
                       /* appropriate TELNET handling */
                       NEXTSTATE waiting;
                  ENDDECISION;
                (FALSE) :
                  CALL seqno_error_handling;
                    NEXTSTATE waiting;
              ENDDECISION;
            (ELSE) : ...; /* handling other states */
          ENDDECISION;
          ENDPROCESS TCP_process;
        ENDBLOCK(TCP);

        BLOCK FTP; ...
          PROCESS FTP_process; ...
          STATE ascii_transfer;
*   S8    INPUT(TCP_FTP_SDU);
*   S9    FTP_data := translate(TCP_FTP_SDU);
*   D10   DECISION test_eof(FTP_data);
            (TRUE)  : OUTPUT(FTP_data);
                      NEXTSTATE closing;
*   S10     (FALSE) : OUTPUT(FTP_data);
                      NEXTSTATE ascii_transfer;
          ENDDECISION;
          ENDPROCESS FTP_process;
        ENDBLOCK FTP;
        ENDBLOCK STACK_block;
```

Table 2: Part 2 of the SDL-PR specification

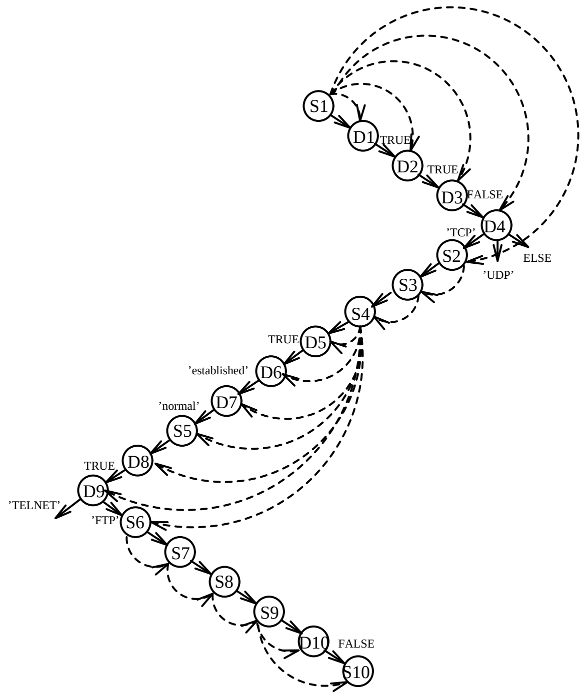

Figure 1: Common path graph with control (solid arrow) and data (dashed arrow) dependency relations

UDP. The TCP process checks the validity of the data, whether the relevant connection is established, and whether flags are set in the `TCP_packet` header field in which case exception handling is carried out. Otherwise, normal TCP operations will be performed, the sequence number will be checked, and if this check is successful the `TCP_packet.data` field will be conveyed to FTP in case FTP is the upper layer protocol by an `TCP_FTP_SDU`. FTP translates the data to *internal ASCII* representation and conveys it to the user.

3 Parallelization and Optimization

Decomposition into operations. At first we decompose the specification of the protocol stack into operations. An operation is either a single or grouped data access *statement*, or a *decision* that leads to branching. An iteration loop defined over the elements of a set of data (e. g. checksum calculation) is defined as one operation. The control dependencies between the statements define a *control dependency graph* consisting of statement and decision nodes, one graph for every SDL process.

Common Path analysis. Based on a stochastic protocol analysis we distinguish the branches (outgoing edges of a decision node) of the dependency graph into those which are taken with *stochastic certainty* (the *common* ones) and those for which the probability is below a certain statistical confidence value (the *uncommon* ones). This is a generalization of the *Common Path* optimization in [CJRS89]. It defines a *common path graph* which is a subgraph of the control dependency graph in every SDL process. Our further optimization will only address the common way a packet takes through the protocol stack, along a common path, and not the uncommon cases. We drop the uncommon branches from every decision node. We call those decision nodes for which there remain multiple outgoing branches *branching nodes*. They enclose non-branching segments on the common path graph. Classic cases of decision nodes that have *only* common branches are nodes representing error checking or connection opening or closing, branching nodes often handle multiplexing or selection of an upper layer protocol. To facilitate the presentation of our method we assume that the code we analyze contains no loops and no recursive procedure calls which implies that the resulting common path graph is acyclic.

