67886 - Switch and Router Design

Dr. David Hay
Ross 78b
dhay@cs.huji.ac.il

Packet Processing Examples

• Address Lookup (IP/Ethernet)
  – Where to send an incoming packet?
    "Use output-port 3, to send packets to MAC address 01:23:45:67:89:ab" – Exact Match
    "Use output-port 4, to send packets to destination network 111.15/16" – (Longest Prefix Match)

• Firewall, ACL
  – Which packet to accept or deny?
    "Drop all packets from evil source network 66.66/16 on ports 6-666"
  – Usually needs 5 fields: source-address, dest-address, source-port, dest-port, protocol

Packet Processing Rate

<table>
<thead>
<tr>
<th>Year</th>
<th>Line</th>
<th>40B packets (Mpkts/s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1997</td>
<td>622Mb/s</td>
<td>1.94</td>
</tr>
<tr>
<td>1999</td>
<td>2.5Gb/s</td>
<td>7.81</td>
</tr>
<tr>
<td>2001</td>
<td>10Gb/s</td>
<td>31.25</td>
</tr>
<tr>
<td>2003</td>
<td>40Gb/s</td>
<td>125</td>
</tr>
</tbody>
</table>

1. Lookup mechanism must be simple and easy to implement
2. (Surprise?) Memory access time is the long-term bottleneck

Memory Technology (2003-04)

<table>
<thead>
<tr>
<th>Technology</th>
<th>Single chip density</th>
<th>$/chip ($/MByte)</th>
<th>Access speed</th>
<th>Watts/chip</th>
</tr>
</thead>
<tbody>
<tr>
<td>Networking DRAM</td>
<td>64 MB</td>
<td>$30-$50 ($0.50-$0.75)</td>
<td>40-80ns</td>
<td>0.5-2W</td>
</tr>
<tr>
<td>SRAM</td>
<td>4 MB</td>
<td>$20-$30 ($5-$8)</td>
<td>4-8ns</td>
<td>1-2W</td>
</tr>
<tr>
<td>TCAM</td>
<td>1 MB</td>
<td>$200-$250 ($200-$250)</td>
<td>4-8ns</td>
<td>15-30W</td>
</tr>
</tbody>
</table>

Note: Price, speed and power are manufacturer and market dependent.
Numbers are a bit outdated but give the general idea

Bottlenecks
Memory, memory, ...

- Bottlenecks – Watts/Megabyte
  - Bottlenecks – Watts/Megabyte
Simplest Task: Exact Matching

- Mostly in bridges
  - Bridges works in layer 2 (Ethernet)
  - Bridges connects two Ethernet networks
- Wire-speed forwarding:
  - Each time a packet arrives at a bridge, forward it according to the destination MAC address
  - Store/update also the source MAC address (learning)
  - Should be done at wire speed

Solution 1: Binary Search

- MAC addresses have values which can be sorted
- Thus, when keeping them sorted, one can perform a binary search on the array and find the right MAC address
- However, each iteration is a memory access $\Rightarrow \log N$ memory accesses $\Rightarrow$ works fine (even using DRAM) for small speed, N (around 10Mb/s, 8K values) but doesn’t scale for large N/higher speeds (not even for 100 Mb/s, 64K values)
- Using faster hardware (SRAM) won’t really solve the problem (and it is more expensive...)

Scaling using Hashing

- Hashing is much faster than binary search on average, however much slower on the worst case (up to linear time...)
- However, one can choose (pre-compute) good hash functions, so the number of collision can be small and bounded
  - Precomputation takes a lot of time, but addresses are not added in rapid rate
  - Applying the hash functions is done on wire-speed
- More sophisticated data structure/hashing techniques can also be applied (e.g. to reduce memory)
  - Bloom Filters, fingerprinting, etc.

Example (Gigaswitch, 1994)

- $N = 64K$; binary search takes 16 memory accesses
- For each 48-bit address $addr$, we first apply $h(addr)$, to get 48-bit value:
  - 16 LSB are the hash-table entry index (64K entries)
  - Each entry is a balanced binary tree of height at most 3, sorted by the remaining 32 MSB
  - The hash function should guarantee that no more than 8 addresses are in the same tree, and that we can disambiguate between addresses using the 32 MSB
    - Solve corner-cases separately (CAM); rehashing
    - 4 memory accesses

IP longest prefix matching

<table>
<thead>
<tr>
<th>Prefix</th>
<th>Next Hop</th>
<th>Interface</th>
</tr>
</thead>
<tbody>
<tr>
<td>OK 0.0.0.0</td>
<td>10.14.11.33</td>
<td>Output-port 1</td>
</tr>
<tr>
<td>better 12.0.0.0/8</td>
<td>10.14.22.19</td>
<td>Output-port 2</td>
</tr>
<tr>
<td>even better 12.4.0.0/15</td>
<td>10.1.3.77</td>
<td>Output-port 3</td>
</tr>
<tr>
<td>best! 12.5.8.0/23</td>
<td>10.1.3.23</td>
<td>Output-port 4</td>
</tr>
</tbody>
</table>

Longest Prefix Match is Harder than Exact Match

- The destination address of an arriving packet does not carry with it the information to determine the length of the longest matching prefix
- Hence, one needs to search among the space of all prefix lengths; as well as the space of all prefixes of a given length
Current Practical Data

