SpliceNP: A TCP Splicer using a Network Processor

Li Zhao*, Yan Luo*, Laxmi Bhuyan
University of California Riverside
Ravi Iyer
Intel Corporation

* Now with Intel Corporation
* Now with University of Massachusetts Lowell
Outline

- Background and Motivation
- Design and Implementation
- Performance Evaluation
- Conclusions
Background

- Front-end of a web cluster, one VIP
- Route packets based on layer 5 information
  - Examine application data in addition to IP & TCP

Problem with HTTP Proxy
- Overhead to copy data between two connections is high
  - Data goes up through protocol stack
  - Data is copied from kernel space to user space
  - Data is copied back to kernel and goes down through protocol stack
TCP Splicing

- A technique to splice two TCP connections inside the kernel
- Data relaying between the two connections can be speeded up
- Can be used to speed up layer-5 switching, web proxy and firewall running in the user space
- Example: content-aware switch
Anatomy of TCP Splicing

Without TCP Splicing

With TCP Splicing

Bookkeeping of connection states, selection of servers, state migration

SEQ # translation
Checksum Recalculation
Etc.
State Transition in TCP Splicing

(1) 3-way handshaking with client

(2) Application specific operations

(3) 3-way handshaking with server

(4) Spliced!

(5) Connection Tear down

SpliceNP: A TCP Splicer using a Network Processor
Outline

- Background and Motivation
- Design and Implementation
  - Design Options
  - Implementation on IXP2400
- Performance Evaluation
- Conclusions
Processing Elements for TCP Splicing

- **ASIC (Application Specific Integrated Circuit)**
  - High processing capacity
  - Long time to develop
  - Lack the flexibility

- **GP (General-purpose Processor)**
  - Programmable
  - Overheads:
    - interrupt, moving packets over PCI bus, ISA not optimized

- **NP (Network Processor)**
  - Optimized ISA for packet processing, multiprocessing and multithreading → high performance
  - Programmable
Background on NP

- Architecture
  - Control processor (CP): embedded processor, maintain control information, process exception packets
  - Data processors (DPs): tuned specifically for packet processing
  - Communicate through shared SRAM and DRAM

- NP operation
  - Packet arrives in receive buffer
  - Packet Processing by CP or DPs
  - Transfer the packet onto wire after processing
Design Options

- **Option (a):** GP-based (Linux-based) switch
  - Overhead of moving data across PCI bus
- **Option (b):** CP creates and splices connections, DPs process packets sent after splicing
  - Connection setup & splicing is more complex than data forwarding
  - However, control packets need to be passed through DRAM queues
- **Option (c):** DPs handle connection setup, splicing & forwarding
Outline

- Background and Motivation
- Design and Implementation
  - Design Options
  - Implementation on IXP2400
- Performance Evaluation
- Conclusions
IXP 2400 Architecture and Development Challenge

Architecture
- XScale core
- 8 Microengines (MEs)
- Each ME
  - run up to 8 threads
  - 4K instruction store
  - Local memory
- Scratchpad memory, SRAM & DRAM controllers

Challenges
- Limited size of instruction memory
- Distribute data to versatile memory units
- C compiler not available for IXP2400
Resource Allocation

- Client-side control block list
  - record states for connections between clients and SpliceNP, states after splicing
- Server-side control block list
  - record states for connections between server and SpliceNP

SRAM (8MB)
- Client side CB list
- Server side CB list
- server selection table
- Locks

DRAM (256MB)
- Packet buffer

Scratchpad (16KB)
- Packet queues

Microengines
- Rx ME
- Client ME
- Server ME
- Tx ME
Comparison of Functionality (I)

- A lite version of TCP due to the limited instruction size of microengines.

