PSi: A SILICON COMPILER FOR VERY FAST PROTOCOL PROCESSING

H. Abu-Amara, T. Balraj*, T. Barzilai, Y. Yemini
IBM, T. J. Watson Research Center
Yorktown Heights, NY 10598

*Columbia University, Computer Science Department, NY, NY 10027

CUCS-477-89

CUCS-477-89

PSi: A SILICON COMPILER FOR VERY FAST PROTOCOL PROCESSING

H. Abu-Amara, T. Balraj†*, T. Barzilai, Y. Yemini
IBM, T.J. Watson Research Center,
Yorktown Heights, NY 10598.

ABSTRACT

Conventional protocols implementations typically fall short, by a few orders of magnitude, of supporting the speeds afforded by high-speed optical transmission media. This protocol processing bottleneck is a key hurdle in taking advantage of the opportunities presented by high-speed communications. This paper describes PSi, a silicon compiler that transforms formal protocol specifications into efficient VLSI implementations. PSi takes advantage of the parallelisms intrinsic to a given protocol to accomplish very high-speed implementations. Initial application of PSi to the IEEE 802.2 (logical link control) lead to processing rates in the order of 10^6 packets per second (p/s). The 802.2 was selected as a benchmark of complexity; light-weight protocols can accomplish even higher processing rates, reaching the limits set by chip clock rates (i.e., a packet per cycle). These speeds significantly exceed typical of software implementations (up to a few hundreds p/s) or special hardware-assisted implementations (up to a few thousands p/s). More importantly, at these rates when the packet size is 10^{3-4} bits the protocol thruput of 10^{9-10} bits/sec reaches the limiting thruput afforded by memory technology. Thus, the protocol processing bottleneck is pushed to the ultimate bounds set by VLSI technologies.

1. BACKGROUND: THE PROTOCOL PROCESSING BOTTLENECK

The rapid advent of optical communication technologies resulted in a significant increase of the speeds at which data may be transmitted. Data transmission rates in the order of 10^{8}-10^{9} bits per second (b/s) are becoming a commercial reality while speeds of 10^{9-12} b/s are explored at various research labs. This increase in transmission rates reverses the traditional relations among processing and communications technologies. In fact, a bandwidth in the order 10^{9-10} b/s is the current limit of very high-speed processor-memory communications. Therefore, the transmission rates afforded by optical communication technology meet or exceed the rates afforded by processing technologies. Unfortunately, in between the network transmission layer, where bits are communicated, and the processor-memory bus, over which these bits are processed, a collection of protocol layers regulate the flow of these bits. These protocol layers are processed at speeds significantly slower than those supported at the transmission and processing levels. Protocol processing speeds thus present a significant bottleneck in utilizing the speeds afforded by optical transmission technologies to accomplish significantly new applications.

Why is protocol processing slow? How can we improve protocol design to support high speed communications? How can we improve protocol implementations to accomplish orders of magnitude speed increase? These questions attracted significant interest recently [1-7]. Figure 1, below, depicts the typical processing tasks associated with a protocol (center rectangle) and the data that they use (surrounding rectangles).

---

† Columbia University, Computer Science Department, NY, NY.
* Work supported in part by DARPA contract #F-29601-87-C-0074

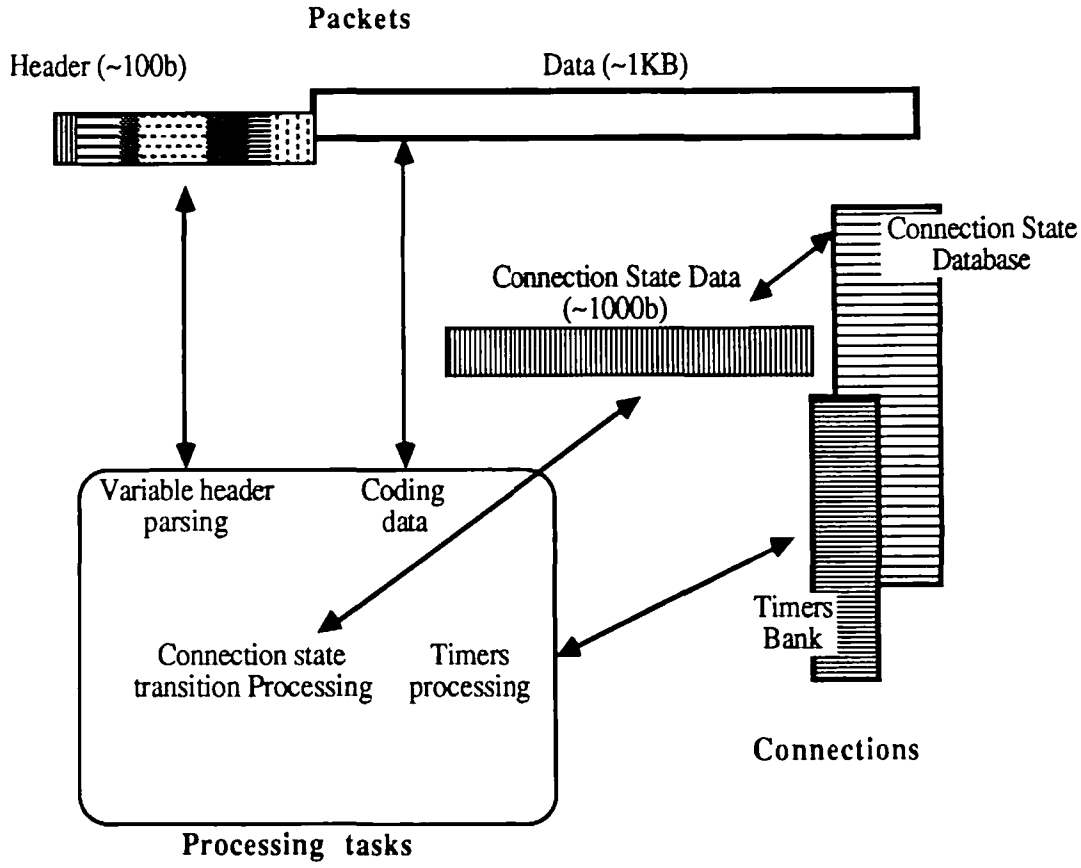

Figure 1: Protocol processing bottlenecks

Protocol processing is slowed by:

* Transfer of large packets through protocol layers, requiring multiple memory accesses for each bit in the packet.
* Large (and variable) header sizes, requiring complex parsing of header into processing components.
* Switching of connection context data (each requiring in the order ~1K bits) to process respective connection events (e.g., packet arrival).
* Maintaining a large bank of software timers; with 3-5 timers per connection, it is necessary to maintain 10^{3-4} timers.
* Processing complex state transitions associated with a protocol event
* Processing data coding functions (e.g., encryption) at higher layer protocols.

