Binary Search Schemes for Fast IP Lookups

Pronita Mehrotra
Paul D. Franzon

Department of Electrical and Computer Engineering
North Carolina State University
{pmehrot,paulf}@eos.ncsu.edu

This research is supported by ARDA
Outline

- Background & Motivation
- Description of the Scheme
  - Scheme using all nodes
  - Scheme using only leaves
- Performance
  - Search Time
  - Build Time
  - Memory Consumption
- Conclusions
Background

- Optical Burst Switching
  - Using Just In Time protocol
  - MCNC/NCSU project

- My group
  - Network Processor for OBS ….
Network Processor Architecture

STAGE 1
- Input Message Buffer
- Input Port
- JIT Message Engine

STAGE 2
- Message Parser
- CRCChecker
- Connection State Checker

STAGE 3
- Ack/Nack Generator
- Port Requester
- Output Port Requester

STAGE 4
- Message Reassembly
- Field Update Module

STAGE 5
- STAGE 6
- STAGE 7
- STAGE 8
- ME-In Controller
- ME-Out Controller
- Request Buffer
- Request Arbiter
- Data Bus
- Output Module
- Output Port

STAGE 6
- Register Access Block
Implementation
The bottleneck of the forwarding engine is the route lookup

- **Scalability in speed**
  - Reduce the number of lookups esp. in main memory
    - Trie based schemes good for IPv4 (worst case 32 lookups)
    - Binary schemes potentially better for IPv6

- **Scalability in memory**
  - Reduce the amount of memory required to store data
    - Direct/Indirect lookup schemes use memory inefficiently
    - Trie based schemes have a higher overhead (due to trie completion)
Problem Definition

- Routers store prefixes and not IP addresses
  - To determine the next hop, the *longest matching prefix* needs to be determined

- Example:
  - Destination address = 10110101
  - 10*, 1011*, 101101* all match
  - Longest prefix match is 101101*
  - Next hop address = 2

<table>
<thead>
<tr>
<th>Prefix</th>
<th>Next Hop</th>
</tr>
</thead>
<tbody>
<tr>
<td>10*</td>
<td>3</td>
</tr>
<tr>
<td>1011*</td>
<td>9</td>
</tr>
<tr>
<td>011*</td>
<td>1</td>
</tr>
<tr>
<td>010110*</td>
<td>5</td>
</tr>
<tr>
<td>001*</td>
<td>4</td>
</tr>
<tr>
<td>101101*</td>
<td>2</td>
</tr>
<tr>
<td>011010*</td>
<td>6</td>
</tr>
<tr>
<td>011100*</td>
<td>1</td>
</tr>
<tr>
<td>10111*</td>
<td>8</td>
</tr>
<tr>
<td>00101*</td>
<td>7</td>
</tr>
</tbody>
</table>
Trie Vs. Tree

Binary Trie

Memory Accesses:

▷ Binary Trie: Number of address bits (32 for IPv4)
▷ Binary Tree: $\log_2(N)$ (~16 for 64K entries)

**Trie or Tree?**

- **Issues with Trie Based Schemes:**
  - Extra Nodes with no data add to the depth of the tree
    - More Memory Accesses Needed
  - Search time proportional to the size of the address
    - Binary Trie for IPv4 can take up to 32 cycles
    - For IPv6 the worst case could be 128 cycles.

- **Issues with Tree Based Schemes**
  - Binary Search works for exact matching
    - Backtracking or wrong paths
    - Unbalanced Approaches
  - Pre-processing overhead higher
  - Search time depends on the number of entries

**Goal:** Binary Tree Scheme with faster build and search times
Binary Search – Sort Step

Sorting Prefixes:

Two prefixes:
A = a₁a₂…aₙ  B = b₁b₂…bₘ

- If n = m,
  - Compare by numerical value

- If n ≠ m,
  - Chop longer prefix and compare.
    - If chopped prefixes are equal then, the shorter prefix is considered larger

After Sorting:
000101*, 001*, 010110*, 011010*, 011100*, 011*, 101101*, 10111*, 1011*, 10*

<table>
<thead>
<tr>
<th>Prefix</th>
<th>Next Hop</th>
</tr>
</thead>
<tbody>
<tr>
<td>10*</td>
<td>3</td>
</tr>
<tr>
<td>1011*</td>
<td>9</td>
</tr>
<tr>
<td>011*</td>
<td>1</td>
</tr>
<tr>
<td>010110*</td>
<td>5</td>
</tr>
<tr>
<td>001*</td>
<td>4</td>
</tr>
<tr>
<td>101101*</td>
<td>2</td>
</tr>
<tr>
<td>011010*</td>
<td>6</td>
</tr>
<tr>
<td>011100*</td>
<td>1</td>
</tr>
<tr>
<td>10111*</td>
<td>8</td>
</tr>
<tr>
<td>00101*</td>
<td>7</td>
</tr>
</tbody>
</table>

Sample Prefix Set
Equivalent Binary Tree

- Sorting gives depth-first-search of corresponding binary trie
- Binary Tree constructed as:
  - If $A$ is a prefix of $B$, then $B$ is the child of $A$
  - If $A < B$, then $A$ lies on the left of $B$

