SpliceNP: A TCP Splicer using A Network Processor

Li Zhao, Yan Luo, Laxmi Bhuyan
Computer Science & Engineering Department
University of California
Riverside, CA
{zhao, yluo, bhuyan}@cs.ucr.edu

Ravi Iyer
Systems Technology Lab
Intel Corporation
Hillsboro, OR
ravishankar.iyer@intel.com

ABSTRACT

TCP Splicing can be used in content-aware switches to tremendously reduce overall request latency. In order to reduce the processing latency further, we propose to offload the protocol processing onto network processors (NPs). An NP consists of a multithreaded multiprocessor architecture that can provide high throughput for packet processing or forwarding. However, offloading any protocol software to an NP needs to be carefully designed due to its low-level programming and limited control memory size.

In this paper, we first analyze the operation of TCP Splicing in detail and evaluate its performance through measurements on a Linux-based switch. Then various possibilities of workload allocation among different computation resources in an NP are presented, and the design tradeoffs are discussed. A content aware switch is implemented using IXP 2400 NP and evaluated for performance comparison. The measurement results demonstrate that our NP-based switch can reduce the http processing latency by an average of 83.3% for a 1K byte web page. The amount of reduction increases with larger file sizes. It is also shown that the packet throughput can be improved by up to 5.7x across a range of files by taking advantage of multithreading and multiprocessing, available in the NP.

Categories and Subject Descriptors

C.2.1 [**Network Architecture and Design**]: Network communications

General Terms

Measurement, Design, Performance

Keywords

Network processors, TCP Splicing

1. INTRODUCTION

Server clusters have been extensively used to build a cost effective, scalable and reliable server system. One of the most important components for such a system is switch, which forwards packets between outside world and the backend servers. A content-aware switch [3] [6] [12] routes packets based on the request content or application layer information. Upon receiving an HTTP request, the switch examines the request content, and chooses a back-end server based on pre-defined rules.

A straightforward way to build a content-aware switch is to use an HTTP proxy, which runs as an application. This proxy first accepts the initial connection from the client, parses the received request, and chooses a back-end server. Then the proxy establishes a second TCP connection with the selected server, and directs the request to it. When the response packet from the selected server is received on the second connection, the proxy forwards this packet to the client through the first connection. This approach is easy to implement as no changes of the operating system on the switch are required. However, the overhead to copy data between these two connections is high: the data from the server needs to go up through the protocol stack, and be copied from the kernel space to the user space; then, this data is copied to the kernel and goes down through the protocol stack again.

TCP splicing [10] [18] solves this data copy problem by splicing the two connections once these two connections are set up. The switch then forwards subsequent data packets on the spliced connection by modifying particular fields (e.g. sequence numbers) in their TCP and IP headers. Since this data forwarding is performed at the IP level, the overhead of copying data between the user space and the kernel space is avoided.

TCP splicing has been used to build content-aware switches based on general purpose processors[10] [4] [20] and ASICs [2]. However, switches based on general purpose processors cannot provide satisfactory performance due to interrupt, moving packets through PCI bus, and large protocol stack overhead in the operating system. Also, the instruction set architecture of the general purpose processors is not tuned for packet processing. ASIC based switches, on the other hand, have no flexibility although they can achieve very high processing capacity. Network processors (NPs) are new processors optimized for packet processing. They usually consist of a general-purpose control processor and data processors that support multiprocessing and multithreading. The NPs operate at the link layer of the protocol like ASICs, thus avoiding the large OS overhead of the general-purpose processors. At the same time, they are programmable. To the best of our knowledge, no study on design of content-aware switches has been done using NPs. The aim of this paper is to design and implement such a switch and demonstrate its high performance and programmability.

It is known that a number of industry projects is undergoing at present on offloading TCP to the NPs, like Intel IXP. All these projects try to offload TCP to the control processor (like XScale

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
ANCS'05, October 26–28, 2005, Princeton, New Jersey, USA.
Copyright 2005 ACM 1-59593-082-5/05/0010 ...$5.00.

in IXP) because it is programmed in C language and there is no memory restriction. Even though such an implementation avoids the latency through the PCI bus, our measurements in this paper will show that the TCP latency is not significantly reduced. Our aim is to design a TCP or TCP splicing version that can be implemented in the data processors (called micro engines in the IXP) so that (1) the latency is reduced by avoiding the embedded Linux kernel in the control processor and (2) packet throughput is increased by using multiple threads and multiple data processors in an NP. However, implementing TCP splicing technique on data processors in NP is not simple. First, the NPs are programmed at a low level language (MicroC or microcode) without the availability of a compiler that can directly translate the TCP splicing code from C to this language. Second, the instruction memory available in an NP is limited, so an incredible amount of effort is needed to reduce and optimize the existing code. We design a lite version of TCP splicing in this paper that we call *SpliceNP*.

In this paper, we first measure the performance of TCP splicing on a Linux based switch and then analyze the function-level processing latency of key TCP splicing operations. The experimental results help us identify the critical functions of the TCP splicing operations that must be offloaded to an NP. Then we carefully allocate these functions to different microengines and threads in the NP for optimal performance. In the process we have to ensure that the computational power of the NP is fully utilized for maximum throughput. We implement the TCP splicing on an ENP2611 board that contains an Intel's IXP2400. Our performance evaluation results show that an NP-based TCP splicing technique can significantly improve processing latency as well as the throughput. While our previous work [21] has shown similar results, the focus on this work is on the protocol design and implementation.

