A Network Processor-Based, Content-Aware Switch

A CONTENT-AWARE SWITCH CAN OPTIMIZE CLUSTER-BASED SERVER ARCHITECTURES BY EXAMINING REQUESTS AND DISTRIBUTING THEM TO SERVERS ON THE BASIS OF APPLICATION-LEVEL INFORMATION. A NETWORK-PROCESSOR-BASED SWITCH CAN REDUCE HTTP PROCESSING LATENCY AND IMPROVE PACKET THROUGHPUT.

Li Zhao, Yan Luo, and Laxmi N. Bhuyan
University of California, Riverside

Ravi Iyer
Intel Corp.

With the explosive growth in Internet traffic, requests have been overloading Web servers. Many ISPs and search engines employ a server cluster to build a cost-effective, scalable, and reliable server system. To make such a distributed-server system transparent to clients, architects usually place a switching device with one virtual Internet Protocol (VIP) address in front of the server cluster as a common interface.

The switch routes packets on the basis of Layer-5 information such as request content or application data. Figure 1 shows such a Layer-5 or *content-aware switch*[1-3] for a server cluster. Compared with a traditional Layer-4 switch, a content-aware switch has the following advantages:

- *Better load balancing.* The switch directs incoming requests on the basis of content type, such as static or dynamic, to dedicated servers optimized for a particular type.
- *Faster response.* Servers maintain in-cache copies of recent service results, so sending requests to servers that have already satisfied the same request can exploit cache affinity and reduce latency.
- *Better resource use.* Servers can be dedicated to an application type, such as static HTML pages or database transactions, making it possible to partition Web content to the different servers to eliminate replication and use resources efficiently.

We designed and implemented a content-aware switch based on an ENP2611 board that contains an Intel IXP2400 network processor (NP). Our performance evaluation results show that this switch significantly improves processing latency as well as throughput. To the best of our knowledge, no other study has focused on the design of content-aware switches based on NPs (see the "Related Work" sidebar).

Overview of content-aware switches

Content-aware switches built based on application-specific integrated circuits[1-3] (ASICs) or general-purpose processors[4-6] have been prevalent for the last 10 years. However, although ASIC-based switches can achieve high-processing capacity, they lack flexibility and programmability. Moreover, switches based on general-purpose processors can't provide satisfactory performance, because of interrupt and moving packets over the peripheral component interconnect (PCI) bus. Using NPs that operate at the protocol's link layer, as with ASICs, solves these problems

Published by the IEEE Computer Society

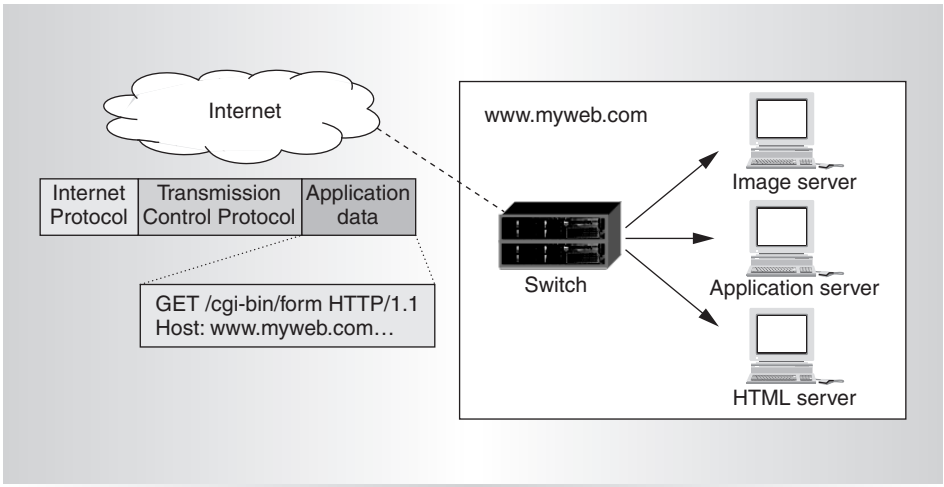

Figure 1. A server cluster with a content-aware switch at the front end.

Related Work

Researchers have studied content-aware switches extensively. Cohen et al.[1] implement a content-aware switch in Linux using TCP splicing. They use an application-level proxy to determine the destination server on the basis of client requests. Yang et al.[2] move all the processing down to the Linux kernel, so that the switch performs all the data forwarding as well as the routing decisions at the kernel level. This move can avoid the overhead of passing the HTTP request packet through the protocol stack to the user-level proxy, as Cohen's team has done. Our approach, implemented on NPs, moves the whole processing further down to the NIC level, thus reducing the end-to-end latency as much as possible. Apostolopoulos et al.[3] built a content-aware switch based on a switch core with custom, intelligent, port controllers and a PowerPC processor. As an application-specific integrated circuit (ASIC) design, this switch can achieve very high throughput. However, we can hardly extend it to incorporate new services such as quality of service scheduling. In addition to ASICs, we can use field-programmable gate arrays to speed up pattern matching in the content-aware switches.[4] However, FPGAs have higher power consumption compared to NPs.

Spalink et al.[5] suggest separating TCP splicing processing on a data forwarder and a control forwarder, which run on the IXP2400 microengines and the host processor (a Pentium), respectively. However, our analysis shows that performing all the processing on the microengines gives better performance. Therefore, we put not only the data forwarder, but also the control forwarder on the microengines.

Besides TCP splicing, TCP hand-off[6] is another mechanism for building a content-aware switch. To reduce the switch's load, TCP hand-off lets the response from the server reach the client directly without going through the switch. This approach requires modifying the TCP state machine in servers' operating system. This would be impractical for large-scale server clusters. Papathanasiou et al.[7] exploit both the TCP splicing and hand-off techniques on a Web switch. The switch performs TCP splicing, whereas back-end servers perform the hand-off operation. Their approach requires that a proxy application run on each of the back-end servers, but it doesn't require any modification to the operating system.

