1. Route lookups

Why high speed lookups?

- IPv4 - $2^{32}$ entries - $4 \times 10^9$
- IPv6 - $2^{128}$ entries - $256 \times 10^{36}$
- Naïve lookup: have a table entry for each IP address
  - IP address: output port
- IPv4 requires 4G entries
  - Memory cost in today's $\$: 60$/G .. 240 $
  - Speed: 50 ns for DRAM, 15 ns for SRAM
  - But
  - Routes are advertised as prefixes
  - Every prefix needs to be unwound
  - Update cost of the table [Gupta 98]

Classless Addressing

<table>
<thead>
<tr>
<th>Class-based</th>
<th>Classless</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>255.255.255.255</td>
</tr>
<tr>
<td>0.0.0.0</td>
<td>191.128.192/18</td>
</tr>
<tr>
<td>0.0.0.0</td>
<td>191.23/16</td>
</tr>
<tr>
<td>0.0.0.0</td>
<td>191.23.14/23</td>
</tr>
<tr>
<td>0.0.0.0</td>
<td>23/8</td>
</tr>
<tr>
<td>0.0.0.0</td>
<td>191/8</td>
</tr>
</tbody>
</table>

Prefixes, speed

- Routing table contains prefixes
  - How many prefixes?
  - Size of table is proportional to prefixes
  - Is it small?
- Prefixes are increasing
- Size of Routing table is increasing
- Lookup algorithms
- Software-based approaches
  - Trie-based algorithms
  - Binary-search on tries, prefixes
- Hardware-based approaches
  - Route-lookup memory
  - Content-addressable memory
Routing Lookups in Hardware [gupta98]

**MAE-EAST routing table (source: www.merit.edu)**

Size of the Routing Table

Source: http://www.telstra.net/ops/bgptable.html

**Longest prefix match**
- With CIDR, route entries are prefixes <prefix, CIDR mask>
- Can be aggregated
- We need to find the longest matching prefix that matches the destination address
- Need to search all prefixes of all length (in order) and among prefixes of the same length

<table>
<thead>
<tr>
<th>Prefix length</th>
<th>Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>64</td>
<td>12000</td>
</tr>
<tr>
<td>65</td>
<td>11000</td>
</tr>
<tr>
<td>66</td>
<td>10000</td>
</tr>
<tr>
<td>67</td>
<td>9000</td>
</tr>
<tr>
<td>68</td>
<td>8000</td>
</tr>
<tr>
<td>69</td>
<td>7000</td>
</tr>
<tr>
<td>70</td>
<td>6000</td>
</tr>
<tr>
<td>71</td>
<td>5000</td>
</tr>
<tr>
<td>72</td>
<td>4000</td>
</tr>
<tr>
<td>73</td>
<td>3000</td>
</tr>
<tr>
<td>74</td>
<td>2000</td>
</tr>
<tr>
<td>75</td>
<td>1000</td>
</tr>
<tr>
<td>76</td>
<td>500</td>
</tr>
<tr>
<td>77</td>
<td>00</td>
</tr>
</tbody>
</table>

**Linear Search**
- Keep N prefixes in a linked list
  - O(N) storage, O(N) lookup time, O(1) update complexity—add at Head of list
  - Arbitrary insertion and deletion O(N)
- Keep N prefixes in a list sorted on prefix length
  - Improve average time for operations
Tree search

Binary tree
- Simple binary tree
  - Each left subtree has key values < root
  - Each right subtree has key values > root
- Full key comparison
- Digital Search tree
- Branch according to selected bits of the key
- Left branch bit value 0, right branch bit value 1
- At each level I, check MSB

Example

Trie

- Same as Digital search tree
- Only leaves store data
- Left to right ordered
- Leaf node has Next Hop information (if prefix found)
- Depth first search; each step compare a bit of the Key
- Fixed length Prefixes
  - P1: 001
  - P2: 100
  - P3: 101
- O(W) lookup
  - W is the length of Prefix (height of the trie)
- Storage O(N) leaves + O(N) for internal nodes
  - N is the number of prefixes

Radix trie

- Store Variable length prefixes (keys)
- Use internal nodes to store prefixes
- A concatenation of all the bits in the path
- Compare bit i at level i
- Look up 1010
- Keep track of prefix seen so far

Trie search

- At each level, search left subtree or right subtree based on the next bit in the address
- On visiting a node with a prefix P, mark BPM=prefix P
- Search ends when there are no more branches; make LPM = BPM
Radix trie