Protocol processing tasks are typically implemented in software, utilizing a traditional single processor Von-Neumann architecture. As a result, execution of the protocol proceeds through sequential processing of the different tasks associated with a given event. The processing hardware is shared not only by the different processing tasks but also by multiple layers. As a result, the entire processing of a given packet is accomplished sequentially through a few layers, resulting in very slow processing of protocol events.

A protocol may require in the order of 10^5 clock cycles to complete the processing of a packet, thus leading to typical processing rates of 10^2 packets per second. Assuming packets of 10^{3-4} bits, this translates to thruputs in the order of 10^{5-6}b/s far below high-speed media rates.

Fast protocol processing may be important not only as means to deliver greater thruputs in fast networks, but also as means to accomplish improved response to massive failures in "slow" networks (i.e., utilizing slow transmission media). Massive failures (e.g., in an X.25 network) involve simultaneous loss of a large number of virtual circuits. This results from a loss of a physical link (e.g., due to sustained noise) or a host crash. The loss of a virtual circuit will trigger time-outs and retransmissions, requiring significant more processing than that of normal packet flow on the circuit. A massive failure of multiple virtual circuits will thus result in a sudden surge of processing demands placed on the protocol controller. This surge may lead to trashing of the protocol controller as its processing queue overflows, finally causing a crash. The problem may further propagate to higher protocol layers as their timers begin to trigger when the lower layer processors response time degrades. As a result a single link (or host) failure may evolve into a total network crash. Such evolutions of massive failures have been observed in actual X.25 networks. Faster protocol processors could, obviously, eliminate such failures due to insufficient processing capacity (to handle massive failures) in network protocol controllers.

The main tenet underlying this paper is that protocol processing speeds are primarily bound by limitations of traditional software-oriented implementation architectures. By pursuing highly-parallel implementations in silicon, the bottlenecks described above disappear ,leading to very high-speed implementations. In what follows we describe other approaches used to accomplish high-speed protocol processing.

2. TOWARDS FAST PROTOCOL PROCESSING

A number of proposals to address the protocol processing bottleneck have been recently explored. One possibility [3] is to establish new light-weight protocol architectures that minimize the respective processing loads: the number and complexity of protocol layers may be reduced, headers may be trimmed and simplified, functionality may be reduced and the state machine simplified. Simplifications may reduce the number of processor cycles used to process a packet to an order of 10^2 (down from 10^5) per layer. Using standard or RISC processors combined with some pipelining the protocol processing thruputs may reach an order of 10^3-10^5 p/s.

Alternatively, it is possible to utilize increasing degree of special hardware processing architectures to improve the protocol processing speeds [6,3,5]. Protocol processing tasks (e.g., header parsing, context switching, state-machine processing) may be pipelined and/or processed concurrently. The idea of pipelining is used in [6,3,5]. Krishnakumar, Krinshnamurthy and Sabnani [6] pursue a special microprogrammable processor architecture to accomplish high-speed implementations of arbitrary protocols. Chesson et al. [3], pursue a VLSI lay-out of a specialized light weight protocol XTP, to accomplish high speed implementation. Kanakia and Cheriton [5] pursue pipelined architecture to accomplish high-speed implementation of high-level protocol processing functions. Williamson et. al. [7] pursued a multiprocessor architecture for parallel protocol processing.

These two avenues to improved protocol processing performance are mutually orthogonal. The combination of both light-weight protocols with specialized processing architectures may yield the ultimate resolution of the protocol processing bottlenecks.

It is easier to conceptualize the problem of high-performance protocol processors and the relations between the different technologies, using a speed chart as provided by figure 2 below. The horizontal axis describes processing speeds as measured in packets processed per second. The vertical axis describes the size of a packet. The thruput measured in bit/second is the product of the packet size and the rates of packet processing in p/s. Since both axes use logarithmic scale, the equal thruput curves are linear. The ultimate limits on protocol processing speeds are established by the boundary lines. The vertical line on the right establish the bounds on the p/s processing rates set by chip-clock rates (and the number of cycles required to process a packet). The 1Gbs thruput line on the right sets the bounds established by memory technology on the speeds at which packet bits may be read and written. While these physical bounds, on clock and memory access rates, may be improved slightly via higher-speed logic technologies, they set the ultimate bounds on protocol processing speeds. The horizontal lines set the bounds on packet size and headers; both may be flexibly adjusted by protocol design to optimize performance goals.

