An Efficient, Low-Cost I/O Subsystem for Network Processors

Dionisios N. Pnevmatikatos and Ioannis Sourdis
Technical University of Crete

Kyriakos Vlachos
Bell Laboratories, Lucent Technologies

An efficient I/O subsystem enables cost-effective network processing. To improve high-speed data transfer, the I/O subsystem sends data directly into the processing core's register file. An implementation of this subsystem in a single-chip network processor, the Pro^3, can sustain advanced inspection firewall processing at 2.5-Gbps TCP traffic.

With the Internet's expansion and ever-increasing line speeds, executing the many different networking protocols is becoming the main bottleneck in high-speed communications. Gigabit Ethernet is already available, and products exist for 10-Gbps transfer rates. New bandwidth-eager software applications and faster processors in desktop and server systems place enormous demands on the current networking infrastructure. As a rule, network bandwidth use doubles every four months. Moreover, guaranteed quality and priority customization are on the way for many data, voice, and video applications.

To meet these stringent processing demands, designers can either use an ASIC to create a custom solution for their application or use a commercially available network processor.[1,2] The ASIC approach can achieve very high processing speeds but is inflexible, because changes in chipset behavior are either impossible or have limited support. This inflexibility is a significant problem, because applications and protocols continually evolve and extend to meet user-desired functionality. In addition, deploying an ASIC-based design requires more time than deploying a network-processor-based one.

Commercial network processors such as Intel's IXP (http://developer.intel.com/design/network/products/npfamily/ixp1200.htm) and Motorola's C-Port (http://e-www.motorola.com/webapp/sps/site/homepage.jsp?nodeId=03M0ylgx1Ks) are programmable,[2] so they can accommodate newer protocols. However, they represent a brute-force approach, using multiple (and simple) processing cores to achieve the desired processing performance levels. The developer must comply with several system restrictions, write a parallel program in a potentially heterogeneous environment, and meet all hard, real-time network-processing constraints. In addition, despite considerable progress, software development tools for such processors are still in their infancy.

Today's processors provide sufficient processing power to execute most, if not all, networking protocols. Despite their high processing potential, even fast workstations fail to sustain this processing even at 1 Gbps, because of their I/O limitations. To alleviate this bottleneck, we propose an intelligent I/O technique that offers two important advantages:

- It relieves the processing core of I/O duties, freeing resources to meet protocol processing needs.
- It facilitates application development. With minor modifications, existing tool chains such as compilers can use the added features.

We demonstrate the viability of this approach by implementing it with Pro^3, a single-chip network processor capable of delivering advanced firewall processing of transmission control protocol (TCP) packets at 2.5 Gbps.

Network processing fundamentals

A processor's basic functions correspond to the steps in Figure 1. Flow classification takes the header information (such as source and destination IP addresses) and produces a flow ID for indexing the connec-

0740-7475/03/$17.00 © 2003 IEEE Copublished by the IEEE CS and the IEEE CASS

tion-related internal data structures. Subsequently, these structures are necessary for obtaining information about this connection's state. This information and the selected header fields are usually sufficient for the protocol software to decide its course of action. This action usually involves forwarding the packet (possibly with modifications) to its destination, potentially creating more control (and possibly data) packets. This general framework for network processing can describe many applications, such as firewalls, routers, and gateways.

A general-purpose processor can perform most of these steps at a high speed but has limited pin bandwidth for transferring the header fields (and the packet information in general) and the state information. General-purpose processors work well for predictable programs for which caches offer low-latency access. In contrast, the behavior of protocol processing and packet data access presents limited locality, underusing the processor's abilities.

On the other hand, low-performance computation engines in today's network processors are limited in both processing and I/O potential. Although the actual processing portion of the code in Figure 1 is, for most applications, relatively short and manageable (involving just a register manipulation), the I/O portions are expensive: The locations present only spatial and no temporal locality, and the latency to collect all the data is quite high. However, using regular processing cores along with intelligent I/O can provide the required performance levels at low cost.

