Networking Operating System from Scratch towards High-Performance COTS Network Facilities

Hirochika Asai <panda@jar.jp>
The University of Tokyo
IIJ-II Seminar, Tokyo, Japan
April 9, 2015
Biography

- Hirochika Asai (panda)

  - Professional history
    - 2013: Received Ph.D in Information Science and Technology from the University of Tokyo
      - “Analysis and Management of the Internet based on Data Flow Profiling”
    - 2013-now: Project Assistant Professor at the University of Tokyo
    - 2014-now: Board member of WIDE Project

  - Research interests
    - Operating system (networking)
      - had been my hobby...
    - Distributed system (especially Internet-wide system)
    - Internet traffic and topology analysis
Trends on Network Functionalities by Software on COTS hardware

- **SDN: Software Defined Network**
  - Separation of
    - Forwarding Plane; by hardware
    - Control Plane; by *software*

- **NFV: Network Function Virtualization**
  - Network function by *software* with virtualization technologies (e.g., virtual machine, container, process)
Networking Operating System

- “Operating System” (OS)
  - Fundamental system software in charge of
    - Resource management (hardware/software)
    - Protection, Filesystem, multitasking etc.

- COTS Network Facilities (using generic CPU)
  = Networking Operating System
  - Inexpensive
  - Flexible
  - Extensible
Departing from Generic OS

Generic OS: Not designed for networking facilities

Fat kernel, many overhead, dirty-slate

Clean-slate approach
Networking Operating System from Scratch

“from scratch”

1. Evaluate the best performance of COTS hardware
   - Bottleneck analysis

2. Design new algorithm/architecture
   - Scheduler
   - Memory management
   - Protection
   - Protocol stack
   - Routing table lookup algorithm
Towards High-Performance Network Facilities with COTS hardware
Network Facilities with COTS hardware

- **Background**
  - Generic CPU (IA) for packet processing
  - PCIe NIC for packet forwarding

- **Goal:** High-performance network facilities w/ software
  - Router: 40GbE/100GbE line-rate routing (1M RiB entries)
  - Middlebox: Firewall, Load-balancer, etc.
  - Server apps: HTTP, Authentication, Accounting, etc.
Network Facilities with COTS hardware

- Background
  - Generic CPU (IA) for packet processing
  - PCIe NIC for packet forwarding
- Goal: High-performance network facilities w/ software
  - Router: 40GbE/100GbE line-rate routing (1M RiB entries)
  - Middlebox: Firewall, Load-balancer, etc.
  - Server apps: HTTP, Authentication, Accounting, etc.
VF SR: Very Fast Software Router

Essential Components

1. Fast packet forwarding
   - High-rate per core/port for in-order processing

2. Fast IP routing table lookup
   - # of routes: >512k (envisioning >800k)
   - High-rate per core for in-order processing
Key Numerical Values of “fast”: Packet Rate for 10/40/100GbE

- Ethernet
  - Minimum frame length: 64-Byte (=Maximum frame rate)
    - 1GbE: 1.488Mpps = 672 ns/packet
    - 10GbE: 14.88Mpps = 67.2 ns/packet
    - 40GbE: 59.52Mpps = 16.8 ns/packet
    - 100GbE: 148.8Mpps = 6.72 ns/packet
VFSR: Very Fast Software Router

- Essential Components

1. Fast packet forwarding
   - High-rate per core/port for in-order processing

2. Fast IP routing table lookup
   - # of routes: >512k (envisioning >800k)
   - High-rate per core for in-order processing
Myths on Packet Forwarding

- Bottlenecks in packet forwarding
  - CPU is slow.
  - Memory copy is so heavy.
  - Interrupts incur excessive overheads.