- N prefixes, each W-bits: O(W) lookup, O(NW) storage and O(W) update complexity
- Wastage of storage space in chains
- Idea: Compress branches with one child
- Patricia tree

Patricia tree

- Lookup 10101
  - Longest prefix P4
- Lookup 10111
  - Longest prefix P3
  - Need to backtrack from P4

Pat tree features

- Pat tree is a complete binary tree (node has degree 0 or 2)
- W-bit prefixes: Worst case O(W^2) lookup, O(W) update complexity
- N leaves and N-1 internal nodes
- Less storage
- Backtrack complexity
- Can be improved
**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>11*</td>
<td>11*</td>
</tr>
</tbody>
</table>

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

**Four-ary Trie (k=2)**

A four-ary trie node:
- **next-hop-ptr (if prefix)**
- **ptr00**, **ptr01**, **ptr10**, **ptr11**

**Luleå algorithm: Motivation**

Degermark et al., "Small forwarding tables for fast routing lookups" in Proc. of SIGCOMM ’97

- Large routing tables
  - Patricia (NetBSD), radix (4.4 BSD) trees
  - 24 bytes for leaves
  - Size: 2 Mbytes → 12 Mbytes

- Naïve binary tree is huge, won’t fit in fast CPU cache memory

- Memory accesses are the bottleneck of lookup

**Goal:** minimize memory accesses, size of data structure

- Design for $2^{14} = 16K$ different next-hops
- Method for compressing the radix tree using bit-vectors

**Table of Median Routing Table Sizes**

<table>
<thead>
<tr>
<th>Year</th>
<th>Median routing table size</th>
</tr>
</thead>
<tbody>
<tr>
<td>1997</td>
<td>40,000 entries</td>
</tr>
<tr>
<td>2000</td>
<td>65,000</td>
</tr>
<tr>
<td>2002</td>
<td>100,000</td>
</tr>
<tr>
<td>2008</td>
<td>250,000</td>
</tr>
<tr>
<td>2014</td>
<td>500,000</td>
</tr>
</tbody>
</table>
Luleå algorithm

- CIDR longest prefix match rule: $e_2$ supersedes $e_1$
- Divide a complete binary tree into three levels
  - Level 1: one big node representing entire tree ≤ depth 16 bits
  - Levels 2 and 3: chunks describe portions of the tree
- The binary tree is sparse, and most accesses fall into levels 1 and/or 2

CIDR longest prefix match rule:
- $e_2$ supersedes $e_1$

Divide a complete binary tree into three levels:
- Level 1: one big node representing entire tree ≤ depth 16 bits
- Levels 2 and 3: chunks describe portions of the tree
- The binary tree is sparse, and most accesses fall into levels 1 and/or 2

Luleå algorithm: Level 1

- Covers all prefixes of length ≤ 16
- Cut across tree at depth 16 - bit vector of length $2^{16}$
  - Root head = 1, genuine head = 1, member of genuine head = 0
- Divide bit vector into $2^{12}$ bit masks, each 16 bits long

Luleå: Finding pointer group

- Group pointers by 16-bit bit masks; how many bit masks to skip?
- Recall: Bit vector is $2^{16}$ total length
- Code word array code ($2^{12}$ entries)
  - One entry/16-bit mask, so indexed by top 12 bits of IP address
  - 6-bit offset six: num/ptrs to skip to find 1st ptr for that bit mask in ptr array
  - Four bit masks, max $4 \times 16 = 48$ bits set, $0 \leq$ six $\leq 63$, so value may be too big
- Base index array base ($2^{10}$ entries)
  - One base index per four code words: num/ptrs to skip for those four bit masks
  - Indexed by top 10 bits of IP address

Base index array base ($2^{10}$ entries):
- One base index per four code words: num/ptrs to skip for those four bit masks
- Indexed by top 10 bits of IP address

Base index array base ($2^{10}$ entries):
- One base index per four code words: num/ptrs to skip for those four bit masks
- Indexed by top 10 bits of IP address

Luleå algorithm: Level 1

- Covers all prefixes of length ≤ 16
- Cut across tree at depth 16 - bit vector of length $2^{16}$
  - Root head = 1, genuine head = 1, member of genuine head = 0
- Divide bit vector into $2^{12}$ bit masks, each 16 bits long