The contribution of this paper is the following:

- We analyze the TCP splicing technique in detail, discuss how this technique reduces the overall latency and present measurement results through experiments based on Linux switches.
- We present how an NP can improve the TCP splicing performance, and derive protocols to implement it on microengines.
- We implement TCP splicing on the ENP2611 that contains an Intel IXP2400 processor, and present a detailed performance evaluation. We plan to make the source code of our splice-NP implementation available soon for academic research.

The rest of the paper is organized as follows. Section 2 analyzes the TCP splicing technique and quantifies the performance improvement by using this technique. The motivation to build a content-aware switch based on network processors and a detailed protocol implementation are presented in Section 3. Section 4 describes details of our design and implementation on an IXP2400 NP. The experimental results are presented in Section 5. Section 6 describes the related work. Section 7 summarizes and concludes this paper.

2. ANATOMY OF TCP SPLICING

In this section, we first describe the TCP splicing technique in detail, and present its state transition. We then obtain experimental results on a Linux-based switch to quantify the achieved performance gain. Based on the analysis on these results, we decide how to offload the whole processing onto a network processor.

2.1 What is TCP splicing?

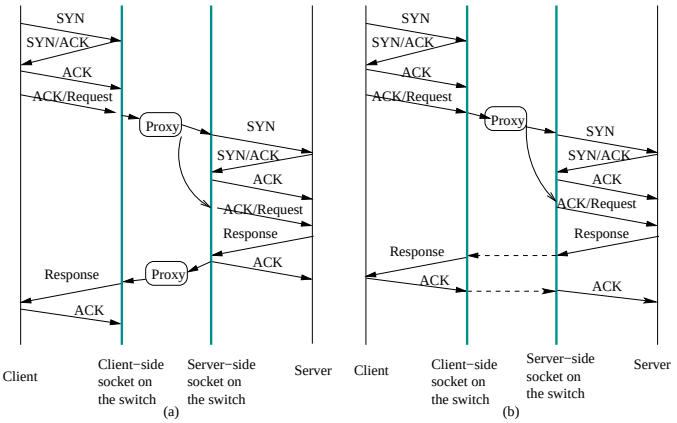

Figure 1: Operations on a content-aware switch. (a) w/o splicing (b) w/ splicing

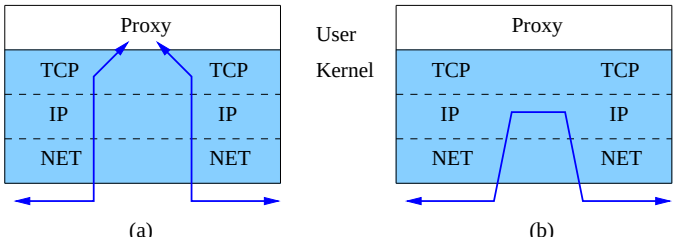

Figure 2: Comparison of protocol stack processing. (a) w/o splicing (b) w/ splicing

Figure 1 shows the operations with and without the TCP splicing technique in a Linux-based switch. In Figure 1(a), an HTTP proxy is running in the user space with a client-side socket waiting for clients' requests. A client initiates a three-way handshaking with the switch, and then sends the HTTP request. The proxy parses this request and establishes another connection with the appropriate server, also through the three-way handshaking. Then the proxy forwards the HTTP request to the server through the second connection. Upon receiving the response data from the server, the proxy forwards it to the client through the first connection. As the proxy runs in the user space, this "forwarding" involves copying data from the kernel space to user space and back to kernel space again, which is illustrated in Figure 2(a). This copying overhead can be avoided by using TCP splicing, shown in Figure 1(b). Once the proxy sends the HTTP request to the server, the response packet is forwarded directly to the client without any control by the proxy. The ACK packet from the client is also forwarded to the server. As this splicing procedure is transparent to both the server and the client, certain fields in the IP and TCP headers of the packets need to be modified, yet all the processing is done at the IP level, as shown in Figure 2(b).

2.2 TCP Splicing State Transition

TCP splicing is based on the standard TCP/IP protocol [19], which maintains the state information for each connection. Therefore, we also need to maintain the state information for TCP splicing. In order to implement TCP splicing, we need to clearly understand various states of this process, shown by the state transition diagram in Figure 3. The state transition from CLOSED

to SYN_RCVD and ESTABLISHED states is the three-way handshaking standardized in TCP/IP protocol. There are two handshakings in the figure. The first one is initiated by the client, which leads to ESTABLISHED state for the first connection. The second three-way handshaking is triggered by receiving an HTTP request packet. After the second connection also enters ESTABLISHED state, these two connections are spliced together and the state migrates to SPLICED. The termination of the spliced connection starts when a FIN packet is received from one side (server or client). This leads to the state transition to FIN_RELAYED1. When the ACK to this FIN packet is received, the state becomes ACK_RELAYED1. The arrival of a FIN packet from another side transits 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 [19]), the state is changed to CLOSED. Notice that data forwarding occurs from SPLICED state until FIN_RELAYED2.