Sorted List:
000101*, 001*, 010110*, 011010*, 011100*, 011*, 101101*, 10111*, 1011*, 10*

NOTE: Tree build implicit in sorted list. No need to build explicitly.
Key Element:

- Store parent information with each node

```
00101* 0...010100

001* 0...000100

... ...

101101* 0...101010
```

- This entry has 5 bits
- Parent with 3 bits

```
00101*

010110* 011* 10*

00101* 011010* 011100* 1011*

101101* 10111*
```

- This entry has 6 bits
- Parent with 4 bits
- Parent with 2 bits
Store Information about all parents in “Path Information” and “Next Hop List”

<table>
<thead>
<tr>
<th>Prefix</th>
<th>Next Hop</th>
<th>Path Information</th>
<th>Next Hop List</th>
</tr>
</thead>
<tbody>
<tr>
<td>00101*</td>
<td>7</td>
<td>0…010100</td>
<td>4</td>
</tr>
<tr>
<td>001*</td>
<td>4</td>
<td>0…000100</td>
<td>-</td>
</tr>
<tr>
<td>010110*</td>
<td>5</td>
<td>0…100000</td>
<td>-</td>
</tr>
<tr>
<td>011010*</td>
<td>6</td>
<td>0…100100</td>
<td>8</td>
</tr>
<tr>
<td>011100*</td>
<td>1</td>
<td>0…100100</td>
<td>8</td>
</tr>
<tr>
<td>011*</td>
<td>8</td>
<td>0…000100</td>
<td>-</td>
</tr>
<tr>
<td>101101*</td>
<td>2</td>
<td>0…101010</td>
<td>9,3</td>
</tr>
<tr>
<td>10111*</td>
<td>8</td>
<td>0…011010</td>
<td>9,3</td>
</tr>
<tr>
<td>1011*</td>
<td>9</td>
<td>0…001010</td>
<td>3</td>
</tr>
<tr>
<td>10*</td>
<td>3</td>
<td>0…000010</td>
<td>-</td>
</tr>
</tbody>
</table>
Sample Search

E.g. 100*

1. Search prefix column
   - Fall between
     011*
     101101*

2. Exact match on “≥” entry
   - Search Complete

3. Inexact match
   a. How many bits match?
   b. 1 in this pos^n → complete
   c. 0 in this pos^n → default (root)

2 bits match, 2^{nd} bit (bit 1 = 1)
Correspond to next hop = 3

<table>
<thead>
<tr>
<th>Prefix</th>
<th>Next Hop</th>
<th>Path Information</th>
<th>Next Hop List</th>
</tr>
</thead>
<tbody>
<tr>
<td>00101*</td>
<td>7</td>
<td>0...010100</td>
<td>4</td>
</tr>
<tr>
<td>001*</td>
<td>4</td>
<td>0...000100</td>
<td>-</td>
</tr>
<tr>
<td>010110*</td>
<td>5</td>
<td>0...100000</td>
<td>-</td>
</tr>
<tr>
<td>011010*</td>
<td>6</td>
<td>0...100100</td>
<td>8</td>
</tr>
<tr>
<td>011100*</td>
<td>1</td>
<td>0...100100</td>
<td>8</td>
</tr>
<tr>
<td>011*</td>
<td>8</td>
<td>0...000100</td>
<td>-</td>
</tr>
<tr>
<td>101101*</td>
<td>2</td>
<td>0...101010</td>
<td>9,3</td>
</tr>
<tr>
<td>10111*</td>
<td>8</td>
<td>0...011010</td>
<td>9,3</td>
</tr>
<tr>
<td>1011*</td>
<td>9</td>
<td>0...001010</td>
<td>3</td>
</tr>
<tr>
<td>10*</td>
<td>3</td>
<td>0...000010</td>
<td>-</td>
</tr>
</tbody>
</table>
Search space can be reduced further by using only leaves and eliminating all internal nodes.

For practical routing tables, this leads to a reduction of about 7% entries.

Additional step is required to find the longest matching prefix and the corresponding next hop.

<table>
<thead>
<tr>
<th>Prefix</th>
<th>Next Hop</th>
<th>Path Information</th>
<th>Next Hop List</th>
</tr>
</thead>
<tbody>
<tr>
<td>00101*</td>
<td>7</td>
<td>0…010100</td>
<td>4</td>
</tr>
<tr>
<td>010110*</td>
<td>5</td>
<td>0…100000</td>
<td>-</td>
</tr>
<tr>
<td>011010*</td>
<td>6</td>
<td>0…100100</td>
<td>8</td>
</tr>
<tr>
<td>011100*</td>
<td>1</td>
<td>0…100100</td>
<td>8</td>
</tr>
<tr>
<td>101101*</td>
<td>2</td>
<td>0…101010</td>
<td>9,3</td>
</tr>
<tr>
<td>10111*</td>
<td>8</td>
<td>0…011010</td>
<td>9,3</td>
</tr>
</tbody>
</table>

