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

Class-based:

A

B

C

0.0.0.0

255.255.255.255

Classless:

162.23/16

191.23/16

191.128.192/18

191.23.14/23

23/8

191/8

0.0.0.0

255.255.255.255
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
With CIDR, route entries are prefixes \(<\text{prefix}, \text{CIDR mask}>\).

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.
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 $i$

Example

```
A  00001
C  00011
E  00101
R  10010
S  10011
```
Trie

- Same as Digital search tree
- Only leaves store data
- Left to right ordered
- Leaf Node has Next Hop information (if prefix found)
- Dept 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

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>0*</td>
<td>L1</td>
</tr>
<tr>
<td>P2</td>
<td>10*</td>
<td>L2</td>
</tr>
<tr>
<td>P3</td>
<td>101*</td>
<td>L3</td>
</tr>
<tr>
<td>P4</td>
<td>10101</td>
<td>L4</td>
</tr>
</tbody>
</table>

Trie node

- next-hop-ptr (if prefix)
- left-ptr
- right-ptr
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 $BPM = \text{prefix } P$
- 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

Practical Algorithm To Retrieve Information Coded in Alphanumeric
Patricia

- 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
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 = $2^{k-1}$
A four-ary trie node

next-hop-ptr (if prefix)

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>p0</td>
<td>p1</td>
<td>p10</td>
<td>p11</td>
</tr>
<tr>
<td>p00</td>
<td>p01</td>
<td>p10</td>
<td>p11</td>
</tr>
</tbody>
</table>
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 $2^{14} \approx 16K$ different next-hops
  - Method for compressing the radix tree using bit-vectors

<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

IP address space: $2^{32}$ possible addresses

```
- Bit offsets
- Level 1
- Level 2
- Level 3
```

```
- 32
- 24
- 16
- 0
```
Luleå algorithm: Level 1

- Covers all prefixes of length $\leq 16$
- Cut across tree at depth 16 $\Rightarrow$ 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

One bit mask:

```
1 1 1 1 0 0 0 0 1 0 1 1 0 0 0 0 1 1 1 1 1 1
```

Depth 16

Root head

Genuine head
Luleå algorithm: Level 1

- **One bit mask:**
  - 100001011110001111

- **Head information stored in pointer array:**
  - **Pix:**
    - 2
    - 14

- **Next-hop table:**
  - 

- **L2 chunk**

- **Problem:** given an IP address, find the index pix into the pointer array

- **One 16-bit pointer per bit set (=1) in bit-mask**
  - Pointer composed of 2 bits of type info; 14 bits of indexing info
  - **Genuine heads:** index into next-hop table
  - **Root heads:** index into array of Level 2 (L2) chunks
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 \textbf{code} ($2^{12}$ entries)
  - One entry/16-bit bit mask, so indexed by top 12 bits of IP address
  - 6-bit offset \textbf{six}: num/ptrs to skip to find 1\textsuperscript{st} ptr for that bit mask in ptr array
  - Four bit masks, max $4 \times 16 = 48$ bits set, $0 \leq \text{six} \leq 63$, so value may be too big
- Base index array \textbf{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

\begin{verbatim}
\begin{tabular}{|c|c|c|}
\hline
\textbf{code:} & \textbf{ten} & \textbf{six} \\
\hline
$2^{12}$ & & \\
\hline
0 & 0 & \\
3 & 3 & 16 \\
10 & 10 & \\
11 & 11 & \\
0 & 0 & \\
\hline
\end{tabular}
\end{verbatim}

\begin{verbatim}
\begin{tabular}{|c|c|c|}
\hline
\textbf{base:} & \textbf{16} \\
\hline
0 & 0 & \\
13 & 13 & \\
\hline
\end{tabular}
\end{verbatim}

e.g. bit vector:
1000100010000000 1011000100001010 10000000000000000 10000000010000000 10000000010101000...
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

- Pix: $\pm 0$ CODeword
- Code: $\pm 675$
- Maptable: $021154$
Luleå: Summary of finding ptr index

ten = code[ix].ten
six = code[ix].six
pix = base[bix] + six + maptable[ten][bit]
pointer = level1_pointers[pix]
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 $O(\text{distinct string lengths})$
- Search: Extract the largest number of bits
- Try match: If match return nexthop else decrease to the next length and repeat

<table>
<thead>
<tr>
<th>L</th>
<th>H</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>10101</td>
</tr>
<tr>
<td>7</td>
<td>1010111</td>
</tr>
<tr>
<td>12</td>
<td>1010110</td>
</tr>
<tr>
<td>101011010100</td>
<td>101011010100</td>
</tr>
</tbody>
</table>
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
Prefixes up to 24-bits

If longest route with this 24-bit prefix is < 25 bits long:

<table>
<thead>
<tr>
<th>0</th>
<th>Next Hop</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 bit</td>
<td>15 bits</td>
</tr>
</tbody>
</table>

If longest route with this 24 bits prefix is > 24 bits long:

<table>
<thead>
<tr>
<th>1</th>
<th>Index into 2nd table</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 bit</td>
<td>15 bits</td>
</tr>
</tbody>
</table>

2^{24} = 16M entries
Routing Lookups in Hardware

Prefixes up to 24-bits

Prefixes above 24-bits

Next Hop

162.54.34

162.54.34.14

162.54.34

24

14

offset

base

0

Next Hop

1

Pointer

Next Hop

Next Hop
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
**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)
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
[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