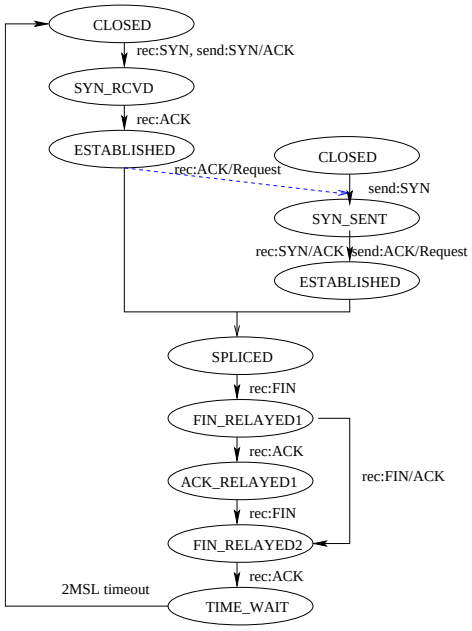

Figure 3: State transition in TCP splicing

Based on this state diagram, we classify packets into two types: control packets and data packets. Control packets are those sent before the two connections are spliced. These packets, such as SYN packets, are used to set up connections. The HTTP request packet is also treated as a control packet as it causes the second connection to be setup. Data packets are those sent after the two connections are spliced. They are response packets from the server, ACK packets from the client, and FIN packets from both sides.

3. USING NP TO IMPROVE TCP SPLICING

In this section, we discuss the motivation to use a network processor (NP) to improve the TCP splicing performance, and present the details of our SpliceNP protocol showing the savings in time.

3.1 Why use A Network Processor?

The HTTP request is parsed by the proxy, which runs on the application level. Hence this request packet is required to go through the protocol stack and be copied from the kernel space to the user space. The latency can be improved by removing the proxy and moving all the processing including parsing the HTTP request into the kernel space [20] [16], as shown in Figure 4(a). However, data has to be moved from host DRAM to NIC and from NIC to host DRAM over the PCI bus. This imposes heavy bandwidth pressure on PCI bus when the number of connections is large. It also introduces interrupt overhead to the host CPU.

To further improve processing of control packets as well as data packets, we propose to move all the processing down to the NIC level. Figure 4(b) and (c) utilize Network Processor (NP) -based network interfaces. The NP usually has a control processor and multiple data processors. The data processors are tuned specifically for processing network packets in the fast path, whereas control processor is used to maintain the control information, and process exception packets. For example, Intel IXP2400 network processor contains one control processor (XScale) and 8 data processors (called microengines). The control processor runs an embedded Linux and shares DRAM with data processors. The data processors receive and transmit packets through NICs.

In Figure 4(b), TCP protocol stack in the control processor (XScale) can be used to create connections to clients and servers, and splice these two connections in the embedded Linux kernel. Then the data packets sent after splicing can be processed on the MEs. A number of industrial projects use this implementation for offloading TCP to an NP. However, our experimental results showed that processing control packets in XScale leads to longer latency due to the following reasons. In order to retrieve control packets, the XScale has to poll an input queue, which is filled by the microengines. After processing these packets, the XScale must put them into an output queue, from which a microengine sends them out. The packet en-queueing, de-queueing and polling time taken together increase the processing latency on control packets. Notice that these control packets fall in the critical path for TCP splicing. This delay is detrimental to the overall performance because longer delay may cause timeout on clients and may lead to packet retransmissions. In effect, this technique replaces Linux by embedded Linux and a powerful Pentium CPU by a weak XScale CPU. Nonetheless, we implemented this and observed that such an implementation increases the latency instead of reducing it.

Given a large number of microengines (MEs) and threads in NPs, Figure 4(c) is a natural evolution over (b). After receiving packets from NICs, the packet processors handle the connection creation, splicing and data forwarding, without the need to communicate with the control processor or the host CPU. The large number of hardware threads in packet processors are capable of fast packet processing and eliminating data copying. We use this architecture to design and implement a content-aware switch called *SpliceNP*. However, implementing a complex splicing software in ME is difficult because unlike XScale they are programmed in MicroC (instead of C) and are limited in control memory.

Compared to a Linux splicer in Figure 4(a), the SpliceNP can reduce the processing latency in four ways:

- Interrupt vs. polling. When NIC in the Linux machine receives packets, it raises an interrupt to the CPU. Although current NICs have the ability to accumulate multiple packets and then notify the processor using a single interrupt, the overhead of interrupt is still high. NPs use polling instead of interrupt to reduce this overhead.

- NIC-to-memory copy vs. no copy. In the Linux-based switch, the NIC has to copy the received packets to the main memory, which requires a DMA transfer through the PCI bus. Similarly, when the packets are sent out, they are transferred from

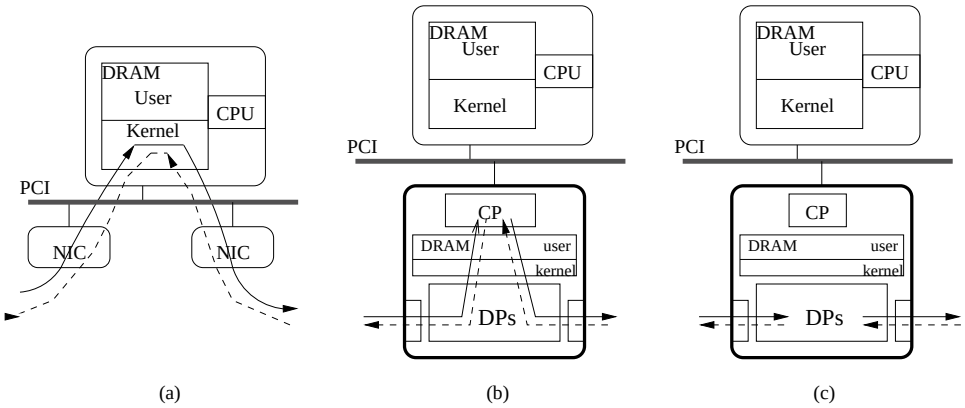

Figure 4: Three architecture candidates for TCP splicing

the memory to the NIC buffer by DMA again. In NP-based switch, however, packets are processed inside the NIC without the need for copy.