References

1. A. Cohen, S. Rangarajan, and H. Slye, "On the Performance of TCP Splicing for URL-Aware Redirection," *Proc. 2nd Usenix Symp. Internet Technologies and Systems* (USITS 99), Usenix, 1999, http://www.usenix.org/events/usits99/cohen.html.
2. C. Yang and M. Luo, "Efficient Support for Content-Based Routing in Web Server Clusters," *Proc. 2nd Usenix Symp. Internet Technologies and Systems* (USITS 99), Usenix, 1999, http://www.usenix.org/events/usits99/yang.html.
3. G. Apostolopoulos et al., "Design, Implementation and Performance of a Content-Based Switch," *Proc. 19th Joint Conf. IEEE Computer and Comm. Societies* (INFOCOM 00), IEEE Press, vol. 3, 2000, pp. 1117-1126.
4. Tarari Inc., Regular Expression Content Processor, http://www.tarari.com/regexEAP/index.html.
5. T. Spalink et al., "Building a Robust Software-Based Router Using Network Processors," *Proc. 18th ACM Symp. Operating Systems Principles* (SOSP 01), ACM Press, 2001, pp. 216-229.
6. V. Pai, et al., "Locality-Aware Request Distribution in Cluster-based Network Servers," *Proc. 8th Conf. Architectural Support for Programming Languages and Operating Systems* (ASPLOS 98), ACM Press, 1998, pp. 205-216.
7. A. Papathanasiou and E. Hensbergen, "KNITS: Switch-Based Connection Hand-off," *Proc. 21st Joint Conf. IEEE Computer and Comm. Societies* (INFOCOM 02), IEEE Press, vol. 1, 2002, pp. 332-341.

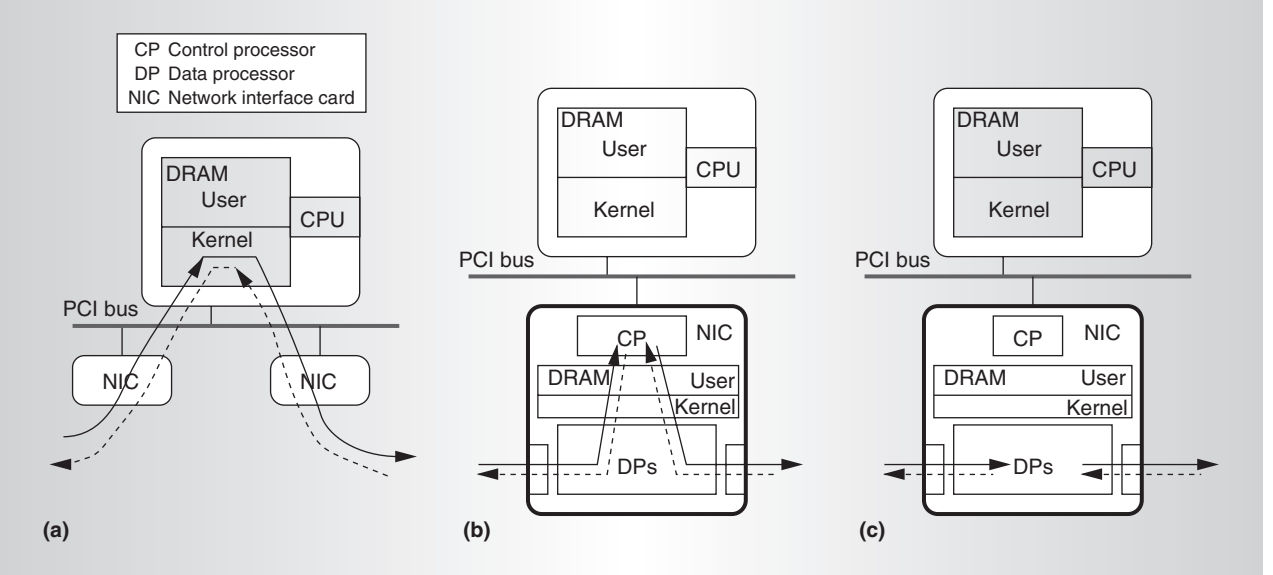

Figure 2. Three architecture candidates for a content-aware switch: Linux-based interface in which processing occurs in the kernel space (a); network-processor-based interfaces in which processing occurs in the control processor (b) as well as in the data processor (c).

and avoids the large overhead associated with general-purpose processors. NPs are also programmable, so they can achieve the same flexibility as general-purpose processors. In addition, NPs have an instruction-set architecture optimized for packet processing, and their hardware is usually equipped with multiprocessing and multithreading, which can provide good throughput.

A content-aware switch is usually built using an HTTP proxy[4-6] running on the application level. The proxy maintains connections with the client and the server separately and forwards data between these two connections. Although this approach is easy to implement, copying data between these two connections results in high overhead. We solve this copying problem through Transmission Control Protocol splicing, which splices the two connections after both are established.[7,8] The switch can then forward subsequent data packets on the spliced connection by modifying particular fields (for example, sequence numbers) in their TCP and Internet Protocol headers. This data forwarding occurs at the IP level, thereby avoiding the overhead of copying data between the user space and the kernel space.

Implementing a content-aware switch using an NP isn't simple. NPs are programmed at a low-level language (microC, or microcode without a compiler to directly translate C code to this language. Moreover, an NP has limited instruction memory, so reducing and optimizing the existing code requires enormous effort.

We studied several design options for a content-aware switch built with an NP, and we found key improvements that an NP-based switch can provide over a Linux-based switch. We also analyzed in detail the TCP splicing technique and derived a splicing protocol for NP, called SpliceNP. Finally, we carefully allocated the workload among the NP's resources for optimal performance.

Design options

Designers have built content-aware switches in Linux machines by inserting loadable kernel modules into the operating system.[6,9] In these switches, the proxy, which runs at the application level, parses HTTP requests. Hence, the request packet must go through the protocol stack and then be copied from the kernel space to the user space. It's possible to reduce latency by removing the proxy and moving all the processing, including parsing the HTTP request, into the kernel space, as Figure 2a shows.[6,9] However, the switch must move the data from the host DRAM to the network interface card (NIC) and from the NIC to the host DRAM over the PCI bus.

This traffic imposes heavy bandwidth pressure on the PCI bus when the number of connections is large. It also introduces interrupt overhead to the host CPU.

To further improve switch performance, we propose moving all processing down to the NIC level. Figures 2b and 2c use NP-based network interfaces. The NP usually has one *control processor* and multiple *data processors*. Data processors are tuned specifically for processing network packets in the fast path, whereas control processors help maintain the control information and processing exception packets. For example, Intel's IXP2400 contains one control processor (XScale) and eight data processors (called microengines, or MEs, in Intel's IXP2400). The control processor runs an embedded Linux operating system and shares DRAM with data processors. The data processors receive and transmit packets through NICs.

In Figure 2b, the TCP stack in the control processor can create connections to clients and servers and then splice these two connections in the embedded Linux kernel. The switch can process on the MEs the packets sent after splicing. Many industrial projects use this implementation for offloading the TCP to an NP. However, splicing the processing in the XScale increases latency because the XScale must poll an input queue (that the MEs fill) to retrieve packets. In addition, after processing these packets, the XScale must put them in an output queue. The ME sends out the packets from there. The packet polling time, enqueueing, and dequeueing together increase processing latency. (Notice that processing these packets falls within the critical path for TCP splicing.) This delay is detrimental to overall performance because longer delays might cause time-outs for clients and lead to packet retransmissions. In effect, this technique replaces Linux with embedded Linux and a powerful Pentium CPU with a weak XScale CPU. Nonetheless, we implemented this technique and observed that it increases, rather than decreases, latency.