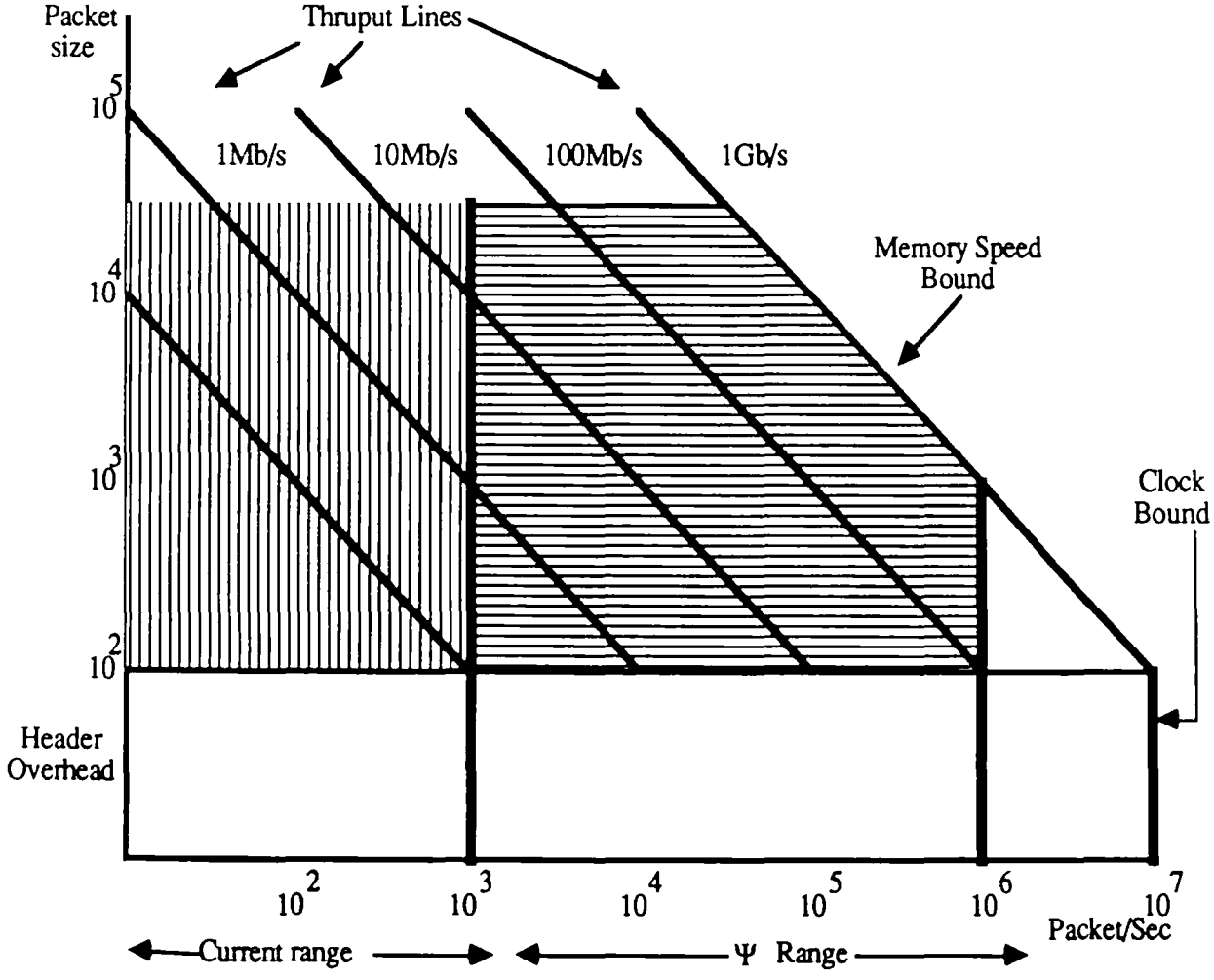

Figure 2: The protocol processing performance landscape

The goal of high-speed protocol processing may be described as increasing the processing rates measured in p/s. Current high-speed protocol implementations range up to a few thousands p/s. The PSi compiler can be used to build protocol silicon processors whose speeds range a few orders of magnitude higher, closing the gaps to the physical limits (or reaching very closely).

3. Ψ , A PROTOCOL SILICON COMPILER

Silicon compilers [8] consist of software tools that transform high-level specifications of processing elements into efficient and correct VLSI layouts. Silicon compilation is a particularly attractive approach to high-speed protocol processor implementations for the following reasons:

1. Protocols admit significant degree of intrinsic parallelisms that may be taken advantage of by a silicon implementation (i.e., vs traditional sequential processing). Thus, for example, different actions associated with a protocol transition may be simultaneously processed. Such parallelism can be used to accomplish reduction in protocol processing time.
2. Protocols typically follow similar processing patterns that may be captured by uniform layer processing architecture templates. Template implementation admits parametrized description of the protocol processor components in terms of appropriate stored silicon macros.
Template architecture can further simplify the process of protocol processor implementation resulting in a faster development cycle and more reliable implementations.
3. Protocol interfaces tend to remain relatively stable to changes. Therefore, pure hardware implementations do not represent inadequate restriction compared to a software implementation.
4. Implementation in silicon offers new forms of optimizing protocol processing structures. Thus, for example, it is possible to eliminate some of the fundamental restrictions associated with software implementations (e.g., excessive data motions). Memory and processing logic may be laid out in a manner to accomplish greater speeds.
5. Protocol processing functions are typically very simple, involving such operations as incrementing counters and setting timers. This lends protocols to very effective silicon compilation and results in processors that are more efficient than general purpose processors intended to handle the complexity associated with general software.

PSi, a Protocol Silicon compiler, aims to take advantage of these opportunities. Specifically, the two goals of PSi research are to accomplish:

* very high-speed protocol implementations for arbitrary protocols
* simplified protocol implementation process leading to reduced development time and increased reliability and uniformity of the resulting implementations

PSi accomplishes the high target speeds by trading off chip area for speed. Conventional hardware implementations of protocols have a single processor or a small number of processors that handle all the protocol connections. Significant time is wasted in switching connection context state data. In the PSi architecture, the state memory and the processing are closely integrated. The result is that every connection has a dedicated processor and context-switching delays are eliminated. This results in duplication of processing logic, and consequently an increase in chip area.

However, through the use of decomposition techniques described in section 5, this added area may be reduced. Further, the elimination of context-swapping logic provides an additional saving in area. The resultant area overhead is a small cost to pay for the dramatic speeds achievable.