- Linux processing vs. IXP processing. Even if the whole TCP splicing is implemented in Linux kernel, the OS overhead like context switch would happen. In NP, with optimized instruction set architecture for packet processing, the hand coded splicer enables us to process packets in a more efficient way.

- Multiple MEs and threads in an NP can process many packets in parallel, thus increasing the throughput.

The first two factors involve a significant portion of packet processing. In order to measure the time taken by various part of the kernel, we use a Linux PC with a Pentium CPU running at 400MHz, and instrument the kernel code with *read-timestamp-counter* instructions for functions that we are interested in. Figure 5 shows a time line analysis of receiving a 41-byte message using TCP/IP. After the packet is copied through DMA to the host memory (t0), the interrupt handler (t1) saves all the CPU registers on the stack and invokes the NIC interrupt service routine tulip_interrupt() (t2), which raises a soft interrupt. At the end of the interrupt handler (intrStop), the soft interrupt softirq is executed and calls net_rx_action() (t3), which finally starts the network layer function ip_rcv() to start the TCP/IP stack processing. We can see from the figure that the DMA and interrupt handling (t0-t3) totally take about 11.7 us, whereas this part is almost negligible in the NP implementation when Figure 4(b) or (c) is employed.

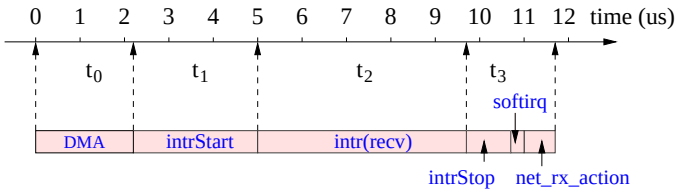

Figure 5: DMA and interrupt handling to receive a packet

3.2 The SpliceNP Protocol

We simplify the NP implementation since SpliceNP is specific for TCP splicing, whereas the TCP code in Linux must handle every situation. Also, in order to fit the code in the control memory of a ME, we drop several functions from the Linux splicer. Table 1 compares processing of a SYN packet for three implementations: traditional TCP, Linux splicer and the SpliceNP protocol.

To process a SYN packet, the packet is de-queued from where the device driver put it, and is checked to make sure that it is a valid IP packet, as shown in step 1 and 2. The IP validation includes checking its version, length and header checksum. Corrupted packets or packets other than IP or TCP are dropped. In SpliceNP, we do not process IP options (step 3) since they are rarely used. Next the TCP header is validated, which include TCP checksum and sequence number (step 4). Then the control block lookup is performed based on a hash value calculated from the source port and IP address of this packet (step 5). A new socket together with the TCP control block is created and its state is set to LISTEN (step 6). In SpliceNP, there are no socket operations since we do not need any interface between the TCP and the application. In step 7, the TCP and IP header template is created. When a packet is being sent out, this template is copied as a whole to the TCP and IP header instead of filling each field one by one. In SpliceNP, however, we do not create this template because IP and TCP headers are updated on the fly. In step 8, the keep-alive timer is set. In SpliceNP, we do not implement any timers. This is left as our future work. Next the TCP option is processed. In SpliceNP, we only process Maximum Segment Size (MSS) option. It is known that TCP options like MSS and SACK are negotiated between the two end points in a three-way handshake. Since the cluster of servers may have various options, the switch may either reject all TCP options, or maintain a minimum set of options for the web servers. Currently we implement MSS option processing in the switch (1460 bytes in Ethernet). Otherwise if we reject the TCP options, the clients and servers will use a smaller MSS (534 bytes), which affects the performance (such as throughput). Finally the state is changed to SYN_RECEIVED state and an ACK packet is sent out.

Tables 2 and 3 illustrate the processing of a SYN/ACK packet and a data packet respectively. Most of the steps are same for all the three cases in SYN/ACk processing, except that the SpliceNP only process MSS option. The TCP header verification in step 4 in Table 2 is avoided in SpliceNP because only forwarding is performed in the spliced state. Steps 8(a) and 8(b) indicate a data packet and a pure ACK packet processing respectively. We can see that the copy in TCP is avoided when splicing is enabled. This shows how the splicing technique improves the switch performance as stated in the previous section. Finally, the window control processing implemented in the traditional TCP is avoided in SpliceNP. With the TCP splicing technique, after the two connections are spliced together,

Table 1: Processing an SYN packet

Step	Functionality	TCP	Linux Splicer	SpliceNP
1	De-queue packet	Y	Y	Y
2	IP header verifi cation	Y	Y	Y
3	IP option processing	Y	Y	N
4	TCP header verifi cation	Y	Y	Y
5	Control block lookup	Y	Y	Y
6	Create new socket and set state to LISTEN	Y	Y	No socket, only control block
7	Initialize TCP and IP header template	Y	Y	N
8	Reset idle time and keep-alive timer	Y	Y	N
9	Process TCP option	Y	Y	Only MSS option
10	Send ACK packet, change state to SYN_RECEIVED	Y	Y	Y

Table 2: Processing an SYN/ACK packet

Step	Functionality	TCP	Linux Splicer	SpliceNP
1-5	Same as above table			
6	Reset idle time and keep-alive timer	Y	Y	N
7	Process TCP option	Y	Y	Only MSS option
8	Verify ACK number and flags	Y	Y	Y
9	Connection-establishment timer	Y	Y	N
10	Initialize receive sequence number	Y	Y	Y
11	Set state to ESTABLISHED	Y	Y	Y
12	Send ACK packet	Y	Y	Y

the flow control is handled by the client and the server only, and the switch does not need to maintain any window size information. However, representing the server, the switch has to send the advertised window size in the TCP header when it accepts the connection from the client. Since the switch has no idea of which server it will connect at that moment, it should choose a number that will not be too different from the one that the real server uses. Otherwise, after the splicing, the client will see a smaller or a bigger window size than the one sent by the switch before the splicing, which will possibly trigger unnecessary data transmission or retransmission [18]. Fortunately, since the client mainly receives data packets from the server, and only sends ACK packets, this window size change does not affect the client's performance. This problem does not happen in the server side because the switch uses the client window size when connecting to the server. Still, a window size needs to be chosen when the switch sends the SYN/ACK packet to the client. One solution is to probe the backend servers to get a set of data and choose the minimum.