Pro^3 architecture

The Pro^3 system architecture accelerates execution of telecommunications protocols by extending a scalar reduced-instruction-set computer (RISC) core with programmable, pipelined hardware.[3-5] Pro^3 incorporates parallelism and pipelining wherever possible, and integrates generic microprogrammed engines with hardwired components optimized for specific protocol processing tasks. The system supports 2.5-Gbps links for up to 512,000 active connections, corresponding to 7.5 million packets per second in worst-case TCP traffic.

As Figure 2 shows, Pro^3 integrates a low-cost, power-efficient scalar RISC processor (the Hyperstone Electronics E1-32X RISC, http://www.hyperstone-electronics.com) in a reconfigurable processing module (RPM). Two RPM modules operate in parallel to allow the concurrent execution of incoming and outgoing packet processing. Dual RPMs also permit load balancing and result in higher throughput. A third Hyperstone processor handles system management and interfaces with an external computer system.

```
For each packet {
      1. Identify connection ID (flow), packet classification
      2. Get state information (last packet seen, and so on)
      3. Consult selected fields (parts of header and body)
      4. Execute protocol code on state and selected fields
      5. Update (?) packet and flow state
      6. Send (?) updated packet
      7. Create (?) other control packets
}
```

Figure 1. Pseudocode for abstracting a protocol's processing functions. (Question marks indicate optional steps.)

Reconfigurable processing module

In the Pro^3 project, we optimized each RPM module to perform packet processing. Each module contains a protocol processing engine (PPE), a field extraction (FEX) engine, and a field modification (FMO) engine. The FEX engine directly loads the required protocol data into the RISC processor for processing. The PPE interfaces the FEX and FMO engines to the integrated Hyperstone RISC. The FMO engine handles packet construction and header modification. These three engines form a powerful three-stage pipeline module that is the processing heart of the system (see the RPMs in Figure 2).

The FEX engine is a custom processor with a three-stage pipeline architecture. It is fully programmable, and operates protocol- or application-specific firmware. The FEX engine extracts all the fields needed for further processing from the received portion of the packet and forwards them to the PPE.

FMO complements FEX, adopting a three-stage pipeline and a programmable operation. FMO composes protocol messages, taking as input the processed results (fields) from the PPE, the original packet data from the delay FIFO buffer, and commands from the protocol execution.

The FEX, PPE, and FMO components can process data with a maximum throughput of 6.4 Gbps. A 32-bit-wide data path and a 200-MHz clock frequency (system clock) make this possible.

PPE architecture

The PPE uses an intelligent I/O subsystem to offload the transfer of packet data and state information from the processor core. Figure 3 shows the PPE architecture, which is based on the assumption that it is possible to