Luleå: Finding pointer group
Luleå: Finding pointer group

- Extract top 10 bits from IP address: bix
- Extract top 12 bits from IP address: ix
- Skip code[ix].six + base[bix] pointer groups in the pointer table

Luleå: Finding pointer in pointer group

- $a(n)$ number of possible bit masks of length $2^n$
  - $a(0) = 1$; $a(n) = 1 + a(n-1)^2 \rightarrow a(4) + 1 = 678$
  - So maptable can be indexed with 10 bits
- ten field of code indexes maptable
  - maptable entries are 4-bit offsets
  - maptable structure: pre-computed and constant
  - For each pattern of the bit mask, the values in each cell is fixed
  - ten value varies depending on tree

Luleå: Summary of finding ptr index

- Extract top 10 bits from IP address: bix
- Extract top 12 bits from IP address: ix
- Skip code[ix].six + base[bix] pointer groups in the pointer table

Luleå algorithm: Levels 2 and 3

- Consist of chunks, pointed to by root heads
- Chunk covers subtree of height 8, so 256 heads
- Three types of chunk:
  - Sparse: 1-8 heads, array of 8-bit indices of the heads
  - Dense: 9-64 heads, like Level 1 but only one base index
  - Very dense: 65-256 heads, same format as Level 1
**Luleå: Summary**

- Tradeoff mutability and table construction time for speed
  - Adding a routing entry requires rebuilding entire table
  - Routing tables don’t often change
- Bottom line
  - Lookup: 8 memory references touching 14 bytes
  - Table: 150 Kbytes for 40,000 entries; 4–5 bytes/entry
- Current state of the art in router IP lookup
- Open issue: scaling to IPv6 (128 bit address)

**Hash tables [Waldvogel 98]**

- Store prefixes of different lengths
- Chain prefixes of same length
- Array Size is $O$(distinct string lengths)
- Search: Extract the largest number of bits
- Try match: If match return nexthop else decrease to the next length and repeat

**Binary Search on Trie Levels [waldvogel98]**

- Define recursive search order
  - Search top table
  - If match, search longer prefixes
  - If no match, search shorter prefixes
- Add markers to guide search
  - Markers are longest sub-prefix found in longer prefix length bins
- At most $\log_2(W)$ lookups
- Scales for IPv6

**Route lookup in Hardware**

- Store all prefixes in memory/high speed cache
- IPv4 – 4G entries
- Store 24 bit prefixes (most route entries)
- 16 M – 1998 Prices 50$ - today 1G can be had for $10
- Store 24-bit prefixes with next hop information in memory
- For longer prefixes use secondary table
  - Two-level page table idea
- One memory access time of 50 nsec
**Route Lookups in Hardware**

Prefixes up to 24-bits

- 2^{24} = 16M entries

Prefixes above 24-bits

**Routing Lookups in Hardware**

Prefixes up to 24-bits

- 162.54.34

Content-addressable Memory

- Fully associative memory
- TCAM – ternary CAM (0,1,*
- Exact match operation in a single clock cycle: parallel compare
- Content (Destination address) is the key, address where content is stored is returned
- 1000X more expensive than DRAM
- CAM: Good for fixed length data
  - Variable length prefixes (TCAM)
  - 2-4 MB
  - Power consumption (10 W)

**Routing lookups in H/W**

- Memory is cheap
- Can achieve nsec lookup times
- Can improve technique to fit in SRAM
- Depends on prefix length distribution
- Update complexity
  - Two memory banks (switch after each update)
  - Update every entry
  - Update ranges but tag entry with prefix length
- Need to delete a lot of entries for each prefix delete
  - /16 → 256 entries, /8 → 64K entries

Gupta 98, Infocom
Research

- Technology trends
- Multicore Processors
  - Energy, work schedule, parallelization
  - Slow path on one core, fast path on another core
  - DVFS
- Virtualization
  - Run more than one OS (RTOS, Linux or BSD)
  - RTOS for fast path, Linux for slow path
- Solid state drives
  - Low energy memory
- SDR
  - More or less work for routers?
  - More updates, more communication

Papers

- [NSDI13] Wirespeed name lookup: A GPU based approach
- [NSDI11] SSL shader
  http://www.ndsl.kaist.edu/~kyoungsoo/papers/sslshader.pdf
- [CoNext] Multilayer packet classification using GPU
  http://winlab.rutgers.edu/~feixiong/docs/conext2014.pdf