4. DESIGN AND IMPLEMENTATION OF SPLICENP

We implemented the SpliceNP protocol using an Intel IXP2400 network processor and obtained various measurements in a web server environment. In this section, we first describe the architecture of the Intel IXP2400 network processor. Then, we study how to efficiently distribute the workload among various resources in such a hardware environment.

4.1 Hardware

We use an ENP2611 board with an embedded IXP2400 network processor, which is connected to the host machine through a PCI bus. The IXP2400 network processor contains nine programmable processors: a general-purpose XScale processor core and eight microengines with the instruction sets tuned specifically for processing network packets. Each microengine has a 16KB instruction memory preloaded by the XScale processor core. Up to eight threads can run in parallel on each microengine. As a result, the eight microengines can simultaneously execute a total of up to sixty-four threads. An SRAM controller and a DRAM controller control access to the SRAM and DRAM respectively.

4.2 Resource Allocation

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. Microengines are divided into four groups: receiving microengines (RX_ME), transmitting microengines (TX_ME), microengines that process packets from the client ports (ClientME) and from the server ports (ServerME). These microengines form a pipeline for processing packets. RX_MEs receive packets from the input ports and put them into the input queue. ClientME or ServerME process packets from these queues and put them into the next output queue. Finally TX_MEs are responsible to transmit those packets out onto the line.

The input and output queues are used to convey packet information between microengines. These queues are implemented in SRAM. They store packet descriptors, which contain the DRAM address, length of packets, input and output ports, etc. TX_MEs send these packets out based on the output port number.

Three major data structures are used in our switch: 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. The URL table is used to select a back-end server for an incoming HTTP request. This table contains a set of pre-defined mappings from URL suffixes to back-end servers.

4.3 Processing on MEs

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

Table 3: Processing a Data or an ACK packet

Step	Functionality	TCP	Linux Splicer	SpliceNP
1-5	Same as above table			
6	Reset idle time and keep-alive timer	Y	Y	N
7	Process TCP option	Y	Y	Only MSS option
8(a)	Wake up receiving process	Y	Direct forwarding	Direct forwarding
	Copy data to application	Y	N	N
8(b)	Delete acknowledged data from send buffer	Y	Direct forwarding	Direct forwarding
	Wake up waiting process	Y	N	N
9	Flow control processing	Y	Y	N

control block. The detailed operations on clientMEs/serverMEs are described below.

4.3.1 Processing on ClientMEs

A clientME processes the SYN packet and ACK packet for three-way handshaking, HTTP request packet and the rest of data packets The SYN packet is processed based on steps described in Table 1 with the control block implemented in the c-list. The request packet is parsed to choose a back-end server based on the URL table. Then the clientME set 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, which is the initial sequence number of the SYN packet sent from the client. The effect of this is that the switch masquerades as the client to send this SYN packet, so that only minimum changes are required in the subsequent forwarding part.

A data packet is processed based on steps in Table 3. The packet is directly forwarded with its IP and TCP header updated. Its destination IP address is changed to the server IP. The acknowledge number is updated with the following formula: $new\_ack\_number = old\_ack\_number - DSEQ + SSEQ$, where DSEQ and SSEQ are initial sequence numbers in the SYN packet sent from the switch and the server respectively. The checksum in both the IP and TCP header are recalculated with the incremental checksum calculation method [15].

4.3.2 Processing on ServerMEs

A serverME processes and SYN/ACK packet and data packets. The SYN/ACK packet processing is based on steps in Table 2, with the control block implemented in s-list. In addition, since the data can be piggybacked with the ACK packet, we do not send a pure ACK packet. Instead, we send the saved request with the ACK. The state of the control block in the c-list is changed to SPLICED thereafter. The corresponding entry in the s-list is deleted. The data packet processing is also similar to the clientME. The difference is on fields that are updated. The source IP is set to the switch IP address VIP. The sequence number is updated with the following formula: $new\_sequence\_number = old\_sequence\_number - SSEQ + DSEQ$.

4.3.3 Implementation of the Control Block List

The c-list is accessed by both the clientME and serverME for each incoming packet. In our implementation, we maintain the c-list as a hash table in the SRAM. In case of collision, the control blocks are implemented as a link list. Since the control blocks might be accessed by multiple threads/microengines simultaneously, updating these control blocks must be performed atomically. We exploit the SRAM locks supported in IXP2400 for this purpose. A free list of control blocks is pre-allocated in the SRAM. For new connections, control blocks are taken from this free list and inserted into the c-list. They are returned back in the free list when connections tear down. The s-list has a similar implementation.