Because of the large number multiple MEs and threads in NPs, Figure 2c is a natural evolution over Figure 2b. After receiving packets from NICs, the packet processors create and splice the connection, and forward the data without needing to communicate with the control processor or the host CPU. The multiple hardware threads in packet processors are capable of fast packet processing for multiple connections simultaneously, and eliminating data copying though the PCI bus. We used this architecture to design and implement our content-aware switch. However, implementing complex splicing software in an ME is difficult. Unlike the XScale control processor, MEs are programmed in microC (instead of C) and have limited control memory. Compared with the Linux-based switch in Figure 2a, an NP-based switch can reduce processing latency in four ways:

- *Interrupt versus polling.* When the NIC in the Linux machine receives packets, it raises an interrupt to the CPU. Although current NICs can accumulate multiple packets and then notify the processor using a single interrupt, the interrupt's overhead is still high. To reduce this overhead, NPs use polling instead of interrupts.
- *NIC-to-memory copy versus no copy.* In the Linux-based switch, the NIC must copy the received packets to the main memory, which requires a direct-memory-access (DMA) transfer through the PCI bus. Similarly, when the processor sends out the packets, they are transferred from the memory to the NIC buffer through DMA again. In an NP-based switch, however, packets are processed inside the NIC without the need for copying.
- *Linux processing versus IXP processing.* Even if the switch implemented the entire TCP splicing in the Linux kernel, there would still be operating-system overhead, as there is with a content-aware switch. In NP, the optimized instruction-set architecture for packet processing lets us process packets more efficiently and reduce the number of instructions executed. For example, we can load an IP or TCP header in one instruction and hide the memory latency by switching to other threads.
- *General-purpose processor versus multiple MEs and threads.* The multiple MEs and threads in an NP can process many packets in parallel, thus increasing throughput.

The first two factors involve significant packet processing. To measure the time taken

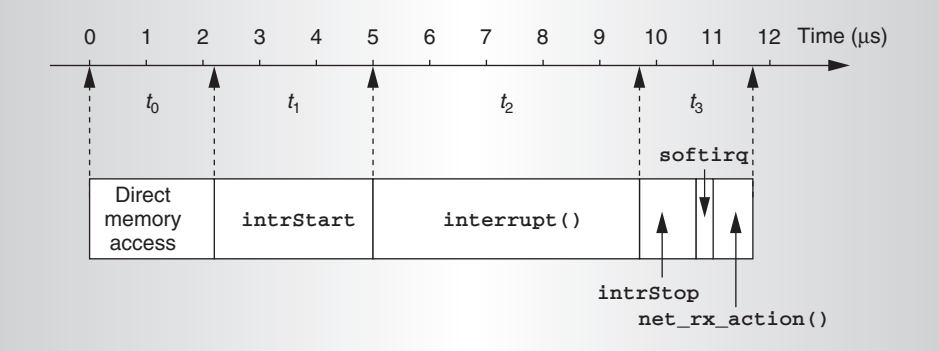

Figure 3. Direct-memory-access and interrupt handling for receiving a 41-byte packet through TCP/IP in a Linux machine.

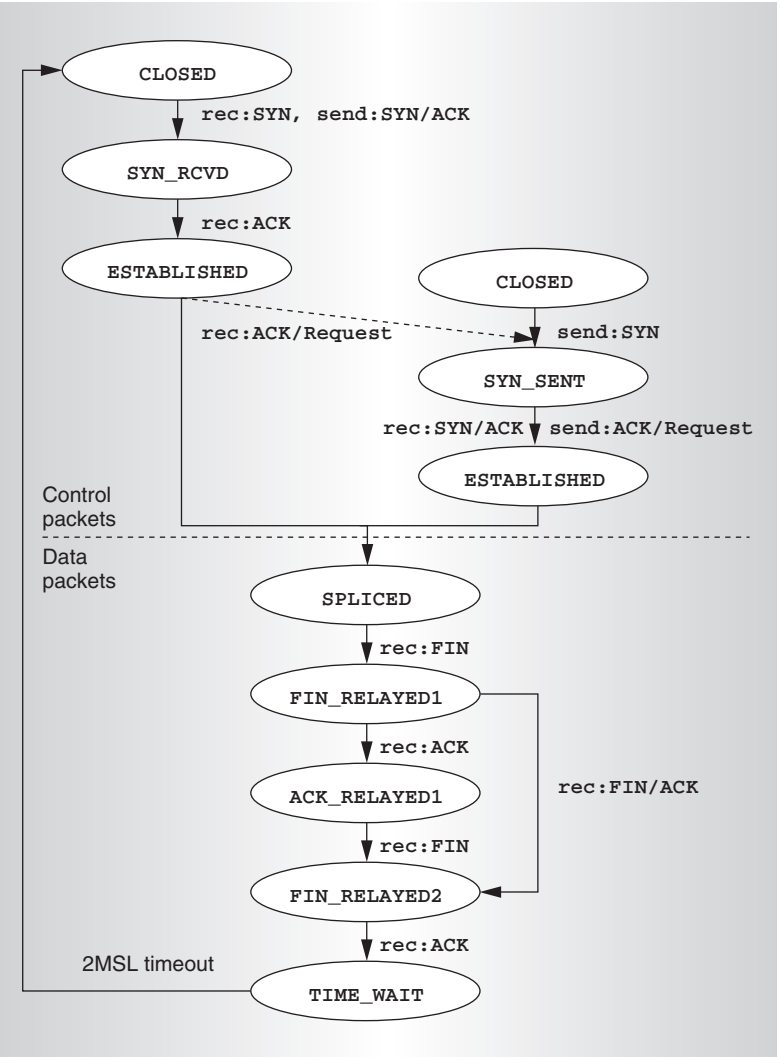

Figure 4. State transition in TCP splicing.

by the various kernel parts, we used a Linux PC with a Pentium CPU running at 400 MHz and equipped the kernel code with instructions that read the time-stamp counter for the functions we were interested in. Figure 3 shows the time line for receiving a 41-byte message using TCP/IP. After the NIC copies the packet through DMA to the host memory (t_0), the interrupt handler (t_1) saves all the CPU registers on the stack and invokes the NIC interrupt service routine tulip **`interrupt()`** (t_2), which raises a soft interrupt. When the interrupt handler **`intrStop`** ends, the CPU executes the soft interrupt **`softirq`**, which calls **`net_rx_action()`** (t_3). This prompts the network layer function **`ip_rcv()`** to initiate TCP/IP stack processing. The DMA and interrupt handling (t_0 through t_3) in the Linux machine took about 11.7 microseconds (µs), whereas this part is almost negligible in an NP implementation that employs Figure 2b or 2c.