PSi is based upon two major concepts: a template layer processing architecture and specialized highly parallel connection, header and output processors. The template architecture structures the processing relations generic to protocol layers. The specialized processors to handle connection state transitions and interfaces with neighboring layers (header and output processors) utilize intrinsic parallelisms of a given protocol and minimization of data motions to optimize the processing speeds. In what follows, we describe concepts in details and examine applications to the IEEE 802.2 (LLC) protocol.

4. THE Ψ LAYER PROCESSING ARCHITECTURE

The goal of this section is to describe the template architecture used by PSi to process protocols. The key components of a protocol (layer) processor are depicted in figure 3 below. Frames arriving from neighboring layers are processed through a pipeline consisting of a header processor, connection processors and an output processor. The frame is split into header and data. The data is stored in a memory (typically shared by multiple layers). The header processor parses the frame header and directs respective actions to an appropriate connection processor. The header processor views connection processors as memory addresses to which it writes the respective frame header elements. The connection processors execute the appropriate state transitions and pass their output to the output processor which, in turn, relays it to neighboring layers.

4.1 The Header Processor

The header processor performs all the connection independent functions of the protocol for incoming messages to the layer. These messages may be incoming packets passed to it by the lower layer, or commands (including commands to send outgoing packets) from the higher level. The header processor first parses the message to determine if it is connection related, and if so, to which connection it refers. If the message is connection-independent (e.g. set up commands for the station, or datagrams), the header processor does the relevant processing and sends any required response directly to the output processor via a set of buffers that behave as a dummy connection processor.

The header processor may be efficiently implemented (depending on its complexity) by fast processors (e.g., signal processors, RISC processors) reaching the range of 10^6 p/s. Alternatively, a special purpose micro-programmed processor may be used [6] and the microprogram be compiled from the header syntactical specifications (using compiler generation techniques). Such implementation may accomplish substantial header processing thruputs. The PSi compiler goes a step further to produce a design for a VLSI implementation of a custom header processor, based on the protocol specifications.

If the message is connection-dependent, the header processor identifies the connection to which it refers. In many cases, this is straightforward --- the message contains the connection number. In some cases, however, getting the connection number involves using a hashing function or other mapping algorithm. For example, in the case of 802.2, for incoming packets, the connection number must be determined from the source and destination addresses contained in the message.

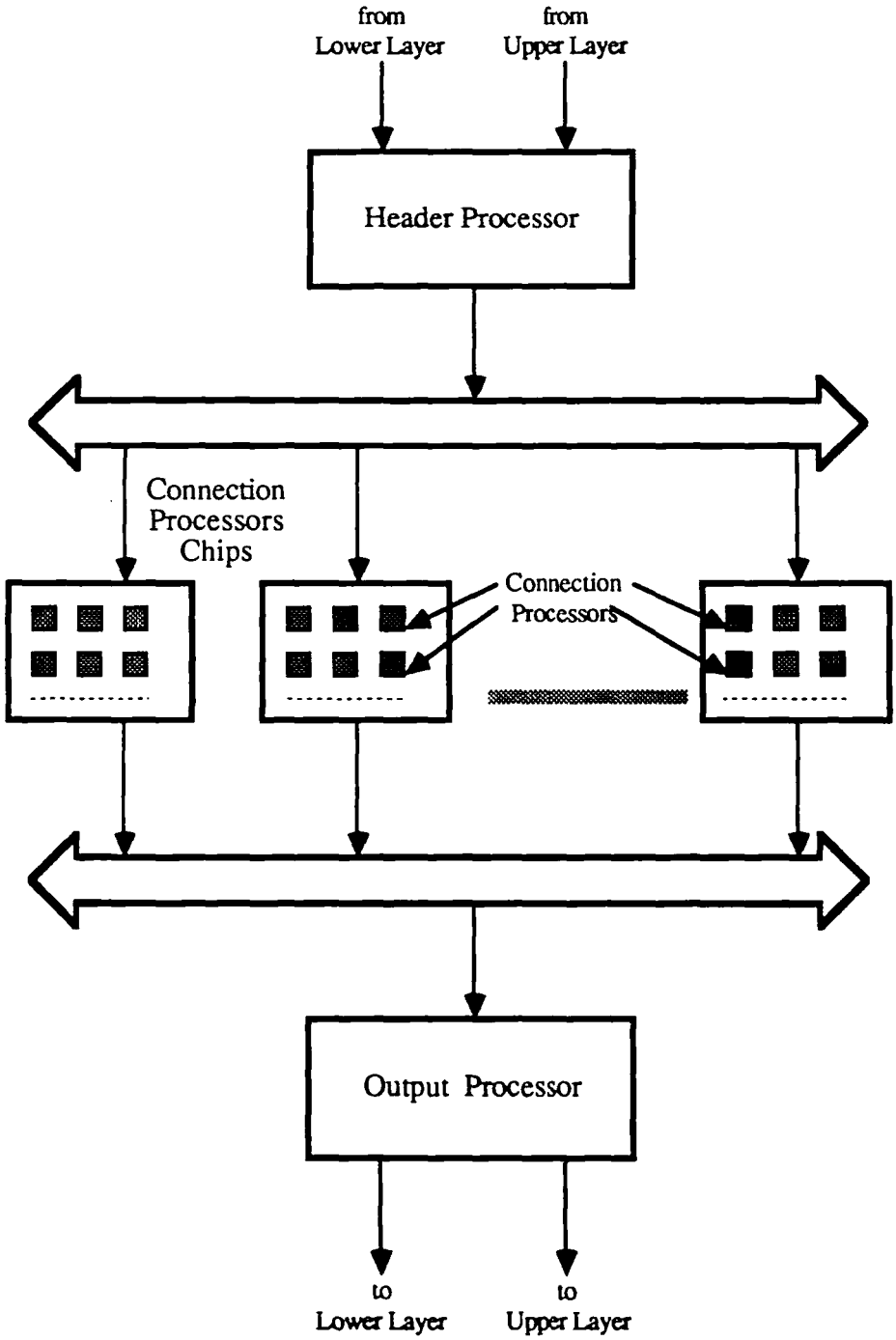

Figure 3: Protocol processor architecture

A direct look-up table cannot be used since the table is very sparse and the area requirements would be prohibitive. Software implementations normally use a hashing function, but this is too expensive in both time and area for our purposes. We plan to use a content addressable memory. The PSi architecture allows the option of using an external module to compute the connection number, if needed.