Processing a SYN packet

<table>
<thead>
<tr>
<th>Step</th>
<th>Functionality</th>
<th>TCP</th>
<th>Linux Splicer</th>
<th>SpliceNP</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Dequeue packet</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>2</td>
<td>IP header verification</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>3</td>
<td>IP option processing</td>
<td>Y</td>
<td>Y</td>
<td>N</td>
</tr>
<tr>
<td>4</td>
<td>TCP header verification</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>5</td>
<td>Control block lookup</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>6</td>
<td>Create new socket and set state to LISTEN</td>
<td>Y</td>
<td>Y</td>
<td>No socket, only control block</td>
</tr>
<tr>
<td>7</td>
<td>Initialize TCP and IP header template</td>
<td>Y</td>
<td>Y</td>
<td>N</td>
</tr>
<tr>
<td>8</td>
<td>Reset idle time and keep-alive timer</td>
<td>Y</td>
<td>Y</td>
<td>N</td>
</tr>
<tr>
<td>9</td>
<td>Process TCP option</td>
<td>Y</td>
<td>Y</td>
<td>Only MSS option</td>
</tr>
<tr>
<td>10</td>
<td>Send ACK packet, change state to SYN_RCVD</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
</tbody>
</table>
## Comparison of Functionality (II)

**Processing a SYN/ACK packet**

<table>
<thead>
<tr>
<th>Step</th>
<th>Functionality</th>
<th>TCP</th>
<th>Linux Splicer</th>
<th>SpliceNP</th>
</tr>
</thead>
<tbody>
<tr>
<td>1-5</td>
<td>Same as processing a SYN packet</td>
<td>Y</td>
<td>Y</td>
<td>N</td>
</tr>
<tr>
<td>2</td>
<td>Reset idle time and keep-alive timer</td>
<td>Y</td>
<td>Y</td>
<td>N</td>
</tr>
<tr>
<td>3</td>
<td>Process TCP option</td>
<td>Y</td>
<td>Y</td>
<td>Only MSS option</td>
</tr>
<tr>
<td>4</td>
<td>Verify ACK numbers and flags</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>5</td>
<td>Connection establishment timer</td>
<td>Y</td>
<td>Y</td>
<td>N</td>
</tr>
<tr>
<td>6</td>
<td>Initialize receive sequence number</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>7</td>
<td>Set state to ESTABLISHED</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>8</td>
<td>Send ACK packet</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
</tbody>
</table>
# Comparison of Functionality (III)

## Processing a data packet

<table>
<thead>
<tr>
<th>Step</th>
<th>Functionality</th>
<th>TCP</th>
<th>Linux Splicer</th>
<th>SpliceNP</th>
</tr>
</thead>
<tbody>
<tr>
<td>1-5</td>
<td>Same as processing a SYN packet</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>Reset idle time and keep-alive timer</td>
<td>Y</td>
<td>Y</td>
<td>N</td>
</tr>
<tr>
<td>7</td>
<td>Process TCP option</td>
<td>Y</td>
<td>Y</td>
<td>Only MSS option</td>
</tr>
<tr>
<td>8a</td>
<td>Wake up receiving process</td>
<td>Y</td>
<td>Direct forwarding</td>
<td>Direct forwarding</td>
</tr>
<tr>
<td></td>
<td>Copy data to application</td>
<td>Y</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>8b</td>
<td>Delete acknowledged data from send buffer</td>
<td>Y</td>
<td>Direct forwarding</td>
<td>Direct forwarding</td>
</tr>
<tr>
<td></td>
<td>Wake up waiting process</td>
<td>Y</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>9</td>
<td>Flow control processing</td>
<td>Y</td>
<td>Y</td>
<td>N</td>
</tr>
</tbody>
</table>

SpliceNP: A TCP Splicer using a Networ Processor
Outline

- Background and Motivation
- Design and Implementation
- Performance Evaluation
- Conclusions
Experimental Setup

- Radisys ENP2611 containing an IXP2400
  - XScale & ME: 600MHz
  - 8MB SRAM and 128MB DRAM
  - Three 1Gbps Ethernet ports: 1 for Client port and 2 for Server ports
- Server: Apache web server on an Intel 3.0GHz Xeon processor
- Client: Httperf on a 2.5GHz Intel P4 processor
- Linux-based switch
  - Loadable kernel module
  - 2.5GHz P4, two 1Gbps Ethernet NICs
Latency on a Linux-based TCP Splicer