TCP splicing technique

As we mentioned, TCP splicing eliminates the overhead of copying data between the kernel and the user space. We modified the TCP splicing protocol to suit our context-aware switch.

TCP splicing state transition

TCP splicing is based on standard TCP/IP, which maintains each connection's state information.[10] Therefore, we also must maintain state information. To implement TCP splicing, we must clearly understand various states of this process, as the state transition diagram in Figure 4 shows. The state transition from **`CLOSED`** to **`SYN_RCVD`** and **`ESTABLISHED`** states is the three-way handshaking standardized in TCP/IP protocol. Figure 4 shows two three-way handshakings. The client initiates the first handshaking, which leads to **`ESTABLISHED`** state for the first connection. Receiving an HTTP request packet triggers the second handshaking. After the second connection also enters **`ESTABLISHED`** state, the switch splices together these two connections, and the state migrates to **`SPLICED`**. The spliced connection's termination starts when a

Table 1. Processing a SYN packet.

Step	Functionality	TCP	Linux-based	NP-based
1	Dequeue packet	Yes	Yes	Yes
2	IP header verification	Yes	Yes	Yes
3	IP option processing	Yes	Yes	No
4	TCP header verification	Yes	Yes	Yes
5	Control block lookup	Yes	Yes	Yes
6	Create new socket and set state to `LISTEN`	Yes	Yes	No socket, only control block
7	Initialize TCP and IP header template	Yes	Yes	No
8	Reset idle time and keep-alive timer	Yes	Yes	No
9	Process TCP option	Yes	Yes	Only maximum segment size option
10	Send ACK packet, change state to `SYN RECEIVED`	Yes	Yes	Yes

FIN packet is received from one side (server or client). This leads to the state transition to **`FIN_RELAYED1`**. When the ACK (acknowledgment) to this FIN (finish)packet is received, the state becomes **`ACK_RELAYED1`**. The arrival of a FIN packet from another side changes the state to **`FIN_RELAYED2`**. The last **`ACK`** packet leads the state to **`TIME_WAIT`**. After 2MSL (maximum segment lifetime, which is usually 120 seconds[10]), the state transitions to **`CLOSED`**. Notice that data forwarding occurs from **`SPLICED`** state until **`FIN_RELAYED2`**.

Based on this state diagram, we classify packets into two types: *control packets* and *data packets*. Control packets are those the client or server sends before the two connections are spliced. The switch uses these packets, such as SYN (synchronization) packets, to set up connections. We also treat the HTTP request packet as a control packet because it triggers the second connection. Data packets are response packets the server sends, ACK packets the client send and FIN packets that both the server and client sends after the switch splices the two connections.

The SpliceNP protocol

To build an NP-based switch, we simplified the splicing protocol in comparison to the original TCP protocol, because the TCP code must handle every situation (some of which aren't needed for the switch). Also, to fit the code in an ME's control memory, we dropped several functions from the Linux-based switch. We developed this protocol for NP, and we call it SpliceNP protocol.[11] Table 1 compares processing a SYN packet for three implementations: traditional TCP, Linux-based switch, and NP-based switch using SpliceNP protocol.

Processing a SYN packet. To process a SYN packet, the switch dequeues the packet from where the device driver put it and checks it to confirm that it is a valid IP packet, as shown in steps 1 and 2 in Table 1. The IP validation includes checking the packet's version, length, and header checksum. Corrupted packets or packets other than IP or TCP are dropped. In NP, we don't process IP options (step 3), because they're rarely used. Next, the switch validates the TCP header, including TCP checksum and sequence number (step 4). Then, the control block lookup takes place based on a hash value calculated from the packet's source port and IP address (step 5). A new socket (together with the TCP control block) is created and its state is set to **`LISTEN`** (step 6). NP doesn't have socket operations because we don't need an interface between the TCP and the application.

In step 7, the switch creates the TCP and IP header template. While a packet is being sent, the switch copies this template as a whole to the TCP and IP header, rather than filling each field one by one. In NP, however, we don't create this template because the switch updates IP and TCP headers on the fly. In step 8, the switch resets the keep-alive timer. In NP, we do not implement any timers; however, we plan to work on this in the future. The next step is TCP-option processing (step 9). In NP, we process only the maximum segment size (MSS) option. We know that TCP options like MSS and SACK (selective acknowledg-

Table 2. Processing a SYN-ACK packet.

Step	Functionality	TCP	Linux based	NP based
1-5	Same as steps 1-5 in Table 1			
6	Reset idle time and keep-alive timer	Yes	Yes	No
7	Process TCP option	Yes	Yes	Only MSS option
8	Verify ACK number and flags	Yes	Yes	Yes
9	Connection-establishment timer	Yes	Yes	No
10	Initialize receive-sequence number	Yes	Yes	Yes
11	Set state to `ESTABLISHED`	Yes	Yes	Yes
12	Send ACK packet	Yes	Yes	Yes

ment) are negotiated between the two end points in a three-way handshake. Because the server cluster may have various options, the switch might reject all TCP options or maintain a minimum set of options for the Web servers. Currently, we implement MSS-option processing in the switch (1,460 bytes in Ethernet). Otherwise, if we reject the TCP options, the clients and servers will use a smaller MSS (534 bytes), which affects performance such as throughput. Finally, in step 10, the switch changes the state to `SYN_RECEIVED` state and sends out an ACK packet.

Processing a SYN-ACK packet. Table 2 illustrates processing a SYN-ACK packet, and Table 3 illustrates processing a data packet. Most of the steps are the same for all three cases in SYN-ACK processing, except that the NP processes only the MSS option. NP avoids the TCP header verification because only forwarding is performed in the `SPLICED` state. Packet type determines Step 8 in Table 3. Data packets follow 8a, and ACK packets follow 8b. When we enable splicing, we avoid the copying that TCP requires.

NP also avoids the window-control processing implemented in the traditional TCP. With TCP splicing, after the two connections are spliced together, only the client and the server handle the flow control. The switch does not need to maintain any window size information. However, representing the server, the switch must send the advertised window size in the TCP header when it accepts the connection from the client. Because the switch has no idea which server it will connect at that moment, it should choose a number that won't be too different from the one the real server uses. Otherwise, after the splicing, the client will see a smaller or a bigger window size than the one the switch sent before the splicing, which could trigger unnecessary data transmission or retransmission.[8] Fortunately, because the client receives mainly data packets from the server and sends only ACK packets, this window-size change doesn't affect the client's performance. This problem doesn't happen on the server side, because the switch uses the client-window size when connecting to the server. Still, the switch must choose a window size when it sends the SYN-ACK packet to the client. One solution is to probe the back-end servers to get a set of data to choose the minimum.