- Caching works poorly in backbone routers
  - 250,000 concurrent flows
- Wire speed lookup needed for 40-byte packets
  - 50% are TCP acks
  - 32 nsec/packet in 10 Gbs and 8 nsec/packet in 40 Gbs
- Lookup dominated by memory accesses → speed is measured by memory accesses
- Prefix length 8-32
- Today 150,000 prefixes → with growth → 1 million prefixes
- Higher speeds need SRAM → Worth minimizing memory

Problem Definition

192.2.2/24, R2
192.2.2/24, R3
192.0.0/22, R2
200.11.0/22, R4
192.2.0.1 192.2.2.100 200.11.0.33

LPM: Find the most specific route, or the longest matching prefix among all the prefixes matching the destination address of an incoming packet

LPM in IPv4

Use 32 exact match algorithms for LPM!

Network Address

Exact match against prefixes of length 1
Exact match against prefixes of length 2
Exact match against prefixes of length 32

Priority Decode and pick

We can start with prefix length 8

Metrics for Lookup Algorithms

- Speed (= number of memory accesses)
- Storage requirements (= amount of memory)
- Low update time
- Scalability
  - With length of prefix: IPv4 unicast (32b), Ethernet (48b), IPv4 multicast (64b), IPv6 unicast (128b)
  - With size of routing table: (sweet spot for today’s designs = 1 million)
- Flexibility in implementation
- Low preprocessing time

Our Toy Example

P1 = 101*
P2 = 111*
P3 = 11001*
P4 = 1*
P5 = 0*
P6 = 1000*
P7 = 100000*
P8 = 100*
P9 = 110*

Packet: 128.0.0.1
  → 100.201
  → P4, P6, P7, P8
  → Forward to P7

Unibit (=Radix) Tries

P1 = 101*
P2 = 111*
P3 = 11001*
P4 = 1*
P5 = 0*
P6 = 1000*
P7 = 100000*
P8 = 100*
P9 = 110*

0 pointer
1 pointer
Unibit Tries

P1 = 101*
P2 = 111*
P3 = 11001*
P4 = 1*
P5 = 0*
P6 = 1000*
P7 = 100000*
P8 = 100*
P9 = 110*

Unibit Tries – Running Example

Input: 1001 Memory: null

Unibit Tries – Running Example

Input: 1001 Memory: P4

Unibit Tries – Running Example

Input: 1001 Memory: P4

Compacting One-Way Branches (variant of PARTICIA tree)

P1 = 101*
P2 = 111*
P3 = 11001*
P4 = 1*
P5 = 0*
P6 = 1000*
P7 = 100000*
P8 = 100*
P9 = 110*

Compacting One-Way Branches (variant of PARTICIA tree)

P1 = 101*
P2 = 111*
P3 = 11001*
P4 = 1*
P5 = 0*
P6 = 1000*
P7 = 100000*
P8 = 100*
P9 = 110*
Unibit Tries - Analysis

- W-bit prefixes, N-prefixes: O(W) lookup, O(NW) storage and O(W) update complexity
- Patricia: O(N) storage (why?)
- Still slow, high memory, but:
  - Simple
  - Extensible to wider fields

Multi-bit Tries

- Binary trie
  - Depth = W
  - Degree = 2
  - Stride = 1 bit

- Multi-ary trie
  - Depth = W/k
  - Degree = 2^k
  - Stride = k bits

Prefix Expansion with Multi-bit Tries

If stride = k bits, prefix lengths that are not a multiple of k need to be expanded.

E.g., k = 2:

<table>
<thead>
<tr>
<th>Prefix</th>
<th>Expanded prefixes</th>
</tr>
</thead>
<tbody>
<tr>
<td>0*</td>
<td>00*, 01*</td>
</tr>
<tr>
<td>1*</td>
<td>11*</td>
</tr>
</tbody>
</table>

Maximum number of expanded prefixes corresponding to one non-expanded prefix = 2^{k-1}

Ternary Content-Addressable Memory (TCAM)

- Replication of next-hop ptr
- Greater number of unused (null) pointers in a node
  - Time ~ W/k
  - Storage ~ NW/k * 2^{k-1}
- Improvement: From Fixed-Stride Tries to Variable – Stride Tries
**Example**

- 0011101101010+00+0100111111111111
- 11=0000011100+010000110101010
- 110+0101010+010011110101000
- 1111010110101101111111111111
- 1111111111111111111

**TCAM Benefits and Disadvantages**

- **Deterministic Search Throughput—O(1) search**
- **Very flexible to other problems as well**
  - Next week: multi-field packet classifications
- **However, relatively costly and energy-consuming**
  - $150 for small (4Mbit) TCAM
  - Energy depends on the number of entries
- **~10 million TCAM devices already deployed**

**Typical Dimensions and Speed**

- 100K-200K rules
- 100-150 symbols per rule
- 133 million searches per second for 144-bit keys
  - Suitable even for 40 Gb/s traffic
- IPv4 and IPv6 lookups are trivial with TCAM
- **Extra symbols** are left in each entry, that can be used to optimize TCAM performance