H. Asai, "Networking Operating System from Scratch"
Myths on Packet Forwarding

- Bottlenecks in packet forwarding
  - CPU is slow.
    - Yes, for packet processing, but forwarding requires only a set of simple instructions
      - e.g., 0.3 ns / CPU cycle @ 3.3GHz CPU
  - Memory copy is so heavy.
  - Interrupts incur excessive overheads.
Myths on Packet Forwarding

- Bottlenecks in packet forwarding
  - CPU is slow.
    - Yes, for packet processing, but forwarding requires only a set of simple instructions
      - e.g., 0.3 ns / CPU cycle @ 3.3GHz CPU
  - Memory copy is so heavy.
    - At least, throughput is enough.
      - e.g., DDR3-1866 Dual Channel: 29.867GB/s (238.933Gbps)
  - Interrupts incur excessive overheads.
Myths on Packet Forwarding

- Bottlenecks in packet forwarding
  - CPU is slow.
    - Yes, for packet processing, but forwarding requires only a set of simple instructions
      - e.g., 0.3 ns / CPU cycle @ 3.3GHz CPU
  - Memory copy is so heavy.
    - At least, throughput is enough.
      - e.g., DDR3-1866 Dual Channel: 29.867GB/s (238.933Gbps)
  - Interrupts incur excessive overheads.
    - Not excessive, but non-negligible for 100 GbE
      - Discuss this later
Real Bottleneck on Packet Forwarding

- PCIe device register access
  - Memory Mapped I/O (MMIO)
    - No cache
      - ~250ns/access [Miller et al. ACM ANCS ’09]

- Read
  - 1529.17 cycles / read
  - 392.1 ns / read

- Write
  - 282.621 cycles / write
  - 72.47 ns / write

※Measure CPU cycles to access to the same register
1 million times by Performance Monitoring Counter (PMC)

CPU: Intel Core i7 4770K
Memory: Corsair DDR3-1866 8GB x4
NIC: Intel X520-DA2
Review: Generic NIC Architecture
Review: Generic NIC Architecture

Packet reception
1. NIC receives a packet
2. NIC transfers the packet data to a buffer in RAM via DMA
3. NIC proceeds the head pointer
4. Software processes the packet
5. Software proceeds the tail pointer to release the packet

April 9th, 2015  H. Asai, "Networking Operating System from Scratch"
Review: Generic NIC Architecture

Packet transmission
1. Software writes a packet to a buffer in RAM
2. Software proceeds the tail pointer to commit the packet
3. NIC transfer the packet data from the buffer in RAM via DMA
4. NIC transmit the packet
5. NIC proceeds the head pointer to notify the packet is transmitted
Review: Generic NIC Architecture

Packet reception
1. NIC receives a packet
2. NIC transfers the packet data to a buffer in RAM via DMA
3. NIC proceeds the head pointer
4. Software processes the packet
5. Software proceeds the tail pointer to release the packet
Review: Generic NIC Architecture

Packet transmission
1. Software writes a packet to a buffer in RAM
2. Software proceeds the tail pointer to commit the packet
3. NIC transfers the packet data from the buffer in RAM via DMA
4. NIC transmits the packet
5. NIC proceeds the head pointer to notify the packet is transmitted
Polling & Bulk Processing
(Transmission, Intel® X520)

```c
txq_tail = 0;
for ( ;; ) {
    txq_head = read_txq_head();
    /* Available Tx queue length */
    txq_len = txq_sz
    - (txq_sz - txq_head + txq_tail) % txq_sz;
    /* Check the available Tx queue length */
    if ( txq_len < n ) continue;
    for ( i = 0; i < n; i++ ) {
        // Set packet to the ring buffer to txq_tail
        txq_ring[txq_tail].pkt = pkt_to_transmit;
        txq_tail = (txq_tail + 1) % txq_sz
    }
    /* Commit */
    write_txq_tail(txq_tail);
}
```

Note: Can be optimized...