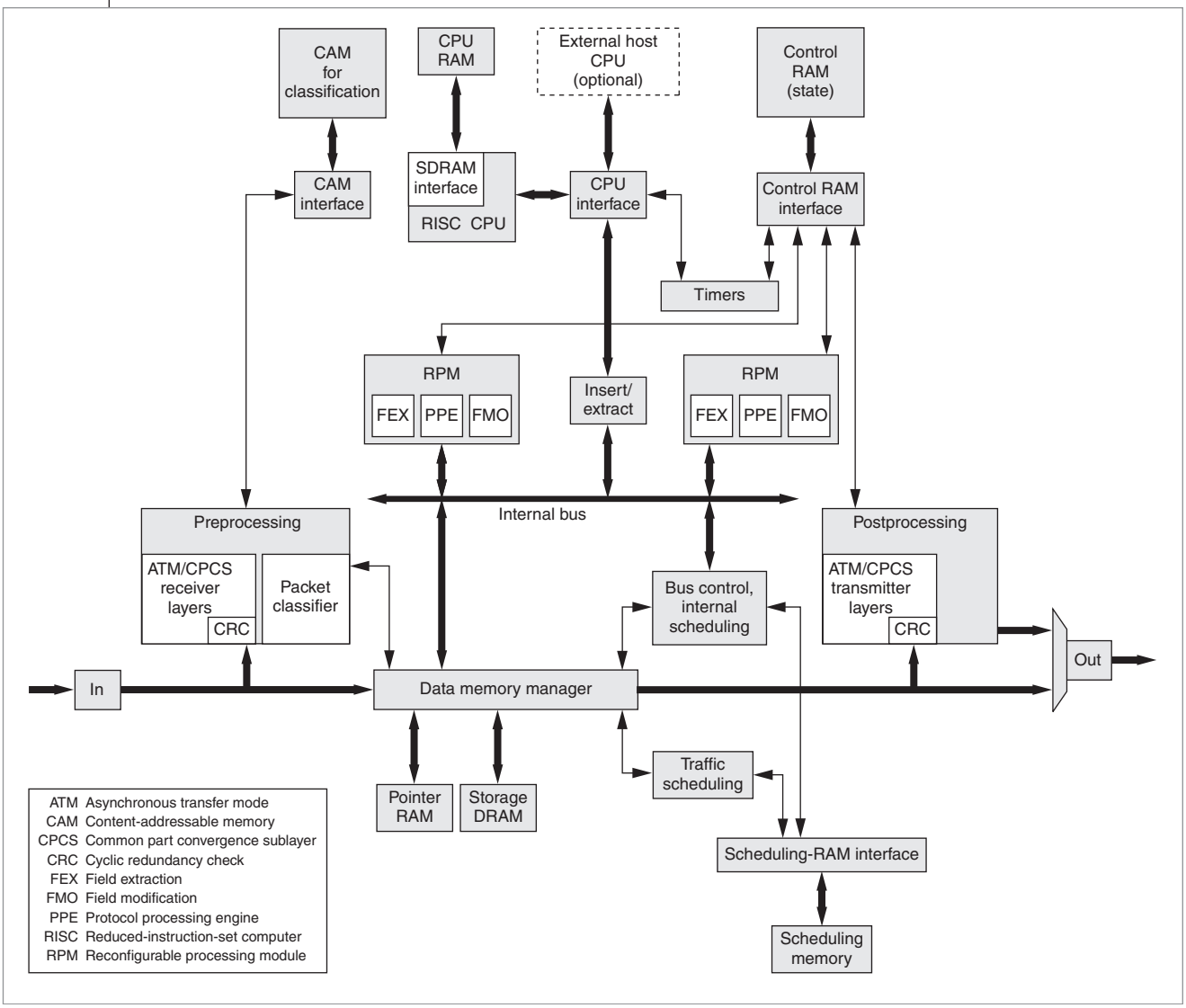

Figure 2. Pro^3 architecture.

modify the processor core to provide additional ports for the register file. The control logic reads the incoming packet data from the transceivers and passes it directly to the processor registers. The state control logic matches the incoming packet with the flow state, reads the required fields from memory, and again passes the information to the processor. This assumes the flow classification step has already occurred, using either hardwired blocks or content-addressable memories (CAMs). When all the necessary information is available, the processor commences protocol processing in its registers. The results are also stored in registers, and the control logic extracts the data and transmits it, if necessary, to the FMO engine.

This architecture has two main advantages:

- It offloads the processor, which then doesn't need to execute load and store instructions for state and packet I/Os.
- It creates a high-level, three-stage pipeline between input-packet data transfer, processing, and output data packets. This pipeline allows overlap of these tasks and leads to better overall performance.

Issues that the architecture still needs to address include the following:

- Synchronization between input, processing, and output in light of pipelined packet processing is crucial for correctness, because all these processes share the processor's register file.