Design and implementation

In addressing the various design and implementation issues associated with our content-

Table 3. Processing a data or an ACK packet.

Step	Functionality	TCP	Linux based	NP based
1-5	Same as 1-3 and 5 in Table 1. Skip 4.			
6	Reset idle time and keep-alive timer	Yes	Yes	No
7	Process TCP option	Yes	Yes	Only MSS option
8a	Wake up receiving process	Yes	Direct forwarding	Direct forwarding
	Copy data to application	Yes	No	No
8b	Delete acknowledged data from send buffer	Yes	Direct forwarding	Direct forwarding
	Wake up waiting process	Yes	No	No
9	Flow-control processing	Yes	Yes	No

aware switch, we examined the Intel IXP2400 NP's architecture and analyzed how to efficiently distribute workload among resources in such a hardware environment.

Hardware

Figure 5 shows an ENP2611 board with an embedded IXP2400 NP connected to the host machine through a PCI bus. The IXP2400 contains a general-purpose XScale core and eight MEs, which have instruction sets tuned specifically for processing network packets. Each ME has a 16-Kbyte-instruction memory that the XScale processor core preloaded. Up to eight threads can run in parallel on each ME. The XScale runs an embedded Linux operating system. All processors share an SRAM and a DRAM.

When a packet arrives at the Ethernet interface, one of the media access controller devices attached to the media switching fabric receives it. Threads in MEs are programmed to move packets into a receive FIFO buffer, do some processing, and put outgoing packets in a transmit FIFO buffer, where they are transmitted to the line.

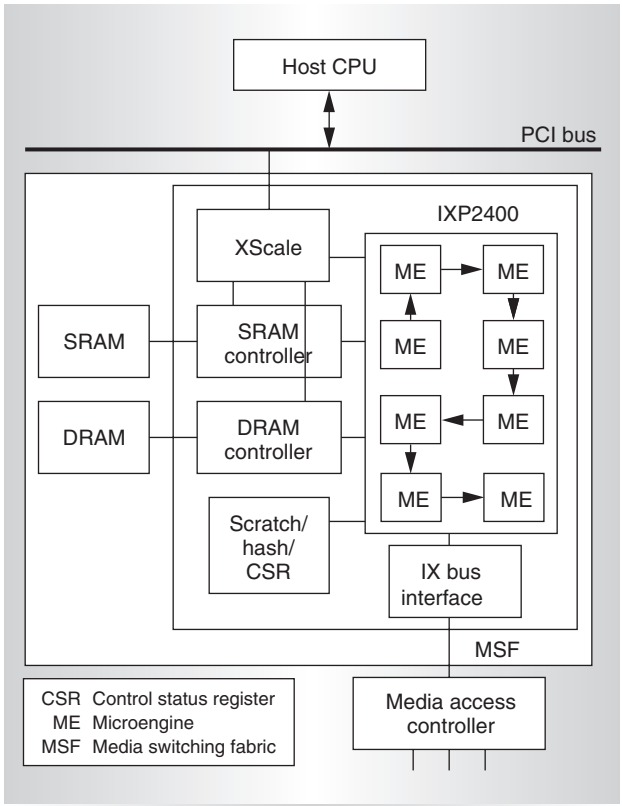

Figure 5. ENP2611 high-level architecture.

Resource allocation

Given the IXP2400's hardware environment, which consists of multiple processors and threads and various memory modules, allocating these resources for minimum packet processing time is a challenge. We must also carefully allocate data in the memory. The two off-chip memory modules—SRAM and DRAM—have not only different sizes but also different access latencies. When unloaded, SRAM's access latency is about 90 cycles, and DRAM's access latency is about 120 cycles.[12] Because SRAM is faster than DRAM, we use SRAM to maintain all the control-data structures. We use DRAM, which is relatively large, for buffering packets.

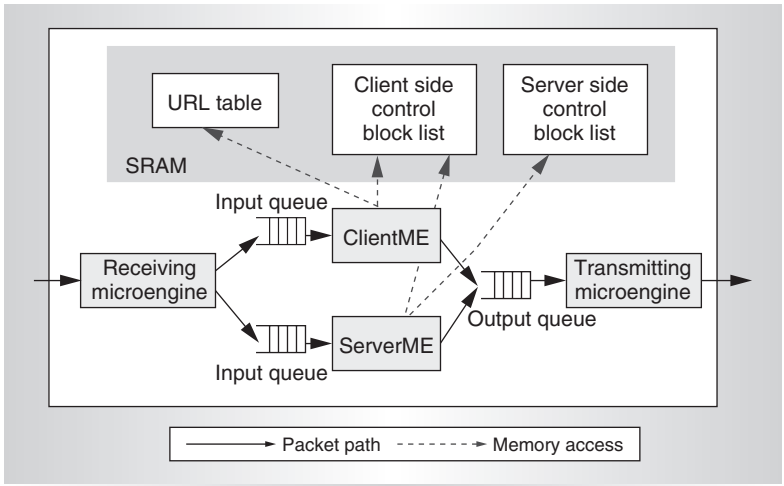

Figure 6. The microengine workload partition for our content-aware switch.

Figure 6 shows the resource allocation for our content-aware switch. We first differentiate client ports from server ports. Client ports connect with the external world (clients). Server ports connect to servers in the cluster and are responsible for receiving packets from servers. MEs fall into four groups: those that receive (RX ME), those that transmit (TX ME), those that process packets from the client ports (clientME), and those that process packets from the server ports (serverME). These MEs form a packet-processing pipeline. RX MEs receive packets from the input ports and put them in the input queue. ClientMEs or serverMEs process packets from these queues and move them to the next output queue. TX MEs transmit those packets out to