Composing the common path graphs. We now combine the common path graphs of the SDL processes to form a monolithic common path graph for the entire protocol stack. We start the construction of the graph at that node at which the data unit whose processing we consider is read from the medium. In SDL this is indicated by an `INPUT` from the environment, we call the corresponding node *root node*. We compose two graphs at those nodes where the first process conveys a data unit by means of an `OUTPUT` statement to an adjacent layer process, which in turn receives it by an `INPUT` statement. We consider the `OUTPUT` and `INPUT` statements simply to define synchronous operations. Furthermore we abandon at this stage of transformation the fact that in SDL the specified processes run independently and concurrently, only synchronizing by asynchronous communication over infinite process-unique input queues. The process we obtain by this composition forms a monolithic process space with one global data space and a unique process control token. The composition ends when an SDL process outputs a data unit to the environment, we call the respective nodes *exit* nodes. Again, for simplicity we assume that the code of the SDL specification is structured so that the resulting structure, which we call the monolithic *common path graph*, is acyclic.

Figure 1 presents a monolithic common path graph obtained by transforming the SDL specification example in Tables 1 and 2. In the SDL specification we have marked the statements which lie on the common path by asterisks. Non-decision statements are annotated by `Sn`, where `n` is an enumeration, and decision statements are annotated by `Dn`. Common branches of decisions are annotated with `CP` and uncommon ones with `BP`. Figure 1 presents the control dependencies, indicated by solid line arrows, which form the monolithic common path graph after composing the individual common path graphs of the processes. `S1` is the root node. `D4` and `D9` are branching points. For reasons of conciseness we do not consider the entire common path graph and omit the subgraphs starting with branches `'UDP'` and `'TELNET'`. `S10` is the exit node for the common path graph we consider.

Data dependency analysis. Based on the common path graph we analyze data dependencies between the nodes by classical data flow analysis. The data dependency graph constitutes a criterion for the maximum possible parallelization of the protocol because two operations may only be executed in parallel if they are not data dependent.

The statements in the specification of the protocol stack can be classified as follows. An *assignment* statement[2] S which assigns a value to a variable v is a *define* statement with respect to v, and a *use* statement with respect to variable w if w appears on the right hand side of the assignment. A *procedure* call is a *define* statement with respect to a result parameter and a *use* statement with respect to a non-result parameter. A *decision* statement with a data parameter v is a *use* statement with respect to v. An `OUTPUT(v)` statement is a *use* statement with respect to v. An `INPUT(v)` statement is a *define* statement with respect to v. The dependencies between statements can then be determined as follows. A statement S_j is *data dependent* on a statement S_i iff there is a variables v so that a) S_i is a define and S_j is a use statement with regard to v, b) S_i precedes S_j in the sequential control flow and S_i is a use and S_j is a define statement with regard to v, or c) S_i precedes S_j and there is an intervening statement S_i' so that S_i and S_j are define statements and S_i is a use statement with regard to v[3].

[2] All statements are assumed to be single assignment statements.

[3] For an overview on data and control dependency analysis see for example [PW86].