Compare 100* with both 011100* and 101101*, 101101* gives a better match. Next hop = 3 again.
Build Time

<table>
<thead>
<tr>
<th>Site</th>
<th>Entries</th>
<th>Binary Search (All Nodes)</th>
<th>Binary Search (Only Leaves)</th>
<th>LSV Scheme</th>
</tr>
</thead>
<tbody>
<tr>
<td>MaeEast</td>
<td>23,113</td>
<td>80ms</td>
<td>80ms</td>
<td>210ms</td>
</tr>
<tr>
<td>MaeWest</td>
<td>35,752</td>
<td>130ms</td>
<td>120ms</td>
<td>330ms</td>
</tr>
<tr>
<td>PacBell</td>
<td>27,491</td>
<td>90ms</td>
<td>90ms</td>
<td>260ms</td>
</tr>
<tr>
<td>Paix</td>
<td>17,641</td>
<td>60ms</td>
<td>50ms</td>
<td>150ms</td>
</tr>
<tr>
<td>AADS</td>
<td>31,958</td>
<td>120ms</td>
<td>100ms</td>
<td>300ms</td>
</tr>
</tbody>
</table>

- Build time of binary schemes less than half that of LSV scheme
  - Sorting step accounts for most of the difference (number of entries half that in LSV scheme)
- Build time in binary scheme using only leaves is not very different from that using all nodes
Most core routing tables do not have any internal nodes
  - About 7% of total entries correspond to internal nodes

In about 93% of cases, the next hop list does not need to be looked at
## Search Time

<table>
<thead>
<tr>
<th>Site</th>
<th>Entries</th>
<th>Binary Search (All Nodes)</th>
<th>Binary Search (Only Leaves)</th>
<th>LSV Scheme</th>
</tr>
</thead>
<tbody>
<tr>
<td>MaeEast</td>
<td>23,113</td>
<td>662ns</td>
<td>610ns</td>
<td>761ns</td>
</tr>
<tr>
<td>MaeWest</td>
<td>35,752</td>
<td>742ns</td>
<td>652ns</td>
<td>845ns</td>
</tr>
<tr>
<td>PacBell</td>
<td>27,491</td>
<td>703ns</td>
<td>656ns</td>
<td>761ns</td>
</tr>
<tr>
<td>Paix</td>
<td>17,641</td>
<td>640ns</td>
<td>634ns</td>
<td>739ns</td>
</tr>
<tr>
<td>AADS</td>
<td>31,958</td>
<td>700ns</td>
<td>640ns</td>
<td>777ns</td>
</tr>
</tbody>
</table>

- For the binary scheme using all nodes, > 10% improvement in average search speed over LSV scheme
  - 1-2 fewer memory accesses required
- For binary scheme using only leaves, 15-20% improvement in speed obtained
Memory Requirement

<table>
<thead>
<tr>
<th>Site</th>
<th>Entries</th>
<th>Binary Search (All Nodes)</th>
<th>Binary Search (Only Leaves)</th>
<th>LSV Scheme</th>
</tr>
</thead>
<tbody>
<tr>
<td>MaeEast</td>
<td>23,113</td>
<td>0.62MB</td>
<td>0.58MB</td>
<td>1.06MB</td>
</tr>
<tr>
<td>MaeWest</td>
<td>35,752</td>
<td>0.96MB</td>
<td>0.9MB</td>
<td>1.64MB</td>
</tr>
<tr>
<td>PacBell</td>
<td>27,491</td>
<td>0.74MB</td>
<td>0.7MB</td>
<td>1.26MB</td>
</tr>
<tr>
<td>Paix</td>
<td>17,641</td>
<td>0.48MB</td>
<td>0.45MB</td>
<td>0.81MB</td>
</tr>
<tr>
<td>AADS</td>
<td>31,958</td>
<td>0.86MB</td>
<td>0.81MB</td>
<td>1.46MB</td>
</tr>
</tbody>
</table>

- Memory consumption for both binary schemes, about half that in LSV scheme
  - Each prefix in LSV scheme gives rise to two entries in the routing table
- Memory consumption for binary scheme using only leaves not very different from that using all nodes
Most IPv4 prefixes exist between 16 and 24 bits
  - The path information field, in this case needs to be only 23 bits wide

For IPv6, in a given domain, much fewer than 128 bits would most likely be required for the path information field
Updates

- Port reassignments and new internal nodes
  - One entry needs to be rewritten

- New leaf nodes
  - Substantial update
  - Consider sparing of table to prevent dead time
Conclusions

- Binary Search Schemes are possibly better suited for large addresses like IPv6
  - Number of lookups depend on the number of entries and not the address size
- Binary Search schemes for larger address size
  - 1-2 fewer memory accesses as compared to LSV scheme
    - Up to 20% faster than LSV
  - Memory requirement ~2x lesser than the LSV scheme
  - Build time is less than half that of LSV scheme
    - Sorting step accounts for most of the difference
  - Update Process at O(N)
- Mainly intended for software implementation
  - Trie based hardware scheme presented at HPSR’02