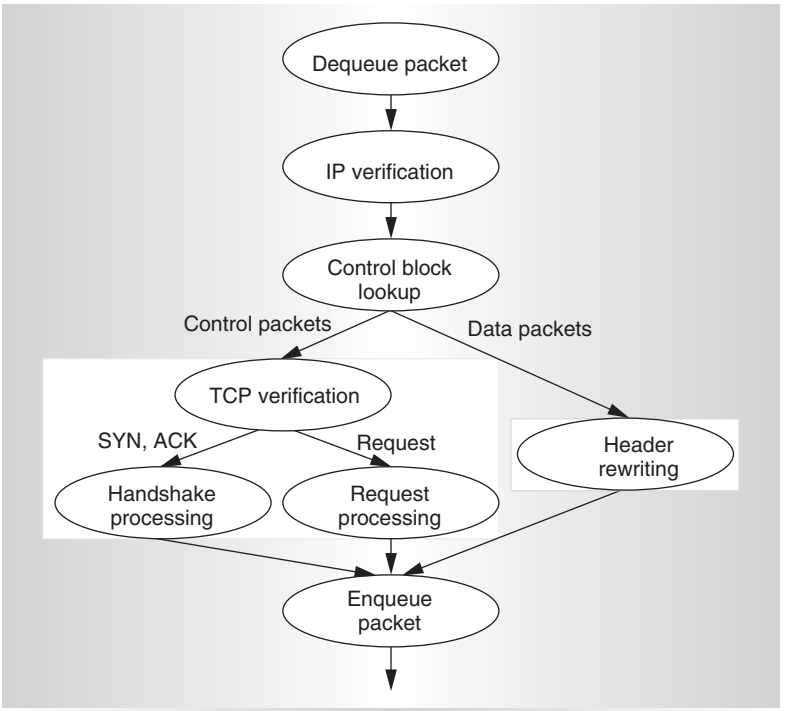

Figure 7. Data flow on the clientME.

the line.

The input and output queues convey packet information between MEs. These queues, which are implemented in SRAM, store packet descriptors containing the DRAM address, length of packets, input and output ports, and so on. TX MEs send these packets out based on the output port number.

Our switch uses three major data structures: a client-side control block list (C-list), a server-side control block list (S-list), and a URL table. The C-list records the state for the connection between the client and the switch, and the state for forwarding data packets after connections are spliced. The S-list records the state for the connection between the switch and the selected server. This switch uses the URL table to select a back-end server for an incoming HTTP request. This table contains a set of predefined mappings from URL suffixes to back-end servers. Future work will include implementing more advanced algorithms. The switch maintains all these data structures in SRAM. In addition, because multiple threads or MEs might access the control blocks simultaneously, updating these control blocks must be atomic. To accomplish this, we exploit the SRAM locks supported in the IXP2400.

Processing on microengines

When a packet arrives, the clientME or the serverME extracts its IP and TCP headers and performs a control block lookup in the control block list. The processing on this packet is based on the state in the control block.

ClientMEs. Figure 7 shows the clientME data flow, starting when the clientME dequeues a packet from the input queue, as we described under "Processing a SYN packet."

Control packet processing and data packet processing are shown in the two shaded boxes in Figure 7. For a control packet, the clientME first validates the packet's TCP checksum and sequence number. Then it checks whether this packet is a SYN, an ACK, or an ACK-request packet. These three types of packets are the only control packets that must be processed at the clientME. The handshake-processing part processes SYN or ACK for connection establishment. For a SYN packet (with *CSEQ* as its initial sequence number), a control block is inserted into the C-list for the new connection, and its processing is based on steps described in Table 1. The ACK packet finishes establishing the connection. The switch parses the request packet in the request-processing module and chooses a back-end server on the basis of the URL table. Next, the clientME sets up the second connection with the selected server by sending a SYN packet with the client's IP and port as its source IP address and port number. The initial sequence number of this SYN packet is set as *CSEQ*. Thus, in effect, the switch masquerades as the client to send this SYN packet, so only minimum changes are required in the subsequent forwarding part. For this second connection, the clientME inserts a control block in the S-list.

If the incoming packet is found to be a data packet—in few cases, it's a FIN packet used to close the connection—the switch processes the packet based on the steps in Table 3. The switch then directly forwards the packet with its updated IP and TCP header and changes the packet's destination IP address to that of the server IP. The acknowledge number is updated with the following formula: *new acknowledge number* = *old acknowledge number* − *DSEQ* + *SSEQ*, where *DSEQ* and *SSEQ* are initial sequence numbers in the SYN packet sent from the switch and the serv-

er, respectively. The checksum in both the IP and TCP header are recalculated using the incremental checksum calculation method.[13]

ServerMEs. Processing on the serverME has a data flow that differs slightly from that in Figure 7. One difference is that the serverME accesses the C-list using a hash value based on the destination IP address and port number. For the control packet, the serverME must handle only the SYN-ACK packet from the server because the clientME sends the SYN packet that initializes a connection with the chosen server. The SYN-ACK packet processing is based on the steps in Table 2. In response to this SYN-ACK packet, the serverME can send an ACK packet, and then the HTTP request. Because the data can piggyback on the ACK packet, we send the saved request along with the ACK. The state of the C-list control block changes to **`SPLICED`** thereafter. The switch deletes the corresponding entry in the S-list.

ServerME data-packet processing is also similar to that on the clientME. The difference lies in the updated fields. The source IP is set to the switch IP address, which is a VIP. The switch updates the sequence number with the following formula: *new sequence number* = *old sequence number* − *SSEQ* + *DSEQ*.

Other implementation issues

When the switch terminates the connection between the server and the client, it must delete the corresponding control block after 2MSL. To implement this time control, we maintain a time-out table in SRAM, with each entry containing a pointer to a control block and a time stamp that records the time when the control block should be deleted. Because the control block's deletion is not on the critical path for a connection, we run a program on XScale that checks the time-out table regularly and deletes the control block if it expires.

Performance evaluation

Our experiments compared our NP-based content-aware switch with a Linux-based switch, focusing on latency and throughput.

Experimental setup

We implemented a content-aware switch that uses a RadiSys (http://www.radisys.com) ENP2611 board containing an Intel IXP2400 processor. The XScale and microengines run at 600 MHz. This board has 8-Mbyte SRAM, 128-Mbyte DRAM, and three 1-Gbps Ethernet ports. We used one port as the client port and the other as a server port. The server port was connected with an Apache (http://www.apache.org) Web server running on an Intel 3.0-GHz Xeon processor. The client port was connected to a Layer-2 switch that connected two clients. Each client ran httperf[14] on a 2.5-GHz Intel Pentium 4 processor. All PCs ran Linux 2.4.20. To compare its performance with that of a Linux-based switch, we also built a Linux-based switch by inserting a loadable kernel module (from the Linux Virtual Server Project, http://www.linuxvirtualserver.org) into its operating system. This switch ran a Linux 2.4.20 kernel on a 2.5 GHz Pentium 4 system with two 1-Gbps Ethernet NICs.

We obtained the following results with one ME for RX ME and one ME for TX ME. In our experiments, we varied the number of clientMEs and serverMEs. However, using one clientME and one serverME yielded the same results as using two clientMEs and two serverMEs. This implies that a pool of processors might not be helpful, because all the MEs compete for the shared SRAM and DRAM.