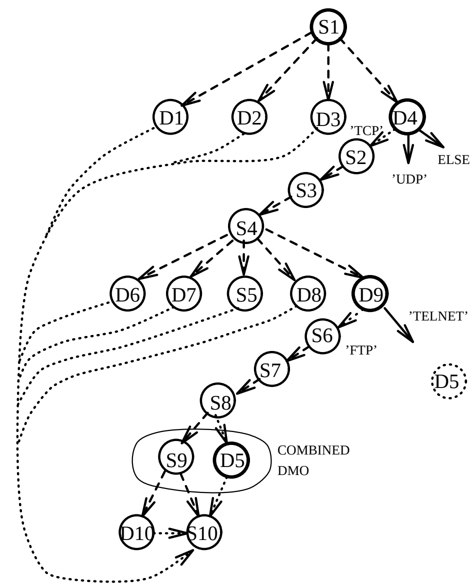

Figure 2: The relaxed dependence graph

The result of the data dependency analysis for our example protocol stack is shown in Figure 1.

Relaxation of control dependencies. The branching nodes in the common path graph and their successor nodes represent system executions which are almost certainly taken. We therefore replace the control dependency relations usually connecting neighboring program statements by a *relaxed control* dependency relation, namely that every non-branching node only depends on its closest preceding branching node. We define that dependencies along the common path are always transitive. Therefore, whenever a non branching node is transitively depending on a branching node by the data dependency relation then we do not need to introduce a new dependency edge. Furthermore, we need to introduce a relaxed control dependency between every non-branching node and every of those reachable exit nodes which are not already transitively depending.

In our example we need to introduce relaxed control dependencies from `D4` to `S2`, and from `D9` to `S6`, plus the edges leading from `D1, D2, D3, D5, D6, D7, D8` and `S5` to the exit node `S10` (see Figure 2).

Lazy Message processing. The idea behind *Lazy Message* processing (see [OP91]) is that combining data manipulation operations (DMOs) can be very efficient. A big portion of the execution time for DMOs can very often be attributed to fetching the operands whereas processing the operands is neglegible. Execution time can be saved if DMOs which have identical operands are executed in a combined fashion, thus saving unnecessary repeated fetch operations. It is therefore desirable to defer the execution of DMO's as much as possible in order to be able to combine as many DMOs as possible.

In order to perform Lazy Mesage processing we have to firstly identify all DMO operations that access entire packets and not only headers. Branching node control decisions do usually not depend on the packet data. In most cases they depend on the packet header. This implies that only exit nodes depend on DMOs and thus that DMOs can usually be deferred to the exit nodes. A DMO is deferred by removing it from a segment and replicating it in each segment depending on the next branching node. The deferral and replication is recursively repeated on every subgraph having its root in the branching node which delimits the segment graph from which the DMO is removed. This means that in general all DMOs will be replicated in all exit nodes which are reachable from their original location. In the exit nodes they are grouped and executed. However, it is still desirable to execute grouped DMOs as early as possible. Therefore they should be grouped and executed at that point in the graph in which no further DMOs can be added on the way down to all reachable exit points.

In our example we combine the DMOs `D5` and `S9` and we execute them together right before the exit node `S10` is reached.

Exiting the common path and ensuring consistency. The operations at the exit node now have to be modified to test all the results of the operations they depend on against the predicted results. If there is a match the normal exit node operations can be executed. However, if there is a mismatch the whole processing has to be restarted from the root node using a non-optimized code version of the protocol stack.

As the different tests have been decoupled from the operations which these tests 'guard' some consistency ensuring mechanisms have to be applied. For example, a division by zero may be executed concurrently with the test for non-zeroness of the respective operand. Thus the division operation must be made robust such that a system failure is excluded. Secondly, the operations which are executed concurrently with the test operations have to be be reversible. This can be ensured by restricting the code to read-only operations or by performing write operations only locally and performing a commit operation on the actual data when

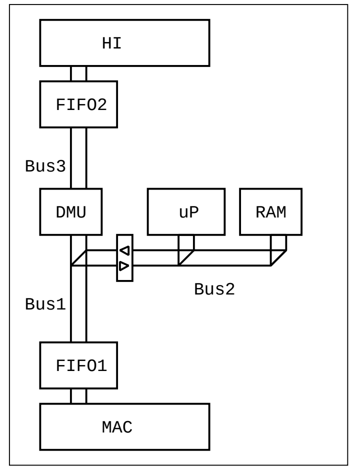

Figure 3: A hardware architecture for the receive path

the exit node is reached and all tests have had a positive result.