The rest of the message header is also parsed and transformed into a format suitable for input to the connection processor. This information --- consisting of an event code and several parameters --- is written to the appropriate connection processor.

Variable format headers are handled by parsing the header in parallel for each possible format and choosing the correct output. This maximizes the parsing speed at the expense of increased chip area. Thus, for example, the LLC control field [13] is parsed according to the three possibilities of an information/ supervisory/ unnumbered-frame and the decision on the case is easily determined upon completion of the parse. Mapping of packet addresses to connection numbers is a difficult problem in certain protocols (including the 802.2). Software implementations utilize hashing to resolve connection number.

4.2 The Output Processor

The output processor performs the inverse function to that of the header processor. It takes the outgoing message from the connection processor and converts it into a format suitable for transmission to the upper and lower layers of the protocol. This involves assembling packet headers in the format prescribed by the protocol, and looking up the source and destination addresses associated with the connection. In the case of protocols where the header formats are not very complex, and the connection-independent processing is simple, the output processor could reside in the same chip as the header processor.

4.3 The Connection Processor (CP)

The main functions of the connection processor are to execute the transitions associated with protocol events, retain the state information and trigger the generation of packet transfers by the output processor. The connection processor receives from the header processor the parameters describing the respective packet to be processed: the event type, associated counter values, and a pointer to the respective packet. Consider, for example (using the 802.2), an arrival of a supervisory frame [13]. The header processor will send to the connection processor the associated event (e.g., Receive Ready, Receive Not Ready, or Reject), and parameters describing the event (e.g., the value of the Command/response bit, the value of the Number Received used for acknowledgement). The detailed architecture of the CP is presented in figure 4 below.

The interface to the header processor (at the top) includes the event type associated with the incoming packet, the parameters associated with the event and a pointer to the respective packet data stored in memory. The flags, counters and timers are active units storing the connection state data. This design eliminates the need to switch or move connection context data as the entire context is stored in active memory (counters, flags, timers). Additionally, the use of dedicated timers per connection eliminates the overheads associated with the maintenance of a timers-bank stored in memory; complex and time-consuming updates and management of time-outs associated with memory-timers are avoided.

The state and incoming packet data are passed through combinatorial logic used to direct the state transition processor. The full arrows indicate the flow of the data through the combinatorial logic and into the state transition machine. The state transition machine executes the appropriate transitions associated with protocol events; it updates the respective counters, flags and timers (through appropriate control signals sent over the respective shaded arrows) and generates output directed to the output processor via the output unit.

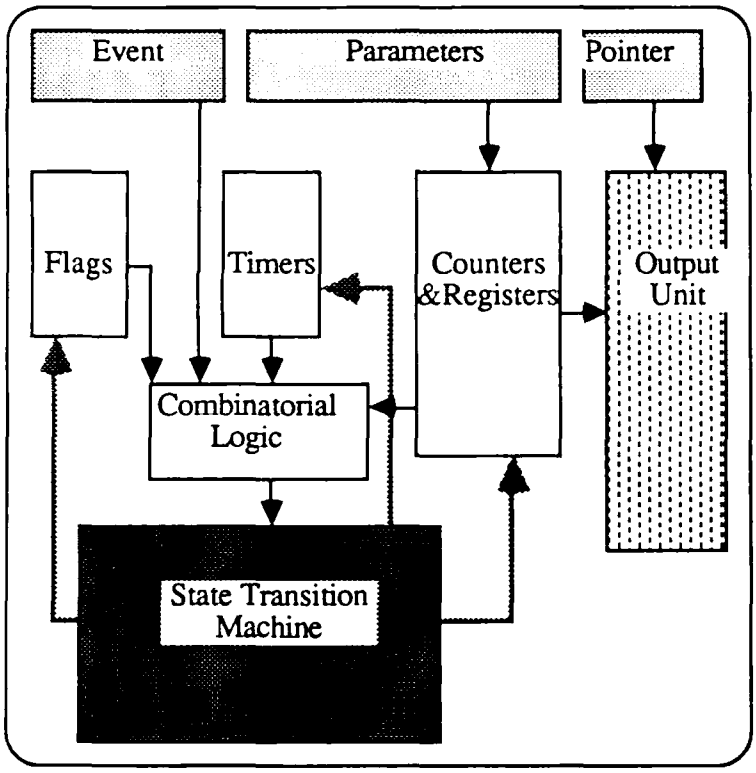

Figure 4: Template Connection Processor Architecture

The connection processor is thus organized as a template architecture parametrized by the specific sizes associated (by a given protocol) with the counters, timers, flags and interfaces. These parts are implemented as parametrized macro-cells of PSi. The state transition machine, the heart of the connection processor, is the part that changes substantially form one protocol to another; its specific composition must be derived from the specifications of the protocol state transition automaton. Deriving highly-parallel, efficient implementations of protocol state transition automata in VLSI is one of the fundamental aspects of PSi; it is explored in the next section.

4.4 The Pipeline

The pipelining of the packet processing is accomplished through two buses, as depicted in figure 3 above, regulating the communications from the header processor to connection processors and from connection processors to the output processor.

A few important aspects of this organization should be noted. First, it is possible for arbitrary combination of header, connection and output processors to be used in implementing the processing pipeline. Second, a connection processor chip may (and will typically) contain a number of connection processing units, depending on the complexity of the protocol. In fact, it may be expected that appropriate light-weight protocols may be translated into scores or even hundreds of connection processors on a single chip. The architecture above aims to support this spectrum of possibilities.