Latency

First, we conducted experiments to obtain the latency of packet processing for an HTTP session. Figure 8 shows the latency spent on the switch when we vary the request file size. Compared with the Linux-based switch, the latency on the NP-based switch is reduced by 83.3 percent (0.6 µs to 0.1 µs) with as small a file size as 1 Kbyte. The larger the file size, the higher the reduction. At 1,024 Kbytes, the latency drops by 89.5 percent.

Using read-time-stamp instruction, we measured the processing latency for data and control packets separately. The timing starts when the clientME or serverME takes the packet from the input queue, and ends when it puts a processed packet in the output queue. For example, the latency on an ACK/request packet records from the time this packet is assembled in the DRAM to the instant the SYN packet to the server is put in the queue.

Table 4 shows the average processing laten-

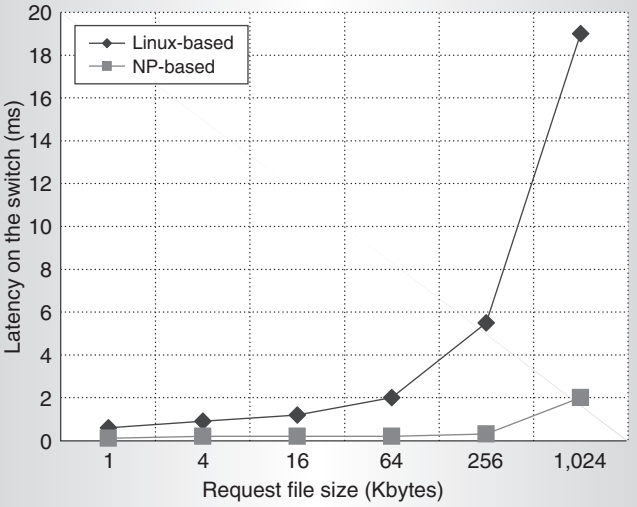

Figure 8. Latency in a Linux-based switch versus an NP-based switch for an HTTP session.

cy on control packets and data packets for both a Linux-based and IXP-based switch. In a Linux-based switch, control-packet processing takes much longer than data-packet processing. This is mainly because the control packets travel through the protocol stack in Linux, whereas the data packets do not. Among control packets, the ACK/request packet has the longest processing time. This request packet, in addition to traversing the protocol stack, is copied into the user space, where the proxy running at the user level parses it. The data packets in Linux-based splicing do not travel through the TCP layer; hence, they consume far less time than the control packets.

Both control-packet and data-packet processing in an IXP-based switch takes considerably less time than in the Linux switch, even though the Linux machine (2.5 GHz) is much faster than the microengines (600 MHz). The response packets from the server and the ACK packets from the client take the same time because our measurement does not include data assembly. We make this exclusion, keeping in mind the latency measurement in Linux, in which we also don't include data assembly. The reduction for control packets and data packets is about 83 percent and 52 percent, respectively. Although many optimized TPC/IP protocol implementations in Linux use polling instead of interrupt, our design still maintains the reduction advantages.

Throughput

To measure the throughput that these two switches achieve, we sent requests of a uniform size as fast as possible from the clients. Figure 9 shows the results. We can see that throughput increases by 5.7 times for small requests, such as 1 Kbyte, in which throughput increases from 8.2 Mbps to 46.4 Mbps. For much larger file sizes, such as 1,024 Kbytes, throughput improves by 2.2 times. Requests for small files show greater improvement because control packets take a larger portion of an HTTP session for small files, and latency reduction for control packets is larger than that of data packets. Therefore, the throughput improvement is more apparent for small requests. As we increase the request file size, data-packet processing becomes dominant, so we see relatively smaller improvement on NP. Here, we use only one clientME and one serverME to process the packets, as Figure 6 shows. We can further improve throughput by using more microengines in the IXP2400.

SRAM versus DRAM

We obtained our results by maintaining the control blocks in SRAM. We also use SRAM to maintain hash tables that help fast table

Table 4. Control- and data-packet processing latency using a TCP protocol stack.

		IXP2400		Linux	
Packet type		**Microengine**	**Latency (μs)**	**Latency (μs)**	**Latency reduction**
Control packet	SYN	client ME	7.2	48.0	85%
	ACK/request	clientME	8.8	52.0	83%
	SYN-ACK	serverME	8.5	42.0	80%
Data packet	Data	serverME	6.5	13.6	52%
	ACK	clientME	6.5	13.6	52%

lookup for control blocks, and we use locks provided in SRAM. Therefore, for each packet to access its control block, there are at least three contiguous SRAM accesses: one for the control block, one for the hash table, and one for the lock. When the switch processes thousands of connections simultaneously, these SRAM accesses can become a bottleneck. Also, maintaining thousands of control blocks in SRAM is impossible because of its size limitation. So, we measured performance when maintaining the control blocks in DRAM, and maintaining the hash tables and implementing the locks in SRAM. In this way, we can also distribute the memory accesses more evenly into the SRAM and DRAM modules and pipeline their accesses. Furthermore, because DRAM is much larger than SRAM, we can increase the number of control blocks so that the switch can support far more connections simultaneously. We've found that the latency obtained on the client side is the same whether we implement the control blocks in SRAM or DRAM. Although latency is greater in DRAM than in SRAM (120 cycles versus 90 cycles), the difference in HTTP latency is negligible given the extensive waiting time for SRAM when we implement all the tables in SRAM.

We then measured the throughput when we implemented the control blocks in DRAM and SRAM as a function of the request rate. We also increased the number of servers to two so that the switch satisfied more requests. Figure 10 shows the results when we fix the request file size at 64 Kbytes. The *x*-axis is the request rate in the unit of requests per second (or connections per second as we send one request in one connection). As we increase the request rate, the inter-arrival time between packets drops accordingly. When we increase the request rate to 1,300, the throughput saturates at 665.6 Mbps when we implement the control blocks in SRAM. However, with control blocks in DRAM, the throughput continues to increase until 720.9 Mbps. This data verifies that we can increase throughput by distributing memory requests to as many modules as possible.

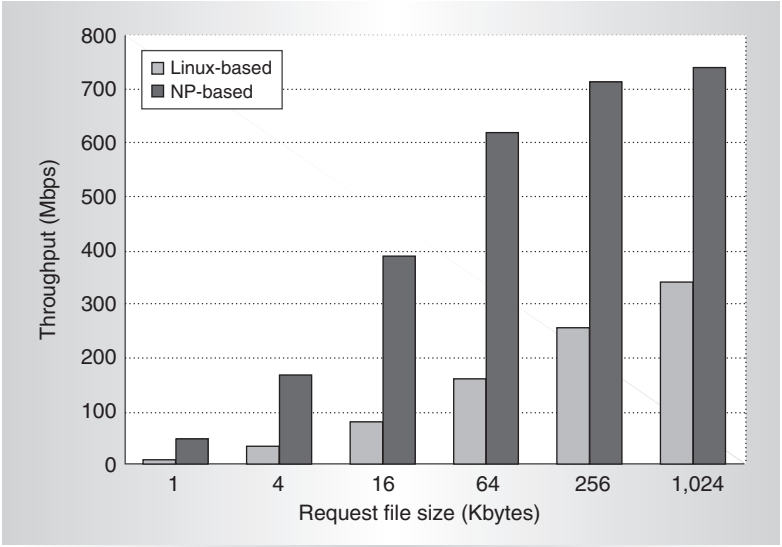

Figure 9. Throughput comparison for HTTP sessions.