4 Scheduling the parallel operations

We now describe the compile time scheduling of the parallelized operations on a given hardware architecture. We consider the classic *resource constrained job scheduling* problem (see [CS90]). Both the classic problem formulation as well as our algorithm do not take delays induced by data transfers into account. We implemented a tool that in the current version uses an enumerative scheduling algorithm.

- First we have to compile a list of all independent resources available in the system on which the protocol is executing. For each operation of the common path graph we need to define which resources will be needed and estimate the execution time.
- Secondly, given this information all possible schedules can be enumerated and the ones minimizing the time at which the exit node are reached can be selected. Note that due to the relaxation of dependencies there are many degrees of freedom for the schedules. In particular it may be that operations of different layers turn out to possibly execute faster in a reverse order compared to the original specification.

The common path of the protocol will then be implemented according to the optimized schedule.

For this example we will use a simple hardware architecture (see Figure 3). It represents a network adapter board with two interfaces: the medium access interface (MAC) to the network and the host interface (HI) to the host machine. Only the receive path is covered by our example. There are two dual-port RAMs acting as FIFO's between the interfaces and the processing part. The processing part is made of a general purpose RISC microprocessor containing a cache, its program RAM (which also contains the state information for all open connections) and a Data Manipulation Unit (DMU), typically a *Digital Signal Processor* or a *Field Programmable Gate Array*. The DMU reads packets from FIFO1, does the necessary translations and calculations and stores the resulting packet in FIFO2. The microprocessor processes the headers and updates the state of the connections. Its bus (Bus2) can be isolated from the bus that connects FIFO 1 to the DMU (Bus1) to allow parallel processing. This is achieved using a bidirectional three-state buffer.

	FIFO1	Bus1	Bus2	uP	RAM	DMU	Bus3	FIFO2	cycles
D1	*	*	*	*					20
D2				*					5
D3				*					1
D4				*					1
D5+S9	*	*				*	*	*	375
D6			*	*	*				50
D7				*					1
D8				*					1
D9				*					2
D10				*					1
S1	*								1
S2				*					1
S5				*					200
S6				*					2
S10		*	*	*		*			10

Table 3: Required resources and execution time for each operation

For our example we will assume that the board is connected to a link delivering packets of 1500 bytes at 600 Mb/s, that the hardware is clocked at 25 Mhz, that the busses are 32 bits wide and that they can transfer one word per clock cycle.

Table 3 gives the list of resources and indicates which resources will be used for each operation as well as how many cycles are needed for execution[4].

[4] Operations `S3, S4, S7` and `S8` are not represented in the tables as they do not require any processing.

Note that the processor will access the packet header only once in FIFO1. Afterwards the header is retained in the processor cache. The time for transferring the header from FIFO1 to the cache has arbitrarily been attributed to operation D1. With an improved scheduling algorithm implicit data transfers like this one would be generated automatically and scheduled optimally.

operation	starting time	end time	operation	starting time	end time
S1	0	1			
D4	1	2			
S2	2	3			
D9	3	5			
S6	5	7			
D1	7	27			
D2	27	32	D5+S9	27	402
D3	32	33			
D6	33	83			
D7	83	84			
S5	84	284			
D8	284	285			
D10	402	403			
S10	403	413			

Table 4: An optimal schedule on the proposed hardware architecture

The optimal solution requires 413 cycles to reach exit node S10 from root point S1. One optimal schedule is given in Table 4. It corresponds to a maximum throughput of 728Mb/s which is about 21% more than the maximum throughput of the network. Whether these 21leave the common path is under investigation. The sequential execution of the original common path suite of operations on the same hardware would take 1036 cycles, which is more than twice as long as with the ompitimzed graph.