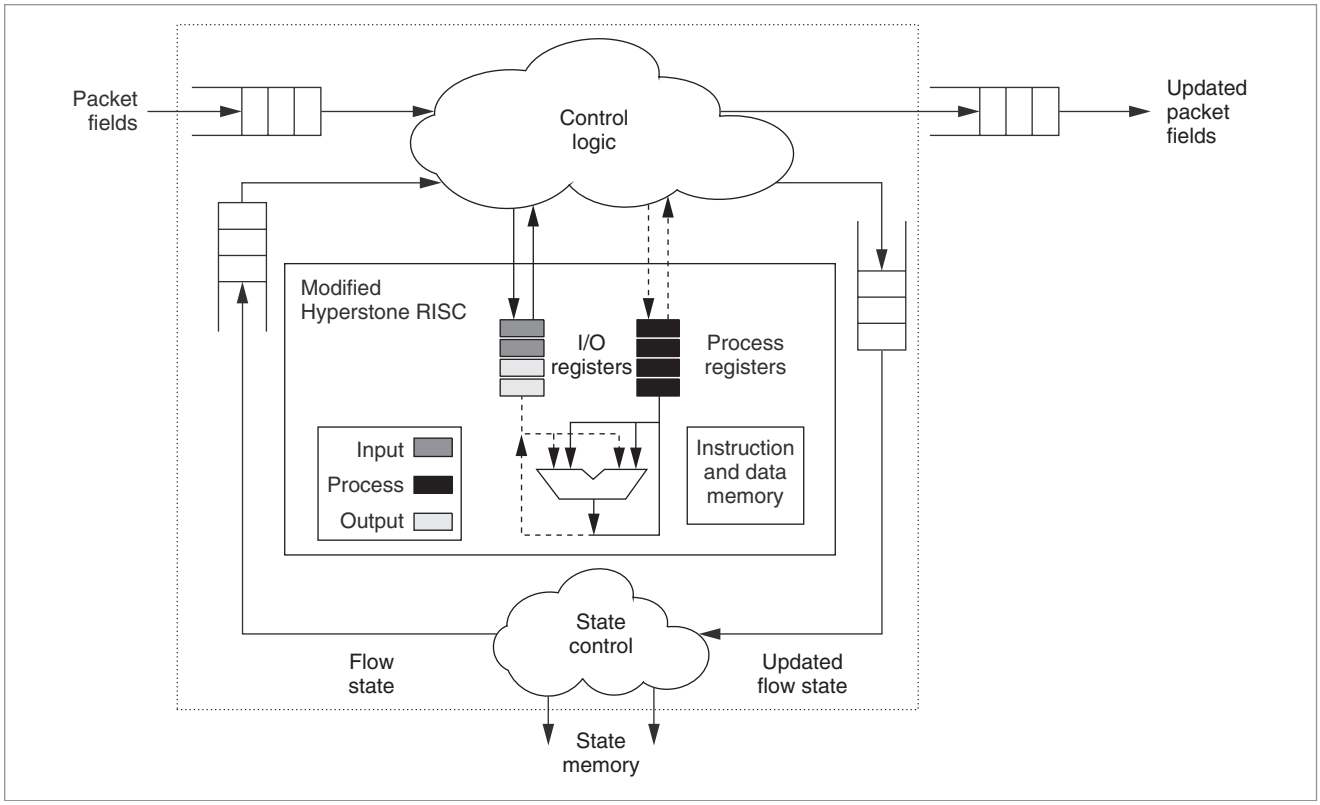

Figure 3. Protocol processing engine (PPE) architecture.

- The total number of registers is a potential limit of the architecture.
- A powerful hardware-software interface is needed to interact with the control logic's I/O portions and transfer the data packets.

Fortunately, solutions for all these issues already exist. We synchronize the pipelined execution of input, processing, and output stages in the control logic. To improve performance, we also provide bypassing of modified flow state information from processing and output stages to the input stage.

The number of registers in the architecture represents a tradeoff: Larger register files are slower but allow larger data transfers without delay. We analyzed the TCP, user datagram protocol (UDP), and asynchronous transfer mode (ATM) signaling protocol, and found that 32 registers are sufficient to hold the necessary state information and the packet header. More registers are necessary to process the packet body, but with pipelining the architecture can process 32 registers at a time. Today's processors contain more than 32 registers (for the use of renaming), and 64 registers can be reasonably fast. We overlap the use of registers between the I/O functions to reduce the register file's size. In our simulations, we assume that two sets of 32 registers handle packet processing and I/Os, and another 32 handle program variables and computations.

Hardware-software interface

For successful operation, the hardware and software must coordinate in two ways:

- upon packet input, the hardware must notify the software to initiate packet processing; and
- upon processing completion, the software must notify the hardware to output the processed packet.

For the first requirement, we devised a simple handshake scheme: A signal from the control logic notifies the processor that a new packet is available for processing; the processor maintains an Idle signal, which informs the control logic that it is not processing any packet. Deactivation of the Idle signal indicates the beginning of packet processing. Reactivation of the Idle signal indicates the completion of processing, and the processed packet can be transmitted to the FMO module.

For efficiency reasons, we do not use interrupts to

Table 1. Fields for the software result register.