A key issue to consider is that of storing connection states in memory (and paying the cost of switching) vs. having dedicated connection processors. The cost of switching connection context is both significant time used to move context data as well as memory management logic. The cost of dedicated connection processors is primarily chip area. A precise evaluation of the tradeoffs depends greatly on the specific protocol and its implementation. However, a crude general assessment is possible. A counter or a timer requires about twice the area of a respective memory. Therefore, the use of dedicated timers or counters instead of memory will result at worst in area waste of 50%. However, since memory is typically organized along word boundaries storing the value of the counter/timer will typically involve waste of more than 50% anyway.
Therefore, having dedicated counters/timers vs. stored values will not result in significant area waste and provides a significant gain in time. The tradeoff is thus reduced to dedicated vs. shared state-transition processing logic.

In the context of light-weight protocols connection processing requires relatively little amount of logic. A single chip may support scores to hundreds of connection processors. Connection processing cells are functionally arranged as a shared memory between the header and output processors. This offers an attractive paradigm for fast processing of protocols. Dedicated connection processors offer significant attraction at little cost. For protocols of greater complexity, it may be desirable to keep the number of connection processors smaller. This requires some multiplexing of multiple connections over a single connection processor and switching among them. One possible approach is to add logic to the connection processors to support local (on chip) storage of connection context and duplicated timers for each stored connection. Thus, a single connection processor chip may include hardware timers and connection context memory to store 5-20 different connections. An alternative approach is to have shared connection memory and software timer-bank shared among the connection processors. This approach is closer to current practices and suffers from similar drawbacks associated with the costs of context switching and management of software timers-bank.

The approach pursued by PSi at this time is that of having dedicated connection processors: one per connection. In the context of light-weight protocols, processing speeds are of paramount importance and thus the cost of dedicated processors is traded against the cost of context switching. Furthermore, since a single chip may accommodate hundreds of connection processors dedicated processing becomes an attractive cost/effective implementation choice.

5. COMPILATION OF PROTOCOL STATE TRANSITIONS AUTOMATA

How should the state transition automaton associated with a given protocol be compiled into efficient silicon implementation? There are a few important aspects that render the answer non-trivial.

Rbusy Path: [(a+b+c)*;d;e;d*;(a+b+c);f]*

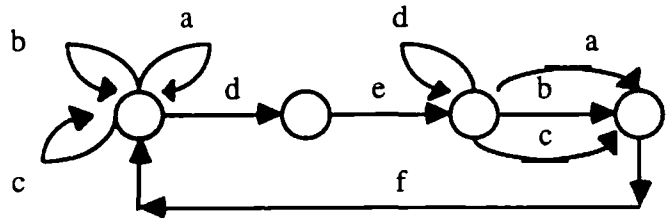

Figure 5: A path expression example

The power of path expressions lies in the way two or more path expressions interact. The automata corresponding to the various path expressions operate independently, synchronizing their activities at occurrences of common events. Thus, each path expression can be used to govern an independent aspect of the protocol in relative isolation from the rest. The entire 802.2 protocol may be decomposed into a small number (under 20) of such path machines that execute transitions in parallel. An arriving event (say RR) may trigger a number of path event executions all of which are simultaneously completed. The complete protocol may be viewed as a result of composing these parallel executions. The parallel decomposition also accomplish a great deal of reduction in the redundancy associated with state-transitions table descriptions (e.g., 60 pages of 802.2 tables were condensed into 4 pages of path expressions).

5.2 Compiling Path-Expressions Into VLSI

How can path-expressions be compiled into efficient VLSI implementations? Answers to this question have been explore by [10,11,12]. Miss Manners, a silicon compiler for path expressions [12], is used as a key component by PSi in compiling the state transition processor. The key idea pursued by Miss Manners is that of implementing in silicon the equivalent of a parse-tree for path expressions. An example is described in figure 6 below which depicts the Rbusy path layout.

The leaves of the tree are connected to the logic associated with the respective events and actions: a,b,c etc. The event logic causes the respective event leaves to be triggered. The logic for actions is ready to be triggered by signals arriving from the respective leaves. The major role of the tree is to sequence the acceptance of events triggered at the leaves and the triggering of actions by the leaves. This sequencing is accomplished via the internal nodes representing the path operators.

The internal nodes of the tree implement the logic of the path operators: "+", ";" and "*". The details of the operator cells are depicted on the left side of the figure. The "+" cell is implemented as an Or-gating of the up going input signals and a free-flow transfer of down-going signals.
The "*" cell is implemented as an Or-gating of the inputs (either from above or below) and a flow of the result both up and down. The ";" cell is responsible to direct the flow of execution coming from above to the left subtree, coming from the left subtree to the right subtree and coming from the right subtree to the parent above; the "*" -cell is merely a notational convenience as it has no logic (only wires). For convenience we have numbered the cells and the respective operators which they implement.

One may attempt a naive, brute force, PLA implementation of the finite state automaton prescribed by typical protocol state transition tables. For example, the state transition tables of the IEEE 802.2 [13] implementations typically stretch over 50-80 pages and would result in a PLA requiring a few scores of chips. A more reasonable approach is to reduce the complexity of the state machine implementation by taking advantage of the replication intrinsic to state transition table descriptions and of the parallelisms intrinsic to protocols. A formalism is required that can capture decomposition of these intrinsic parallelisms into independent processing logic elements.

5.1 Path-Expression Specifications of Protocols

One attractive formalism to describe state transition processes is offered by the use of path-expressions [9]. A path expression is essentially a regular expression description of a state transition system; it is formed from events and actions symbols combined with three operators: ";" representing sequential composition, "+" representing non-deterministic choice and "*" representing repetition.

For example, consider the following path describing a component machine of the 802.2 protocol, used to handle processing of activities related to situations when the remote station is busy.

Rbusy path:

[(RR+REJ +I_LPDUr1)*;RNR; (Rb=1&Snd=0&load(Is_Ct)&IH_Rb);
RNR*; (RR+REJ+I_LPDUr1); (Rb=0&IH_Rb_Off)]*

This path describes the following execution behavior: an arrival of an event of type RR (receive ready), REJ (reject) or information packet should be accepted with no response (machine cycles through as denoted by the *).