~392.1ns

~72.47ns
Tx Performance by bulk size (Intel® X520)

Note: Also confirmed 59.52 Mpps Tx (2 Intel® X520-DA2) @ 1 core from Intel® Core i7-4770K

April 9th, 2015 H. Asai, "Networking Operating System from Scratch"
Intel® XL710’s Operation

Transmission

<table>
<thead>
<tr>
<th>Host</th>
<th>NIC (PCIe)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1)</td>
<td>(1)</td>
</tr>
<tr>
<td>(2)</td>
<td>(2)</td>
</tr>
<tr>
<td>(3)</td>
<td>(3)</td>
</tr>
</tbody>
</table>

Reception

(1) Write the tail pointer (MMIO write)
(2) Transfer the packets via DMA
(3) Write-back the transfer status
Polling & Bulk Processing
(Transmission, Intel® XL710)

txq_tail = 0;
for ( ;; ) {
    completed = check_wb_status(txq_tail, n);
    /* Check the transmission is completed */
    if ( !completed ) continue;
    for ( i = 0; i < n; i++ ) {
        // Set packet to the ring buffer to
        txq_ring[txq_tail].pkt = pkt_to_transmit;
        txq_tail = (txq_tail + 1) % txq_sz
    }
    /* Commit */
    write_txq_tail(txq_tail);
}
Intel® XL710’s performance

![Graph showing performance of Intel® XL710 with different bulk sizes and QP values. The graph illustrates the line-rate and performance in Mpps for #QP = 1, 2, 3, with bulk sizes from 1 to 6.](image)
Same Strategy for Forwarding (Routing for 1 route)

Throughput [Gbps]

Frame size [byte]

My implementation
Linux
Line rate

Bulk polling & transmission
(dynamic bulk size = received queue length)

※ 1 route
TTL and checksum calculation
are done by CPU

Transmitter (pix)

Router (pix)

Hardware switch
(interface counter for evaluation)
Latency measurement

- **Experimental setup**
  - **Tester**
    - Spirent Communications Spirent TestCenter
      - Chassis: SPT-N4U-110
      - Module: CV-10G-S8
    - Supported by 株式会社東陽テクニカ様 during Interop Tokyo 2014
Low Latency

- Low latency (~10us) for 90% of line-rate traffic

- 0.001Mpps loss (need investigation...)

Graph shows the latency in microseconds for different test traffic rates (64-byte frame) in Gbps.
Revisiting the Overhead of Interrupts for Faster Packet Processing & I/O

Push 15 general purpose registers onto the stack, pop 15 general purpose registers from the stack, and then return to the restore point while popping the original stack pointer etc.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Latency</th>
<th>Throughput</th>
</tr>
</thead>
<tbody>
<tr>
<td>PUSH (@0F_2H)</td>
<td>1.5</td>
<td>1</td>
</tr>
<tr>
<td>POP (@0F_2H)</td>
<td>1.5</td>
<td>1</td>
</tr>
<tr>
<td>CLI (@06_2A/2D)</td>
<td>5</td>
<td>2</td>
</tr>
</tbody>
</table>

30 CPU cycles for push/pop instructions ➔ 10 ns @3GHz CPU

Referred from Intel® 64 and IA-32 Architectures Optimization Reference Manual
Interim Summary

- Faster packet forwarding
  - Reduce slow PCIe MMIO
    - Key: Bulk processing
      - Read
        - 392.1 ns / read
      - Write
        - 72.47 ns / write
  - Avoid using interrupt handlers for 40GbE/100GbE
    - Key: Polling, Tickless
      - 10 ns to save and restore CPU’s registers
VFSR: Very Fast Software Router

- Essential Components

1. Fast packet forwarding
   - High-rate per core/port for in-order processing

2. Fast IP routing table lookup
   - # of routes: >512k (envisioning >800k)
   - High-rate per core for in-order processing
Poptrie: A Compressed Trie with Population Count for Fast and Scalable Software IP Routing Table Lookup

Hirochika Asai (Univ. of Tokyo)
Yasuhiro Ohara (NTT Communications)
Fundamental Algorithm for Longest Prefix Match