Field	Description
Length	Number of words to transfer
Start register	Number of the first register
MoreFields	More fields follow (pipelining)
Type	Type of packet (internal)
MemPacket	For storing large or multiple packets in memory

initiate packet processing. The processor waits for packets, using the following instruction sequence:

```
        move R4, Wait
Wait:   jump_register R4
```

This is an infinite loop, but after transferring the packet data, the hardware control logic overwrites R4 with the packet handler's address. So the loop ends, and packet processing begins. A hardware dispatch table maintains each packet's starting address.

On the output side, the interface is more involved. To output a packet, the control logic must know its size, where to send it, and other similar information. Moreover, the result of processing can be multiple packets, not just one. To handle these cases, we defined a format for a *software result register*. This is the first register that the output logic reads, and it defines all subsequent actions.

The format divides the register into the fields listed in Table 1.

The output control logic interprets these fields. For multiple output packets, the interface uses one software result register per packet.

The proposed hardware-software interface is register based and easily expressible in software with a function (or system) call that on return updates the appropriate registers. Therefore, even with the defined interface, designers can use traditional compiler tools to develop, optimize, and debug the application code. This advantage can be crucial to timely product development.

PPE implementation

Figure 4 shows the PPE module, which includes three units: the modified Hyperstone (MHY) RISC core, the read-write control RAM (RWR), and the RPM glue logic (RPG). The input module transfers packets and flow state into MHY's register file for processing. The output module transfers the process results (fields, flow state, and commands) to the field modifier module. The control and monitoring module (CMM) is responsible for monitoring and debugging, and initializes all internal structures (the dispatch table, and so on).

The input module also supports internal-state bypass, detecting the location of the latest version of flow state for each incoming packet. Because processing is pipelined, the correct state information is available in either the MHY register file (just updated from the previous packet), the FIFO RWR buffers (the state was updated but is not yet written to memory), or the FIFO state buffer. The RWR module performs three major tasks. For each packet, it

- reads the appropriate state information from the control RAM and provides it to the RPG's input submodule,
- receives the updated state from the newly processed packet and writes it to the control RAM, and
- acts as a searchable write buffer to ensure that reading the control RAM always provides the correct results.

We implemented all PPE modules except the RISC processor in VHDL. The modified Hyperstone RISC came as a VLSI hard-macro, along with a gate-level model for simulations. The Pro^3 performance target was an operating frequency of 200 MHz, sufficient for achieving line-speed processing at 2.5 Gbps. We designed Pro^3 using semicustom logic for all control and storage, and memory blocks for large buffers. We synthesized the design using UMC 0.18-micron technology, achieving the target operating frequency after minor design modifications. We then incorporated this design into the Pro^3 chip, which UMC fabricated. We are currently testing the chip, and we have already verified the correct operation of the internal processing pipeline. We are also testing the I/O transceivers; once they've proven functional, we will proceed with wire-speed full-system testing.

PPE evaluation

Our evaluation addresses the architecture's processing efficiency and implementation cost. We used VHDL simulations to obtain the actual number of useful cycles and the number of stalled cycles for various input packets (of differing lengths) and total processing times. We assumed 10 active IP connections, with processing times from 20 to 60 cycles. We randomly selected 20% of the packets to produce two responses. We also varied the control RAM's response time to between 5 and 10 cycles to model contention between multiple processing