When the connection between the server and the client is terminated, the state of this control block becomes TIME_WAIT. This control block is deleted after 2MSL (120 seconds). To implement this time control, we maintain a timeout table in SRAM. This timeout table is a circular array. Each entry contains a pointer to a control block, and a timestamp that records the time when its control block should be deleted. As the deletion of the control block is not on the critical path for a connection, we put a timeout-table checking program on the XScale. Its main functionality is to check the timeout table regularly, and compare the timestamp with the current time. If this entry expires, its control block will be deleted from the c-list and put pack to the free list.

5. PERFORMANCE EVALUATION

We conduct two experiments. The first one is to see how much improvement can be achieved by using TCP splicing, thus we compare the performance of a proxy-based Linux switch to that of a splicing-based Linux switch (we call it *Linux splicer*). The second experiment compares the NP-based switch (SpliceNP) with the Linux splicer in terms of latency and throughput.

5.1 Experimental Setup

For the switch without splicing, we run a user-level proxy with its operations indicated in Figure 1a. For the Linux splicer, we insert a *loadable kernel module* [8] into the operating system. As presented in Figure 1b, the proxy is still running at the user level. After the proxy sets up two connections and forwards the HTTP request, it makes a system call to inform the TCP splicing module to splice these two connections. This splicing operation records the states for the two connections and closes the client-side and server-side sockets. *Netfilter* [11] is used to direct the subsequent packets to the forwarding function so that these packets are forwarded directly in the kernel. Both Linux-based switches run a Linux 2.4.20 kernel on a 2.5GHz Pentium 4 system with two 1Gbps Ethernet NICs.

The SpliceNP is built up on an ENP2611 board that contains an Intel IXP2400 processor. Both XScale and microengines run at 600MHz. This board has 8MB SRAM and 128MB DRAM, with three 1Gbps Ethernet ports. We use one port as the client port and the other as a server port.

The server runs *Apache* [1] web server on an Intel 3.0GHz Dual Xeon with 1GB of memory. The client runs *httperf* [9] to generate HTTP requests. It is a Pentium 4 CPU running at 2.5GHz. All PCs are running Linux 2.4.20.

5.2 Comparison of Linux-based Switches

We measure the latency perceived by the client for a complete HTTP session as a function of request file size. As a base-line of this experiment, we also connect the client and server directly

with a cross-over cable. All the requested files are retrieved directly from the server's memory/cache. Figure 6 shows the average latency for various sizes of request files.

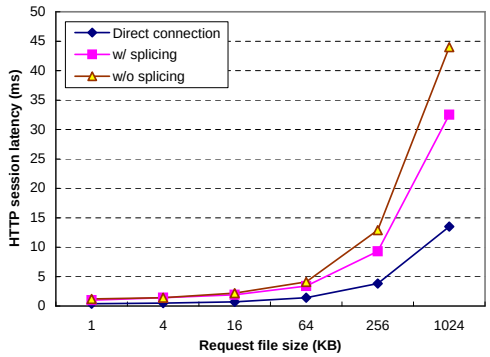

Figure 6: Latency of an HTTP session w/ and w/o splicing

It is observed that the latency for an HTTP session is reduced by using TCP splicing. The reduction is more apparent when we increase the file size. This is expected since a larger file involves more packet transmissions, which lead to more copying overhead in case of the switch without splicing. One exception is that there is no improvement for a small file size (e.g. 1K bytes). The reason is that the latency gain using splicing is too little to hide the overhead due to the splicing operation (recording the states for the two connections and closing the client-side and server-side sockets).

Table 4: Function level comparison for a data packet from server

Layer	w/o splicing		w/ splicing	
	Function	Latency (us)	Function	Latency (us)
IP	ip_rcv()	6	ip_rcv()	6
TCP	tcp_v4_rcv()	30	tcpsp_in()	13
App	read(),write()	16	N/A	N/A
TCP	tcp_sendmsg()	9	N/A	N/A
IP	ip_queue_xmit()	7	tcpsp_xmit()	6

We also observe that the control packet processing does not benefit from TCP splicing because TCP splicing does not alter the control packet operation. It is the data packet processing that determines the performance improvement. In order to further understand the splicing technique, we do a function level analysis and measure the time spent on different functions by instrumenting the Linux kernel. We identify key functions in IP, TCP and application layers from the protocol stack for this purpose. The MAC layer functions are not considered as they do not make any difference. We use a request file size as 16K bytes, and do similar experiments as the previous one. Table 4 shows the function level (protocol level) measurement for processing one data packet sent from the server (its size is determined by the MTU, which is 1500 bytes for Ethernet).

The functions we list here are encountered on a path of the response packet. Without splicing, the response packet goes up through the IP and TCP layer (functions `ip_rcv()` and `tcp_v4_rcv()`). The payload of this packet is copied to the user space when the application layer function `read()` is called. Till now the data is received by the proxy through the first connection. To send this data out through the second connection, function `write()` is called, which in turn calls TCP layer function `tcp_sendmsg()` to copy the payload from the user space to the kernel space. This packet finally goes down to the IP layer. With TCP splicing, after the response packet passes through the IP layer, it is executed by function `tcpsp_in()`, which modifies the IP and TCP headers. Then this updated packet is processed by function `tcpsp_xmit()`, which calls IP layer functions. The latency shown in the table clearly indicates how splicing improves the performance.

5.3 Comparison of Linux Splicer and SpliceNP

First we conduct experiments to obtain the latency of packet processing for an HTTP session in SpliceNP, and compare it with the Linux splicer. Some of the results also appear in our recent work [21] in the context of a content-aware switch design.