Upon arrival of an RNR (receive not ready) event, a few local flags and counters are set (e.g., Rb is a remote busy flag) and the higher layer is informed (i.e., IH_Rb denotes a message to the higher layer of a remote busy condition). Successive arrivals of RNR are ignored. Upon arrival of a frame of type RR, REJ or I_LPDUr1 the respective local flags are set to indicate that the remote station is no longer busy (i.e., Rb=0), the higher layer is informed (IH_Rb_Off) and execution returns to the beginning of the path.

This example demonstrates two additional important features of protocol event processing. First, the actions associated with an event may typically be processed in parallel. Thus, an arrival of an RNR causes all four actions Rb=1, Snd=0, load(Is_Ct) and IH_Rb to be executed simultaneously. Second, typical processing associated with a protocol event involves setting of flags (Rb, Snd), adjusting counters (load(Is-Ct)) or timers, and/or informing neighboring layers (IH_*). These tasks require fairly simple processing logic (vs. the complexity of a general purpose processor).

The finite state machine associated with the Rbusy path is depicted in figure 5 below. The event letters (a,b, c...) are used to replace the more cumbersome (but more meaningful) notations above (RR, REJ, ...).

The tree cells follow the path-expression in sequencing the acceptance of external events or triggering actions associated with respective leaves. Sequencing is best conceptualized in terms of a "token" traversing the tree nodes. The "*"-cells circulate the token internally offering their offspring nodes to take the token whenever they are available to accept it. The ";"-cells simply transfer the token through. The "+"-cells select a token, whenever available, from their offspring nodes passing it upward to their parent cells.

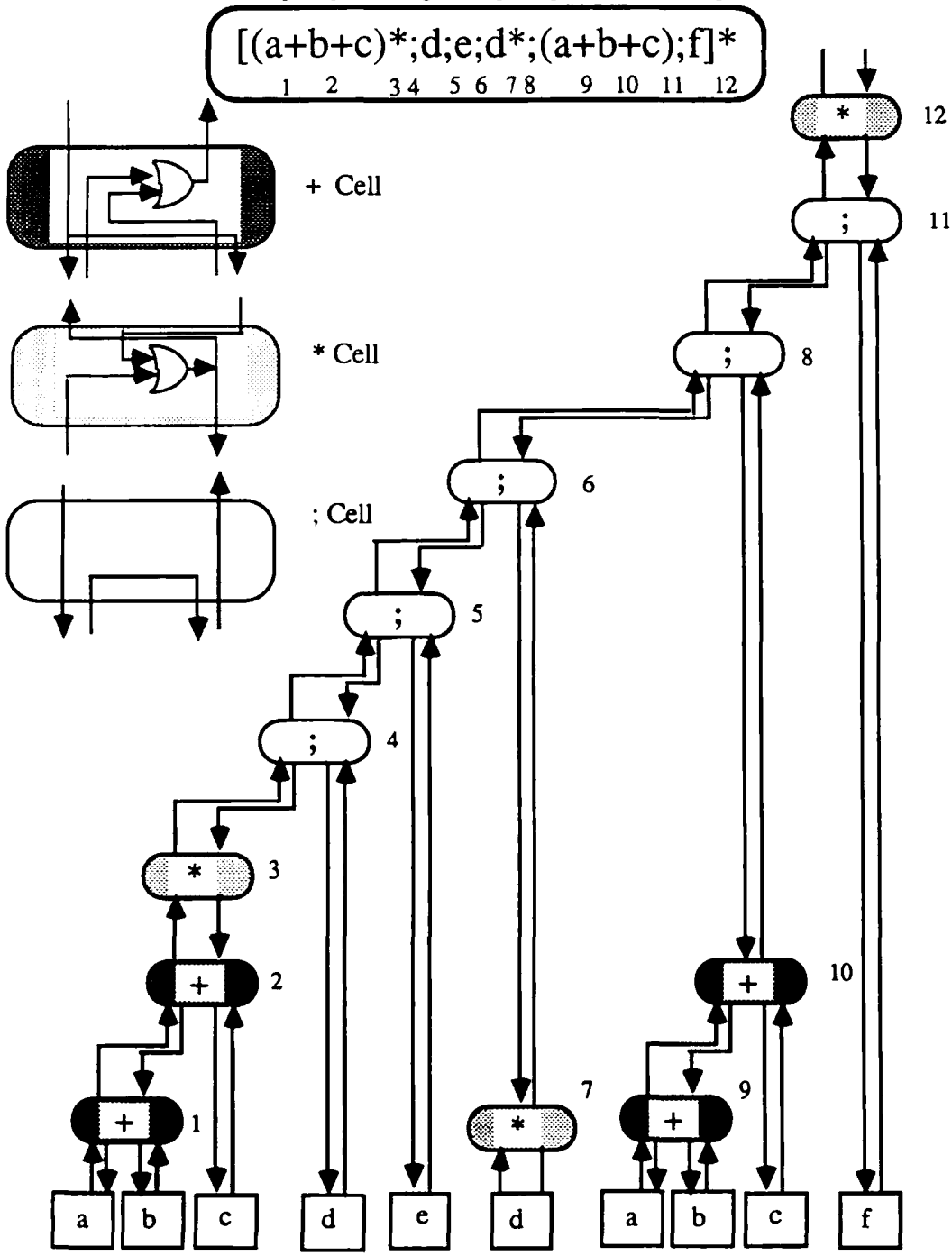

Figure 6: Layout of the Rbusy Path

Initially the token is kept at the root of the tree (12) where it is circulating until one of the offsprings of the "*"-cell is ready to accept it. The ";" cells below (11-8-6-5-4) will force the leftmost offspring to be selected thus making the token available for the "*"-cell #3, corresponding to the initial repetition of the subpath (a+b+c)*. The token will circulate at "*"-cell (3) making itself available to its offsprings. This means that a triggering (by an external arrival) of either "a", "b" or "c" will cause the token to accept the respective event (by moving down the tree to the corresponding leaf) and then return, through the two "+"-cells (1) and (2), back to the "*"-cell (3). This execution pattern implements the subpath (a+b+c)* of the Rbusy path.