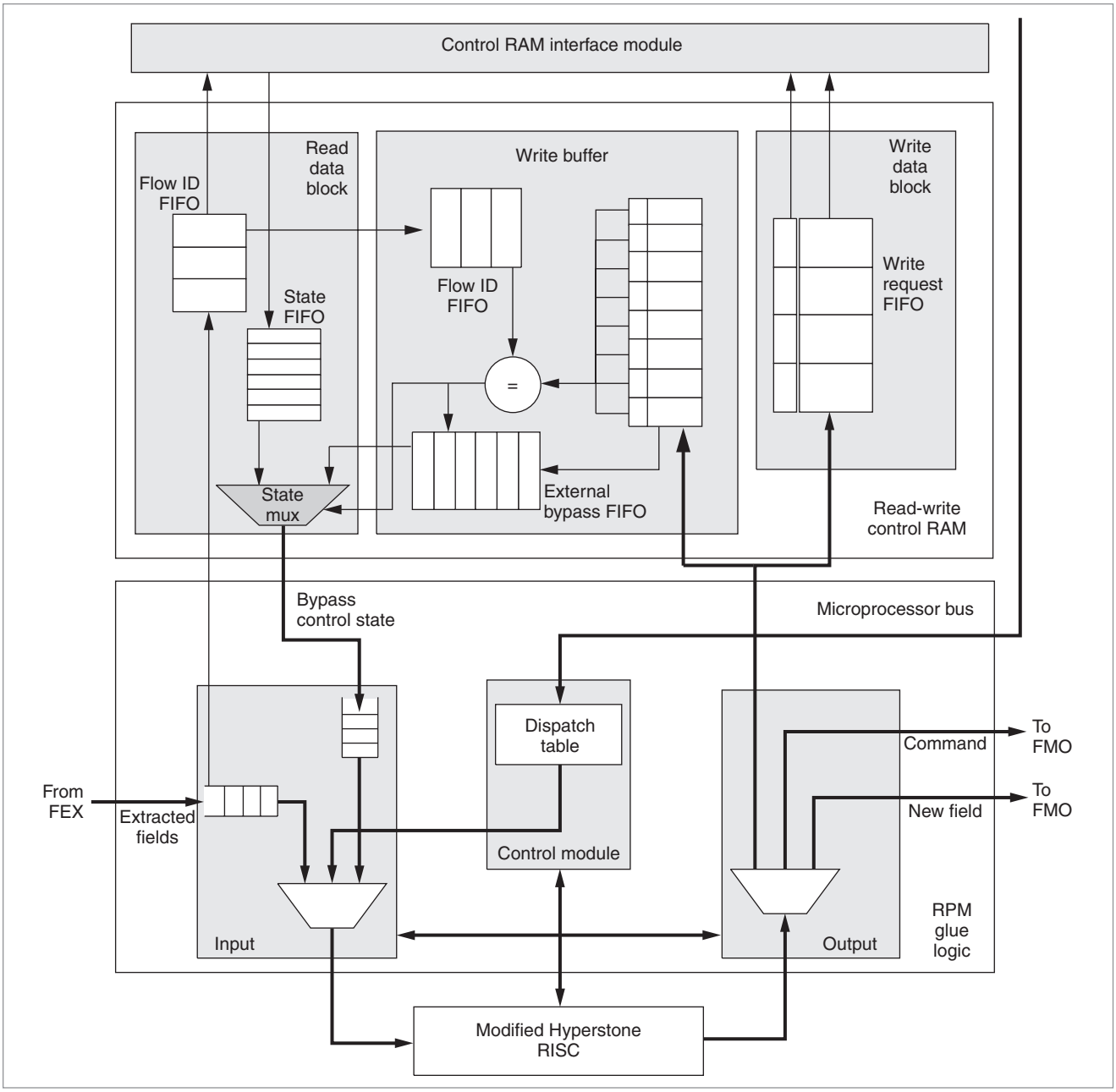

Figure 4. PPE microarchitecture.

engines, and we simulated the system for packet sizes of 40, 80, and 120 bytes (10, 20, and 30 32-bit words).

Table 2 gives the results of these simulations. The first portion shows the packet latency: the number of cycles required to transfer a packet and execute the protocol code on it. The second portion shows the corresponding efficiency (percentage of time in which the PPE performs useful work) compared to the bare processing time (excluding the time to transfer data to and from the processor).

These results indicate that the hardwired I/O mechanism is effective when the computation cost is high. This situation is usually the case, because 40 or 50 instructions correspond to highly tuned protocol code. In several cases, the cost can be higher, further improving the architecture's efficiency. The pipeline's operating efficiency improves as the processing time for the packet grows, meaning that for longer processing times, a larger fraction of latency—from I/O packet and state data transfer—remains hidden; thus, the system essen-

Table 2. PPE latency and processing efficiency for various protocol processing durations.