Figure 7 shows the latency of a Linux-based switch and an IXP-based switch when we vary the request file size. Note that this figure is different from Figure 6 in that the latency shown here is the processing latency on the switch. whereas Figure. 6 shows the complete HTTP latency measured from a client. We can see that the latency is reduced significantly by using IXP2400. It is reduced by 83.3% (0.6 ms to 0.1 ms) with a small file size as 1KB. And the larger the file size, the reduction is higher. At a very large file size as 1024KB, the latency is reduced by 89.5%.

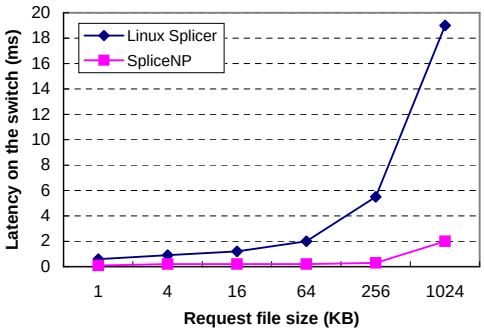

Figure 7: Latency comparison for an HTTP session

We measure the processing time for both the control and data packet. Read stamp instruction is used for this purpose. The timing starts from the point when the clientME/serverME takes the packet from the input queue, and ends at the moment when a processed packet is put into the output queue. For example, the latency on an ACK/Request packet records the period from the time when this packet is assembled in the DRAM, to the instant when the SYN packet to the server is put into the queue.

Table 5 shows the average processing latency on control packets and data packets for both the Linux-based and IXP-based switch. In a Linux-based switch, processing of the control packets takes much longer than that of the data packets. 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 takes the longest time to be processed. It is largely because this request packet, in addition to traversing the protocol stack, is copied into the user space, where it is parsed by the proxy running at the user level. The data packets in a Linux-based splicing do not travel through the TCP layer, hence consume much less time than control packets.

Processing of both control packets and data packets in an IXP-based switch takes much less time compared to 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

Table 5: Processing latency for control and data packets

Packet Type		IXP2400		Linux	Latency
		Microengine	Latency (us)	Latency (us)	reduction
Control Packet	SYN	clientME	7.2	48	85%
	ACK/Request	clientME	8.8	52	83%
	SYN/ACK	serverME	8.5	42	80%
Data Packet	Data	serverME	6.5	13.6	52%
	ACK	clientME	6.5	13.6	52%

and ACK packets from the client take the same time because we do not include data assembly during our measurement. This is done keeping in mind the latency measurement in Linux, where data assembly is not considered either. The improvement percentage for control and data packets are listed in the last column. The reduction on the latency for control packets and data packets is about 83% and 52%, respectively.

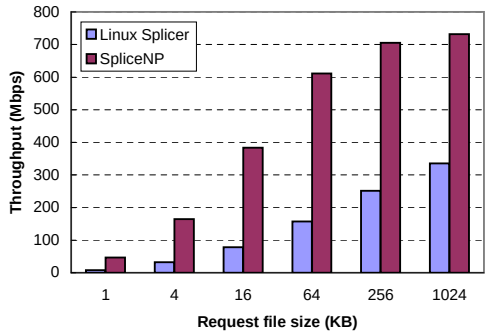

Figure 8: Throughput comparison for HTTP sessions

We also measure the throughput achieved by these two switches by sending requests of a uniform size as fast as possible from the clients. Figure 8 shows the result. We can see that the throughput is increased by 5.7x for a small size request like 1KB (8.2 Mbps to 46.4 Mbps). For a much larger file size like 1024KB, the improvement is 2.2x. Requests for smaller files have higher improvement than larger files because control packets take a larger portion in HTTP session for small files. Since the latency reduction for control packets is larger than that of data packets on SpliceNP, the improvement is more for small requests. As we increase the request file size, data packet processing becomes dominant, thus we see relatively smaller improvement on SpliceNP. It may be reminded here that we use only one clientME and one serverME to process the packets. The throughput may be further improved by using more microengines in the IXP2400.

6. RELATED WORK

TCP splicing has been studied extensively in the literature. Cohen et al. [4] implement a content-aware switch in Linux using TCP splicing. The two main components in their switch are an application level proxy (proxy-s) and a loadable kernel module (sp-mod). The proxy-s accepts TCP connections from the clients, and determines the destination server based on the clients' requests. Once the proxy-s establishes another connection with that server, it sends a splice command to the sp-mod, which splices the two connections together. The subsequent packets are forwarded at the IP level in the kernel. At this moment, the proxy-s is removed out of the data path. In this approach, data forwarding is performed in the kernel level, but the routing decision is still made at the application level.

Also using TCP splicing, Yang et al. [20] implement a content-aware distributor in Linux kernel between the network interface card (NIC) and the TCP/IP stack. The data forwarding as well as the routing decision are all performed at this level. This can avoid the overhead of passing the HTTP request packet through the protocol stack to the user level proxy as in [4]. Our approach, implemented on network processors, moves the whole processing further down to the NIC level, thus reduces the end-to-end latency as much as possible. Their distributor also exploits pre-forked connections, i.e. the switch pre-establishes some connections with the servers so that the second three-way handshaking is avoided. However, this optimization also has its downside. After the two connections are spliced, both the sequence number and the ACK number in the TCP header, and both the source and destination IP address in the IP header must be updated. Our approach, without this optimization, requires less changes in the packet header. In the TCP header, either the sequence number or the ACK number requires to be updated. In the IP header, either the source or the destination IP address needs to be updated. This leads to less computation in the TCP checksum and IP checksum calculation (incremental checksum [15]).