Upon occurrence of the event "d", the token will flow from the "*"-cell (3) through its father ";"-cell (4) into "d", accepting the event and then climb back to (4) which will direct send it to its father ";"-cell (5) and then to the leaf "e". Note that this flow through ";"-cells involves no gate delays as it involves no execution logic. As the token arrives to "e", it will trigger the execution of the respective actions and then move up through the ";" cell (5) to its father (6) and down to the "*"-cell (7) where it will be circulating waiting for an arrival of the event "d" below. This entire sequencing of events corresponds to the execution of the subpath (a+b+c)*;d;e;d*. The sequencing of the remainder path follows suit through similar argument.

A few important aspects of this path expressions implementation should be noted. First, the time associated with the execution of an event is the time to traverse the respective subtree between the leaf of the event and the next waiting point for an incoming event plus the time required to complete the actions associated with any leaves traversed. The time to traverse the tree may be easily predicted as the only contributions are the gate-delays associated with "+"-cells (";"-cells do not contribute delay). The time to execute the respective actions may also be easily predicted as actions involve simple (parallel) operations to adjust counters, set timers, set flags, or trigger an output message. Therefore, the time to execute an event may be accurately predicted directly from the syntactical structure of the path-expression. This provides an invaluable tool in assessing and optimizing the timing associated with execution of protocol events.

Second, tree structured logic admits efficient lay-outs as far as area consumption goes [14]. Path expressions layout provide significant advantage in implementing sparse transition automata as no time or area penalty (unlike PLAs) is incurred for non-existing transitions. Protocol state machines are typically sparse in that at a given state one anticipates a small number of possible events to occur; unexpected events are typically handled uniformly (i.e., as errors to be ignored). This typically permits decomposition of the protocol into highly sparse parallel automata where a typical state has 1-3 transitions leading out. Path-expression lay-out then take advantage of the intrinsic sparsity of protocol transition automata.

6. CONCLUSIONS

The ideas presented above have been pursued in the implementation of PSi. The 802.2 protocol was selected as a benchmark, as its complexity is substantially greater than what may be expected from high-speed protocols. Preliminary implementation of the 802.2 shows that the total time associated with the processing of a protocol event is about 10^{-6} sec. Therefore, one can accomplish the target thruput rates of 10^6 p/s even for protocols of full-fledged complexity. The area requirements of the 802.2 do not permit laying more than a small number of connection processors on a single chip. This, however, is radically different for light-weight protocols where scores of connection processors may be laid out on a single chip.

The main value of the 802.2 as a benchmark has been to establish that protocol processing bottlenecks are due primarily to implementation architecture rather than design. A silicon implementation can overcome these bounds and eliminate the bottleneck entirely. This results in high-performance protocol implementations whose thruputs are bounded only by the physical limits of memory and processing technologies. The key reasons why PSi implementations can accomplish these thruput levels are: reduction of the data motions associated with protocol processing to the minimum possible and decomposition of intrinsic protocol parallelism into parallel path machines. The ability to decompose the intrinsic parallelisms of a protocol is key to accomplishing efficient VLSI implementations. Current PSi research is focused on automating the process of such decompositions.

In summary, we feel that the main contributions of this work is to establish that efficient VLSI implementations can remove the protocol processing bottleneck and deliver the kind of thruputs required by high-speed networks. A no-less significant value of a protocol silicon compiler is the great simplification, uniformity and streamlining of the implementation process due to the template architecture and the high-level design tools used.

BIBLIOGRAPHY

[1] K.A. Lantz, W. Nawicki, M. Theimer, "Network Factors Affecting the Performance of Distributed Applications", Proceedings, ACM SIGCOMM 84, 1984.

[2] R.A. Watson and S.A. Mamrak, "Gaining Efficiency in Transport Services by Appropriate Design and Implementation Choices", ACM Transactions on Computer Systems, Vol. 5, no. 2, May 1987.

[3] G. Chesson, B. Eich, V. Schryver, A. Chereson, A. Whaley, "XTP Protocol Definition", report revision 3.1, Silicon Graphics, March 1988.

[4] D.R. Cheriton, "VMTP: A Versatile Message Transaction Protocol", technical report RFC 1045, DARPA, February 1985.

[5] H. Kanakia, D.R. Cheriton, "The VMP Network Adapter Board (NAB), High Performance Network Communication for Multiprocessors", Proceedings, ACM SIGCOMM 88, 1988.

[6] A.S. Krishnakumar, B. Krishnamurthy, K. Sabnani, "Translation of Formal Protocol Specifications to VLSI designs", Proceedings, Protocol Specification, Testing &Verification VI, Rudin &West (ed.), North Holand, 1987.

[7] R. Williamson et. Al., These proceedings.

[8] D.D. Gajski (ed.), "Silicon Compilation", Addison Wesley Co., 1988.

[9] R. H. Cambell, A.N. Habermann, "The Specification of Process Synchronization by Path Expressions", Lecture Notes in Computer Science 16, Goos & Hartmanis (ed.), Springer Verlag, 1974.

[10] A.N. Habermann, "Implementation of Regular Path Expressions", Tech report no ANH7902, Carnegie Mellon University, 1975.

[11] T.S. Ananthraman, E.M. Clarke, M.J. Foster, B. Mishra, "Compiling Path Expressions into VLSI Circuits", Distributed Computing, Vol 1, no 3, 1986.

[12] T.S. Balraj, M. J. Foster, "Miss Manners: A Sspecialized Silicon Compiler for Synchronizers", Proceedings of the 4-th MIT Conference on Advanced Research in VLSI, April 1986.

[13] IEEE Standards For Local Area Networks: 802.2, Logical Link Control, IEEE, NY, 1984.

[14] C.E. Leiserson, " Area Efficient VLSI Computation", MIT Press, Cambridge, Mass. 1983.