Protocol process duration (cycles)	Packet latency (cycles)			Processing efficiency (%)		
	40-byte packets	80-byte packets	120-byte packets	40-byte packets	80-byte packets	120-byte packets
20	27	42	42	75	47	47
30	33	40	42	91	75	71
40	43	44	45	93	91	89
50	53	53	53	94	94	94
60	63	63	63	95	95	95

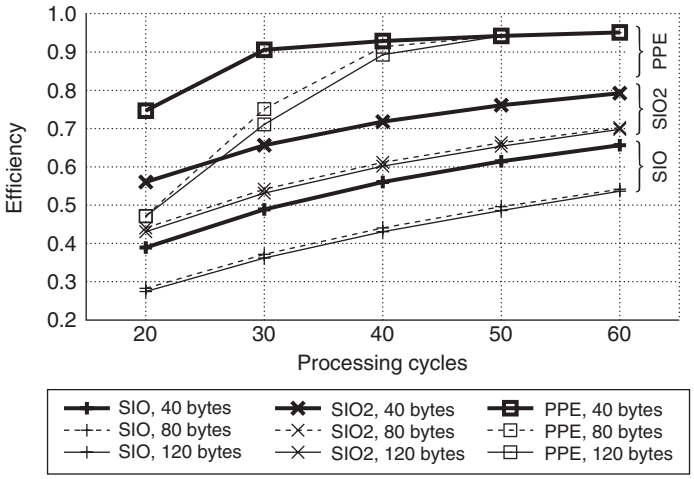

Figure 5. Efficiency versus protocol processing duration for PPE, serial I/O (SIO), and optimized serial I/O (SIO2) models.

tially performs these operations for free.

These results are encouraging but do not address the optimality of our approach compared to other architectural alternatives. To address this issue, we defined two simple architectures. The first is serial I/O (SIO); it operates exactly like the PPE but performs the input, processing, and output operations sequentially. We assume memory-mapped FIFO interfaces that are read and written with load and store instructions. In this way, SIO models an unmodified processor integrated into the processing pipeline.

The second architecture alternative is an optimized version of SIO, called SIO2, which models a more-aggressive implementation that loads only the useful header and body fields. In SIO2, the processor reads or writes only half the header fields and state information. However, to support random access between fields, such an architecture requires storing packets in memory and not in FIFO buffers. Neither SIO nor SIO2 models memory latency, contention, or FIFO data availability; they assume that data is always available when needed. Therefore, these models represent an unrealistic best case for sequentially accessing and processing packet data.

To compare the PPE approach with these models, we performed a series of experiments. These experiments varied the process duration from 20 to 60 cycles, and we simulated packet sizes of 40, 80, and 120 bytes. The results in Figure 5 clearly show that the PPE approach is successful in overlapping the I/O operations required for packet processing. SIO is clearly worse, in most cases achieving an efficiency of around 50%. SIO2 makes fewer references to the memories. SIO2 improves on SIO's performance, achieving an 80% best efficiency, but only for large packets and processing times. However, neither models include any of the overheads for transferring data to and from the pipeline. If we had modeled these overheads, the performance difference between PPE and SIO or SIO2 would have been even greater.

The next question is, How much must we spend to achieve this performance? To answer this question, we synthesized the HDL code for the design's I/O portions. We do not include the processing core's gates and area, because we want to measure only the extension's cost. We used a general 0.18-micron library and the Synopsys dc_shell to perform the synthesis. Table 3 gives the results, which include all structures needed by our architecture but exclude the cost of changes in the processing core. This cost is relatively difficult to measure because it varies depending on the core's architecture. The cost, however, includes the extra registers in the register file (64 in our simulations) and the implementation of the hardwired handshake mechanism for notifying the hardware of processing completion.

ARCHITECTURES USING an approach similar to ours include the Pipe processor and the iWarp processor. The

Pipe processor uses two register-mapped queues to synchronize the producer-consumer relationship in its decoupled architecture.[6] The iWarp processor also uses a register-mapped network interface to efficiently communicate with its systolic peers.[7] We use a similar handshaking mechanism to efficiently wake up the software, but we advocate the use of many registers to efficiently present the packet data to the application code.

Several network-processing architectures are available today. The network processor that exhibits the most similarities to our work is Intel's IXP. In that system, each microengine can initiate a transfer of a 64-byte block from the main chip buffer that holds the packets. However, each microengine supports up to four threads and incurs thread scheduling and synchronization overheads. Furthermore, compilers cannot easily handle this complex structure, leaving the programmers to explicitly manage it. Our approach targets simpler processing cores and is easier to integrate into existing compilers.