- Latency is reduced by TCP splicing
Latency vs Request File Size

- Latency reduced significantly
  - 83.3% (0.6ms → 0.1ms) @ 1KB
- The larger the file size, the higher the reduction
  - 89.5% @ 1MB file
Comparison of Packet Processing Latency

Table 5: Processing latency for control and data packets

<table>
<thead>
<tr>
<th>Packet Type</th>
<th>IXP2400</th>
<th>Linux</th>
<th>Latency reduction</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Microengine</td>
<td>Latency (us)</td>
<td>Latency (us)</td>
</tr>
<tr>
<td>Control</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Packet</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SYN</td>
<td>clientME</td>
<td>7.2</td>
<td>48</td>
</tr>
<tr>
<td>ACK/Request</td>
<td>clientME</td>
<td>8.8</td>
<td>52</td>
</tr>
<tr>
<td>SYN/ACK</td>
<td>serverME</td>
<td>8.5</td>
<td>42</td>
</tr>
<tr>
<td>Data</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Packet</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Data</td>
<td>serverME</td>
<td>6.5</td>
<td>13.6</td>
</tr>
<tr>
<td>ACK</td>
<td>clientME</td>
<td>6.5</td>
<td>13.6</td>
</tr>
</tbody>
</table>
## Analysis of Latency Reduction

<table>
<thead>
<tr>
<th>Linux-based</th>
<th>NP-based</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Interrupt: NIC raises an interrupt once a packet comes</strong></td>
<td>polling</td>
</tr>
<tr>
<td><strong>NIC-to-mem copy</strong></td>
<td>No copy: Packets are processed inside without two copies</td>
</tr>
<tr>
<td>Xeon 3.0Ghz Dual processor w/ 1Gbps Intel Pro 1000 (88544GC) NIC, 3 us to copy a 64-byte packet by DMA</td>
<td></td>
</tr>
<tr>
<td><strong>Linux processing: OS overheads</strong></td>
<td>IXP processing: Optimized ISA 6.5 us</td>
</tr>
<tr>
<td>Processing a data packet in splicing state: 13.6 us</td>
<td></td>
</tr>
</tbody>
</table>
Throughput vs Request File Size

- Throughput is increased significantly
  - 5.7x for small file size @ 1KB, 2.2x for large file @ 1MB
- Higher improvement for small files
  - Latency reduction for control packets > data packets
  - Control packets take a larger portion for small files
Conclusions

- Implemented TCP splicing on a network processor
- Analyzed various tradeoffs in implementation and compared its performance with a Linux-based TCP splicer
- Measurement results show that NP-based switch can improve the performance significantly
  - Process latency reduced by up to 83%
  - Throughput improved by up to 5.7x
Thank you!

Questions?
TCP Splicing

\[\text{client} \quad \text{content switch} \quad \text{server}\]

- **Step 1:** SYN(CSEQ)
- **Step 2:** SYN(DSEQ)
- **Step 3:** ACK(CSEQ+1)
- **Step 4:** DATA(CSEQ+1)
- **Step 5:** ACK(CSEQ+1)
- **Step 6:** SYN(CSEQ)
- **Step 7:** DATA(CSEQ+1)
- **Step 8:** ACK(CSEQ+lenR+1)

\[\text{lenR}: \text{size of http request.} \quad \text{lenD}: \text{size of return document}\]
TCP Handoff

- Migrate the created TCP connection from the switch to the back-end server
  - Create a TCP connection at the back-end without going through the TCP three-way handshake
  - Retrieve the state of an established connection and destroy the connection without going through the normal message handshake required to close a TCP connection
- Once the connection is handed off to the back-end server, the switch must forward packets from the client to the appropriate back-end server