The total packet processing time is mainly to be attributed to the compund processing of the DMOs `D5` and `S9`. However, the compound execution of the DMOs is responsible for the considerable gain in efficiency in processing the whole packet.

5 Conclusions

We presented a method for the derivation of optimized parallel protocol implementations from formal specifications. The parallelization covers multiple protocol layers which distinguishes our method from the existing previously cited approches. Also, the degree of parallelism in the architecture we used as example is relatively limited compared to other approaches ([BZ92], [TZ93]). However, we have shown that even with this limited parallelism we gained considerably in efficiency. We claim that further parallelization will not improve our realtively good cost efficiency ratio, in particular because we saw that the most time consuming part of the packet processing is the data access related to data manipulation operations.

Further research will address incorporating our method into an integrated protocol engineering methodology which will cover all aspects of protocol specification, verification, performance evaluation, design and implementation. Our method is well suited for being integrated into a more complex context because it is relatively formal, even if not all parts are fully formalized yet. The incorporation of our method into a protocol engineering tool as well as improvements on the scheduling algorithm are further goals for future research.

Acknowledgements

The work of both authors was supported by the Swiss National Science Foundation.

References

[BHS91] F. Belina, D. Hogrefe, and A. Sarma. *SDL with Applications from Protocol Specification.* Prentice Hall International, 1991.

[BZ92] T. Braun and M. Zitterbart. Parallel transport system design. In A. Danthine and O. Spaniol, editors, *Proceedings of the 4th IFIP conference on high performance networking*, 1992.

[CJRS89] D. D. Clark, V. Jacobson, J. Romkey, and H. Salwen. An analysis of tcp processing overhead. *IEEE Communications Magazine*, 27(6):23–29, June 1989.

[CS90] T.C.E. Cheng and C.C.S. Sin. A state-of-the-art review of parallel-machine scheduling research. *European Journal of Operational Research*, 47:271–292, 1990.

[CT90] D. D. Clark and D. L. Tennenhouse. Architectural considerations for a new generation of protocols. In *Proceedings of the ACM SIGCOMM '90 conference*, Computer Communication Review, pages 200–208, 1990.

[Hei92] B. Heinrichs. XTP specification and parallel implementation. In *Proceedings of the International Workshop on Advanced Communications and Applications for High Speed Networks*, pages 77–84. Siemens AG, Munich, 1992.

[KS89] A. S. Krishnakumar and K. Sabnani. VLSI implementation of communication protocols - a survey. *IEEE Journal on Selected Areas in Communications*, 7(7):1082–1090, September 1989.

[OP91] S. W. O'Malley and L. L. Peterson. A highly layered architecture for high-speed networks. In M. J. Johnson, editor, *Protocols for High Speed Networks II*, pages 141–156. Elsevier Science Publishers (North-Holland), 1991.

[PS92] T. F. La Porta and M. Schwartz. A high-speed protocol parallel implementation: Design and analysis. In A. Danthine and O. Spaniol, editors, *Proceedings of the 4th IFIP conference on high performance networking*, 1992.

[PW86] D. A. Padua and M. J. Wolfe. Advanced compiler optimizations for supercomputers. *Communications of the ACM*, 29(12):1184–1201, 1986.

[RK92] E. Rütsche and M. Kaiserswerth. TCP/IP on the Parallel Protocol Engine. In A. Danthine and O. Spaniol, editors, *Proceedings of the 4th IFIP conference on high performance networking*, 1992.

[TM92] A. N. Tantawy and H. Meleis. A high speed data link control protocol. In A. Danthine and O. Spaniol, editors, *Proceedings of the 4th IFIP conference on high performance networking*, 1992.

[TZ93] A. Tantawy and M. Zitterbart. Multiprocessing in high performance IP routers. In B. Pehrson, P. Gunninberg, and S. Pink, editors, *Protocols for High Speed Networks, III*, pages 235–254. North-Holland, 1993.