Another approach for overlapping I/O and processing is possible via simultaneous multithreading (SMT), in which different threads share the multiple execution resources of an instruction-level-parallel processor.[8] Indeed, such an architecture can achieve similar benefits to those of our approach. However, SMT architectures require a significantly more-complex processor architecture to support instruction-level parallelism and to simultaneously support multiple threads. Therefore, an SMT processor's implementation cost is far higher than that of our simple circuits. ■

Table 3. Synopsys synthesis results.

No. of cells	Combinational area (μm^2)	Area for memory and flip-flops (μm^2)	Total area (μm^2)
10,028	136,974	933,235	1,070,206

Acknowledgments

We conducted this work under a Lucent subcontract in the context of the IST-Pro[3] project. We thank Jorge Sanchez, Nikos Nikolaou, and Kostas Pramataris for their contributions to the development of the PPE architecture.

■ References

1. N. Cravotta, "Network Processors: The Sky's the Limit," *EDN*, 24 Nov. 1999, pp. 108-119; http://www.e-insite.net/ednmag/.
2. P.N. Glaskowsky, "Network Processors Mature in 2001," *Microprocessor Watch*, no. 93, 21 Mar. 2002; http://www.mdronline.com/publications/mpw/issues/mpw093.html.
3. G. Konstantoulakis et al., "A Novel Architecture for Efficient Protocol Processing in High-Speed Communication Environments," *Proc. 1st European Conf. Universal Multiservice Networks*, IEEE Press, 2000, pp. 425-432.
4. C. Georgopoulos et al., "A Protocol Processing Architecture Backing TCP/IP-Based Security Applications in High-Speed Networks," *Proc. 5th IFIP TC6 Int'l Symp.* (Interworking 00), *Lecture Notes in Computer Science*, vol. 1938, Springer Verlag, 2000; http://www.telenor.no/fou/om/konferanser/presentation.shtml.
5. N. Nikolaou et al., "Application Decomposition for High-Speed Network Processing Platforms," *Proc. 2nd European Conf. Universal Multiservice Networks*, IEEE Press, 2002, pp. 322-329.
6. M.K. Farrens and A.R. Pleszkun, "Implementation of the PIPE Processor," *Computer*, vol. 24, no. 1, Jan. 1991, pp. 65-70.
7. S. Borkar et al., "Supporting Systolic and Memory Communication in iWarp," *Proc. 17th Ann. Int'l Symp. Computer Architecture* (ISCA 90), IEEE CS Press, 1990, pp. 70-81.
8. D.M. Tullsen, S.J. Eggers, and H.M. Levy, "Simultaneous Multithreading: Maximizing On-Chip Parallelism," *Proc. 22nd Ann. Int'l Symp. Computer Architecture* (ISCA 95), ACM Press, 1995, pp. 392-403.

Dionisios N. Pnevmatikatos is an associate professor of electrical and computer engineering at the Technical University of Crete and a research associate at the Institute of Computer Science, Foundation for Research and Technology-Hellas. His research interests include the architecture and design of computer systems, networking systems, and system software. Pnevmatikatos has a BS in computer science from the University of Crete, Greece, and an MSc and PhD in computer science from the University of Wisconsin-Madison. He is a member of the IEEE and the ACM.

Ioannis Sourdis is an MSc candidate in the Department of Electronic and Computer Engineering at the Technical University of Crete. His research interests include the architecture and design of computer systems, networking

systems, reconfigurable hardware, and network processing using FPGAs. Sourdis has a diploma in electronic and computer engineering from Technical University of Crete.

Kyriakos Vlachos is a member of the technical staff at Bell Labs, Lucent Technologies. His research interests include broadband access networks, optical packet switching, and network processors. Vlachos has a Dipl-Ing and PhD, both in electrical and computer engineering from National Technical University of Athens. He is a member of the IEEE.

■ Direct questions and comments about this article to Ioannis Sourdis, Microprocessor and Hardware Laboratory, Electronic and Computer Engineering Department, Technical University of Crete, Chania, Crete, GR 73 100, Greece; sourdis@mhl.tuc.gr.