G. Apostolopoulos et al. [2] build a content-aware switch based on a switch core with custom built intelligent port controllers and a PowerPC processor. The PowerPC processor in this switch performs operations of connection setup and parsing the HTTP request. After the two connections are spliced, the powerPC is taken out of the data path and the port controllers handle all the packet processing. As an ASIC design, this switch can achieve very high throughput. However, it can hardly be extended to incorporate new services such as QoS scheduling.

Tammo Spalink et al. [17] suggest that TCP splicing processing be separated on a data forwarder and a control forwarder, which run on the IXP2400 microengines and the host processor (a Pentium), respectively. They also present some preliminary results for the data forwarder. However, our analysis shows that performing all the processing on the microengines give better performance. Therefore, not only data forwarder, but also the control forwarder are put on the microengines. The details of design and tradeoffs are in Section 4.

Compared to TCP splicing, TCP handoff [13] can release some load from the switch. However this approach requires that the TCP state machine inside the operating system in each of the servers be modified. This would be impractical to large scale server clusters.

Very recently, Papathanasiou et al. [14] exploit both the TCP splicing and hand-off techniques on a web switch. The switch performs TCP splicing whereas back-end servers perform the handoff operation. Their approach requires that a proxy application runs on each of the back-end servers, though no modification is required to the operating system.

7. CONCLUSIONS AND FUTURE WORK

In this paper, we first analyzed the performance of a Linux-based switch using TCP splicing technique, and showed that this tech-

nique can reduce the overhead considerably. To reduce its processing latency further, we implemented TCP splicing software on a network processor - Intel's IXP2400. We analyzed various tradeoffs in implementation and compared the performance of the NP-based switch with the Linux-based one. Our experimental results showed that the processing latency of SpliceNP is reduced by about 83.3% for a 1K byte web page. It also showed that the throughput can be improved by up to 5.7x. The SpliceNP software will be released to the public soon.

Our future work includes processing all the TCP options in the network processor since they can affect the performance. We plan to further breakdown the functionality of the ClientME/ServerME and assign them to more MEs, so that the processing can be parallelized and pipelined to improve the throughput. In addition, we plan to incorporate other functionalities such as Quality of Service (QoS) by identifying the packet flows and providing differentiated service to an individual flow.

Acknowledgement

This work is supported by NSF grants CCF-0220096 and 0233858, and grant from Intel Corporation.

8. REFERENCES

[1] Apache Software Foundation, http://www.apache.org

[2] G.Apostolopoulos, D.Aubespin V.Peris, P.Pradhan, D.Saha Design, Implementation and Performance of a Content-Based Switch proceedings of IEEE INFOCOM-2000

[3] Cisco Systems, Cisco Content Services Switch, http://www.cisco.com/en/US/products/hw/contnetw/ps789/prod_models_home.html

[4] A.Cohe, S.Rangarajan, H.Slye, On the Performance of TCP Splicing for URL-Aware Redirection. In Proceedings of the 2nd USENIX Symposium on Internet Technologies and Systems, Boulder, CO, Oct. 1999

[5] Erik J. Johnson and Aaron R. Kunze, IXP 1200 Programming The Microengine Coding Guide for the Intel IXP2400 network Processor Family, Intel Press

[6] Foundry Systems, Foundry ServerIron XL/G, http://www.b2net.co.uk/foundry/foundry_serveriron_xlg_web_switch.htm

[7] Tom Halfhill, Intel Network Processor Targets Routers, Microprocessor Report, September 1999

[8] Linux Virtual Server Project, http://www.linuxvirtualserver.org

[9] David Mosberger and Tai Jin, HP Research Labs A Tool for Measuring Web Server Performance, 1998

[10] David A. Maltz, Pravin Bhagwat, TCP Splicing for Application Layer Proxy Performance, IBM Research Report RC 21139, 1998

[11] Netfilter, http://www.netfilter.org

[12] Nortel Networks, Alteon Web Switches, http://www.nortelnetworks.com/products/01/alteon/webswitch/index.html

[13] V.S. Pai, M.Aron, G.Banga, M.Svendsen, P.Druschel, W.Zwaenepoel, E.Nahum, Locality-Aware Request Distribution in Cluster-based Network Servers. In Proceedings of the 8th Conference on Architectural Support for Programming Languages and Operating Systems, San Jose, CA, Oct.1998

[14] Athanasios E. Papathanasiou, Eric Van Hensbergen, KNITS: Switch-based Connection Hand-off, Twenty-First Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE , Volume: 1 , 2002

[15] RFC1624: Computation of the Internet Checksum via Incremental Update, May 1994

[16] Marcel-Catalin Rosu, Daniela Rosu, Kernel Support for Faster Web Proxies, USENIX Annual Technical Conference, June 2003

[17] Tammo Spalink, Scott Karlin, Larry Peterson, Yitzchak Gottlieb, Building a Robust Software-Based Router Using Network Processors, Proceedings of the eighteenth ACM symposium on Operating systems principles, pages 216 - 229, 2001

[18] Oliver Spatscheck, et al., Optimizing TCP Forwarder Performance, IEEE/ACM Transactions on Networking, 2000

[19] The Linux Kernel Archives, http://www.kernel.org

[20] Chu-Sing Yang and Mon-Yen Luo, Efficient Support for Content- Based Routing in Web Server Clusters. In Proceedings of the 2nd USENIX Symposium on Internet Technologies and Systems, Boulder, CO, October 1999

[21] Li Zhao, Yan Luo, Laxmi Bhuyan and Ravi Iyer, Design and Implementation of A Content-aware Switch using A Network Processor. In Proceedings of the 13th IEEE Symposium on High Performance Interconnects, Stanford University, CA, August 2005