Binary Radix Tree

Problem with binary radix tree
- Depth up to 32 (for IPv4)
- Too many pointers

→ Slow
Principle Ideas towards Faster IP Routing Table Lookup Algorithm

- **Reduce the number of instruction**, especially memory access
  - 1 or a few cycles for most of bitwise instructions
  - Memory access latency (in Intel Core i7-4770K)
    - L1 cache: 4-5 cycles
    - L2 cache: 12 cycles
    - L3 cache: 27.85 cycles
    - DRAM: ~65 ns

- **Reduce memory footprint**
  - Maximize CPU cache efficiency
    - L1/L2/L3 cache size in Intel Core i7-4770K
      - 64 KiB, 256 KiB, 8 MiB
Reduce the number of instructions: Starting from $2^k$-ary radix tree

To reduce the depth of the tree (i.e., # of memory access)

The necessary descendant internal or leaf nodes are placed in the internal node so that it can be contained completely within the CPU cache, so that the performance of Poptrie is due to the small memory footprint. The admirable aggregation", is not our contribution, and is applicable to other lookup technologies as well. Unless otherwise noted, the aggregation option.

First, the descendant array in the multiway trie is changed to be a bitwise array (i.e., a bit vector) in Poptrie. The implementation, the contiguous arrays of internal and leaf nodes are aggregated, is not our contribution, and is applicable to other lookup technologies as well. Unless otherwise noted, the aggregation option.

Figure 2 illustrates an example of the indirect index of internal and leaf nodes with the total size of an internal node.

In the basic poptrie contains the internal node, so the size of an internal node is then 24 bits of the key IP address. An element in the descendant array points to its next-level child internal node or the leaf node. At the depth of the tree.

The value $n$-th bit in the vector $\text{leafvec}_0$ is necessary to provide the starting point of the descendant array. Here, we can utilize the CPU instruction to accelerate the calculation of the number of 1s in the least significant chunk of the address chunk. Each bit in the value $n$-th chunk in the key IP address. An element in the descendant array, corresponding to the value of the corresponding bits in the $n$-th chunk of the address chunk. Each bit in the value $n$-th chunk in the key IP address.

The starting offset for the later evaluation, unless otherwise noted explicitly. In the aggregation option.

The size of leaf nodes in Section 3.3, and the additional operations in Section 3.4. The internal node in the basic poptrie contains $2$ base1 elements in ascending order up to $2$ base1. Each node holds $2$ base1 elements in the current level, and the bit-1 indicates a descendant node, corresponding to the value of the corresponding child node: the bit is set to 1 if the corresponding child node is a leaf node, and it is set to 0 if the corresponding child node is an internal node, and then if there is no descendant internal node, the bit-0 indicates a leaf node.

The array can start with $2$ base0, so that they form a contiguous array. The array starts with $2$ base0 base1 value 00b 01b 10b 11b, with corresponding child is an internal node, and it is set to 0 if the corresponding child is a leaf node. In other words, the bit-0 indicates a leaf node.

The starting offset for the later evaluation, unless otherwise noted explicitly. In the aggregation option.

The admirable aggregation", is not our contribution, and is applicable to other lookup technologies as well. Unless otherwise noted, the aggregation option.
Reduce memory footprint: Pointer Compression w/ Population Count

Index: # of 1’s bits in the least significant N bits

Index: # of 0’s bits in the least significant N bits

Which k? 64-bit CPU \( \Rightarrow k=6 \) (so that vector is in \( 2^6 = 64 \) bits)

April 9th, 2015
H. Asai, "Networking Operating System from Scratch"
Further Compression: Leaf Vector to Remove Redundant Leaf Nodes

One of the problem with the basic data structure
- Redundant leaf nodes for prefixes that do not match k-bit boundary
- e.g., /1 (/7, etc. as well) may create 32 redundant leaf nodes when k=6

![Diagram](image-url)
Visualized Lookup Algorithm Example

For the destination address 0110b
Direct Pointing

(a) Without Direct Pointing.  (b) Direct Pointing.