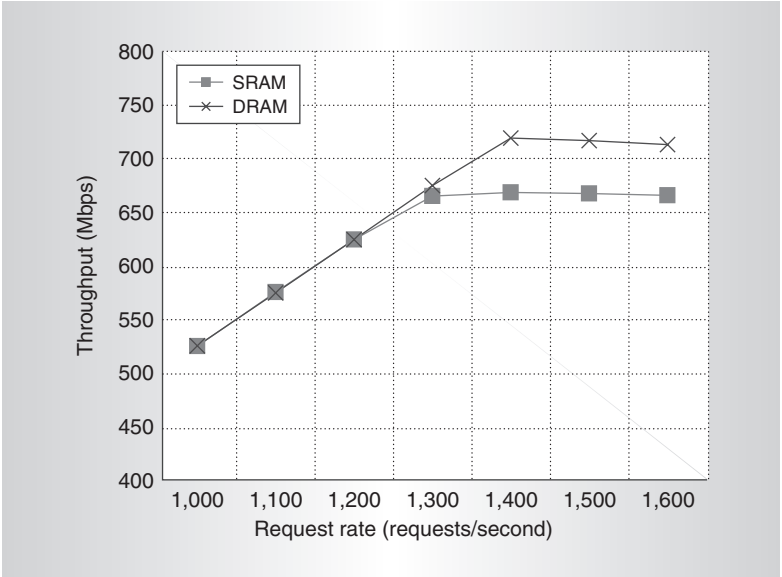

Figure 10. Throughput comparison for control blocks implemented in SRAM and DRAM.

This article presents the latest results in our effort to enhance our NP-based content-aware switch.[15] Our future work includes processing all the TCP options in the network processor because they can affect performance. We plan to further break down the clientME's and serverME's functionality and assign them to more MEs, so that we can create parallel and pipelined processing to improve throughput. In addition, we plan to incorporate other functionalities such as quality of service by identifying the packet flows and providing differentiated service to an individual flow.

References

1. Cisco Systems, Cisco Content Services Switch, http://www.cisco.com/en/US/products/hw/contnetw/ps789/index.html.
2. Foundry Systems, Foundry ServerIron XL/G Web Switch, http://www.b2net.co.uk/foundry/foundry_serveriron_xlg_web_switch.htm.
3. Nortel Networks, Alteon Web Switches, http://www.nortelnetworks.com/products/01/alteon/webswitch/index.html.
4. A. Cohen, S. Rangarajan, and H. Slye, "On the Performance of TCP Splicing for URL-Aware Redirection," *Proc. 2nd Usenix Symp. Internet Technologies and Systems* (USITS 99), Usenix, 1999; http://www.usenix.org/events/usits99/cohen.html.
5. IBM, IBM WebSphere Edge Server, http://www.ibm.com/software/webservers/edgeserver/.
6. C. Yang and M. Luo, "Efficient Support for Content-Based Routing in Web Server Clusters," *Proc. 2nd Usenix Symp. Internet Technologies and Systems* (USITS 99), Usenix, 1999, http://www.usenix.org/events/usits99/yang.html.
7. D. Maltz and P. Bhagwat, "TCP Splicing for Application Layer Proxy Performance," IBM Research Report RC 21139, 1998.
8. O. Spatscheck et al., "Optimizing TCP Forwarder Performance," *IEEE/ACM Trans. Networking*, vol. 8, no. 2, 2000, pp. 146-157.
9. M. Rosu and D. Rosu, "Kernel Support for Faster Web Proxies," *Proc. 2003 Usenix Annual Technical Conf.*, Usenix, 2003, pp. 225-238.
10. G.R. Wright and W.R. Stevens, *TCP/IP Illustrated, Volume 2: The Implementation*, Addison-Wesley, 1995.
11. L. Zhao et al., "SpliceNP, A TCP Splicer Using Network Processors," *Proc. Symp. Architectures for Networking and Communications Systems*, ACM Press, 2005, pp. 135-143.
12. E.J. Johnson and A.R. Kunze, *IXP2400/2800 Programming The Complete Microengine Coding Guide for the Network Processor Family*, Intel Press, 2003.
13. A. Rijsinghani, ed., "RFC1624: Computation of the Internet Checksum via Incremental Update," Network Working Group, May 1994, http://rfc.sunsite.dk/rfc/rfc1624.html.
14. D. Mosberger and T. Jin, "httperf: A Tool for Measuring Web Server Performance," *ACM Sigmetrics Performance Evaluation Rev.*, 1998, pp. 31-37.
15. Li Zhao et al., "Design and Implementation of a Content-aware Switch Using a Network Processor," *Proc. 13th IEEE Symp. High Performance Interconnects* (HOTIC 05), IEEE Press, 2005, pp. 79-85.

Li Zhao is a senior engineer in the System Technology Laboratory at Intel Corp. She has a PhD in computer science from the University of California, Riverside. Her research interests include computer architecture, network computing, and performance evaluation. She is a member of the IEEE.

Yan Luo is an assistant professor in the Department of Electrical and Computer Engineering at University of Massachusetts, Lowell. His research interests include network processor architecture, Internet router and Web server, network security, and performance evaluation. Luo has a PhD in computer science from the University of California, Riverside. He is a member of the IEEE and ACM.

Laxmi N. Bhuyan is a professor of computer science and engineering at the University of California, Riverside. His research interests include network processor architecture, Internet routers, and parallel and distributed processing. Bhuyan has a PhD from Wayne State University. He is a Fellow of the IEEE, the ACM, and the American Association for the Advancement of Science.

Ravi Iyer is a principal engineer in the Systems Technology Laboratory at Intel an associate editor for *IEEE Transactions on Parallel and Distributed Systems*. His research interests include computer architecture, server design, network protocols and acceleration, workload characterization, and performance evaluation; his current emphasis is on large-scale CMP architectures and technologies. He has a PhD in computer science from Texas A&M University.

Direct questions and comments about this article to Li Zhao, Intel Corp., 2111 NE 25th Ave., M/S JF2-58, Hillsboro, OR 97124; li.zhao@intel.com.