Lookup $s$ bits at the first stage (like other algorithms)
Evaluation for Random Traffic

REAL-Tier1-A: Global Tier-1’s BGP Router
REAL-Tier1-B: Domestic ISP’s BGP Router

Look-up rate [Mbps]

# of threads

April 9th, 2015
H. Asai, "Networking Operating System from Scratch"
Performance Evaluation for Real Traffic (WIDE Transit)

![Graph showing lookup rates for different data structures and algorithms.]

April 9th, 2015

H. Asai, "Networking Operating System from Scratch"
Detailed Analysis on CPU Cycles per Lookup for Random Traffic

CPU cycles per lookup vs. Binary radix depth for different data structures:

- SAIL
- D16R
- D18R
- Poptrie
- Poptrie

April 9th, 2015  H. Asai, "Networking Operating System from Scratch"
Structural Scalability

Lookup performance for random traffic [Mlps]

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>SYN1</th>
<th>SYN1</th>
<th>SYN2</th>
<th>SYN2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>-Tier1-A</td>
<td>-Tier1-B</td>
<td>-Tier1-A</td>
<td>-Tier1-B</td>
</tr>
<tr>
<td>SAIL</td>
<td>102.86</td>
<td>99.98</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>D18R (modified)</td>
<td>115.45</td>
<td>117.48</td>
<td>102.59</td>
<td>104.22</td>
</tr>
<tr>
<td>Poptrie$_{18}$</td>
<td>188.02</td>
<td>187.69</td>
<td>174.42</td>
<td>175.04</td>
</tr>
</tbody>
</table>

RIB dataset (synthetic RIB dataset)

<table>
<thead>
<tr>
<th>Name</th>
<th># of prefixes</th>
<th># of nhops</th>
<th>Name</th>
<th># of prefixes</th>
<th># of nhops</th>
</tr>
</thead>
<tbody>
<tr>
<td>SYN1-Tier1-A</td>
<td>764,847</td>
<td>45</td>
<td>SYN2-Tier1-A</td>
<td>885,645</td>
<td>87</td>
</tr>
<tr>
<td>SYN1-Tier1-B</td>
<td>756,406</td>
<td>19</td>
<td>SYN2-Tier1-B</td>
<td>876,944</td>
<td>33</td>
</tr>
</tbody>
</table>
Interim Summary

- Fast IP routing table lookup
  - 914 Mlps w/ 4 core
    - Global tier-1 ISP’s full route (531k routes)
    - Random traffic
  - 175 Mlps per core
    - Synthetic 800k routes
    - Random traffic
Ongoing Project

- **Socket API extension for middleboxes/VNF (Virtualized Network Function)**
  - **Virtual machine** ← ETSI’s approach
    - Network abstraction: Virtual NIC
      - Pros: Any kinds of OS works
      - Cons: Overhead of virtualization (incl. VMEntry/VMExit)
  - **Container**
    - Network abstraction: Virtual NIC
      - Pros: Linux works (when we use Linux Container)
      - Cons: Overhead of virtualized NIC driver
  - **Process** ← My focus
    - Network abstraction: Socket API
      - Pros: No overhead (a few scheduler overhead) in my design
      - Cons: Socket API is not good for packet processing
Non-TCP/UDP Socket

- Existing socket / IPPROTO
  - SOCK_RAW
    - Privileged socket...
  - SOCK_DGRAM (IPPROTO_UDP) / SOCK_STREAM (IPPROTO_TCP)
    - Basically UDP/TCP (Cannot handle Ethernet, IP)

- Socket
  - SOCK_DGRAM + IPPROTO_ETHERNET (IPPROTO?)
    - Bind a MAC address
  - SOCK_DGRAM + IPPROTO_IP (IPPROTO?)
    - Bind an IP address
Conclusion

- **VFSR: Very Fast Software Router**
  - Essential Components
    1. Fast packet forwarding
       - High-rate per core/port for in-order processing
    2. Fast IP routing table lookup
       - # of routes: >512k (envisioning >800k)
       - High-rate per core for in-order processing

- **Socket API extension for process-based NFV**
  - as ongoing work