CS419: Computer Networks

Lecture 6: March 7, 2005

*Fast Address Lookup:*
Forwarding/Routing Revisited
Best-match Longest-prefix forwarding table lookup

- We looked at the “semantics” of best-match longest-prefix address lookup
  - As a linear walk through the list of FIB entries, in order of longest-to-shortest prefix
- But we didn’t look at how to do this fast!
Tree Bitmap

- This is a fast address lookup algorithm from George Varghese (UCSD)
- Used in high-speed routers (Cisco)
  - George has a startup doing this
- This lecture based on this paper:
Main goals:

- Wire-speed forwarding at OC-192 (10 Gbps)
- 24 million packets per second!!!
  - For small packets (TCP acks)
- Minimize memory accesses (4-7 for 41K FIB entries!)
- Performance guarantees
  - Not just for lookup, but for constructing the tree as well
Other goals

- Operate in software and hardware modes
  - Variations on hardware: single-chip, off-chip memory, CAMs
- Minimize memory size
- Take advantage of memory characteristics (i.e. cache line associated with a read)
- Tunable across many architectures
Tuning to different memories

- All these algorithms involve traversing some kind of tree structure.
- The trick to tuning is deciding where in physical memory to stick different parts of the tree...
Tuning to different memories
Tuning to different memories
Example: Multiple memory banks

- Put top of tree in Bank A, bottom tree in Bank B, run two lookups in parallel
Example: Size of memory “burst”

<table>
<thead>
<tr>
<th>Memory Technology with Data path Width</th>
<th>ASIC pins</th>
<th>Data Rate (Mbps)</th>
<th>Logical # of Banks</th>
<th># of Random Memory Accesses every 160 ns</th>
<th>ASIC Pins/ Random Memory Access</th>
<th>Block Size (bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>PC-100 SDRAM (64-bit)</td>
<td>80</td>
<td>100</td>
<td>2</td>
<td>4</td>
<td>20</td>
<td>32</td>
</tr>
<tr>
<td>DDR-SDRAM (64-bit)</td>
<td>80</td>
<td>200</td>
<td>4</td>
<td>4</td>
<td>20</td>
<td>64</td>
</tr>
<tr>
<td>Direct Rambus (16-bit)</td>
<td>76</td>
<td>800</td>
<td>8a</td>
<td>16</td>
<td>4.75</td>
<td>16</td>
</tr>
<tr>
<td>Synchronous SRAM (32-bit)</td>
<td>52</td>
<td>100</td>
<td>1</td>
<td>16</td>
<td>3.25</td>
<td>4</td>
</tr>
</tbody>
</table>

Various memory parameters determines the number of bytes that be read in one memory access. This in turn determines how to structure the lookup tree.
Some types of trees

- Next we’ll look at a number of tree structures, each more advanced (and harder to understand!) than the last:
  - Unibit tries
  - Expanded tries
  - Lulea (bitmap)
  - Tree bitmap
Unibit tries

Prefix Database

<table>
<thead>
<tr>
<th>Prefix</th>
<th>Bit Pattern</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>*</td>
</tr>
<tr>
<td>P2</td>
<td>1*</td>
</tr>
<tr>
<td>P3</td>
<td>00*</td>
</tr>
<tr>
<td>P4</td>
<td>101*</td>
</tr>
<tr>
<td>P5</td>
<td>111*</td>
</tr>
<tr>
<td>P6</td>
<td>1000*</td>
</tr>
<tr>
<td>P7</td>
<td>11101*</td>
</tr>
<tr>
<td>P8</td>
<td>111001*</td>
</tr>
<tr>
<td>P9</td>
<td>1000011*</td>
</tr>
</tbody>
</table>

Unibit Trie

Legend:
- Prefix Node
- Place Holder Node
- next bit = 0
- next bit = 1
Unibit tries

- Traverse the tree one bit at a time
- If terminate at a prefix node, use that as the next hop
- If terminate at a “place holder” (non-prefix) node, use most recently traversed prefix node as the next hop
- One-way branches can be compressed out
Unibit tries

- Small memory and update times
- Main problem is the number of memory accesses required
  - 32 in the worst case
  - Way beyond our budget of approx 4
    - (OC48 requires 160ns lookup, or 4 accesses)
Expanded tries

To speed up lookup, branch on multiple bits at each decision instead of just one
- The number of bits used is the “stride length”

Otherwise, lookup algorithm similar to unibit
- i.e. remember most recently traversed prefix in case of non-prefix termination
Prefix expansion without leaf pushing
Expansion

- Prefixes that don’t fall on stride boundaries must be “expanded” to fill all slots
- Eg P6 expanded to four slots
- Or, P2 expanded initially to four slots, but then P4 and P5 take precedence over P2
Expanded trie inefficiencies

- Expansion uses up more space
- Also, each entry requires two fields
  - A pointer to the next node in the tree
  - A prefix
- This is because some entries require both a pointer and a prefix
  - i.e. P2, P5, and P6
- Update speed versus memory size tradeoff
We can combine pointer and prefix…(leaf pushing)
Some observations

- Leaf pushing increases update time
  - Prefix can appear in many nodes (i.e. P5)
- Because of memory “burst” reads, the entire node can be read with one memory access
  - Try to make node size match burst size
Lulea uses a bitmap to compress out repeated entries.
Lulea bitmap
Lulea bitmap processing

- Doesn’t this just increase processing?
  - Have to shift through the bitmap…
- Yes, but memory access is by far the bottleneck
  - Hardware easily process the bitmaps
  - Even software can execute many instructions in one memory access
Lulea trie performance

- Very compact storage
- Very fast lookup
- But, updating the Lulea trie can be very expensive
- For instance, adding a short prefix can result in a lot of leaf pushing...many entries must be modified
Tree Bitmap: first insight

- Avoid the problems of expansion and leaf pushing by going back (conceptually) to the basic Unibit tree
- BUT: Avoid the problem of many pointers by storing child nodes in contiguous memory areas as an array
  - Instead of many pointers, calculate offset into child array
Tree Bitmap with three-bit strides
Tree Bitmap: second insight

- To compress, use two bitmaps instead of just one
  - Internal prefix bitmap
  - External pointers bitmap
- This avoids leaf pushing
  - (which is what gives Lulea potential large update times)
Tree Bitmap’s two bitmaps

Extended Paths Bitmap

Internal Tree Bitmap
Compact “nodes”

- Each child node contains only:
  - Internal Tree Bitmap
  - Extended Paths Bitmap
  - One pointer to child array

- But what about the next hop info for stored prefixes???
  - This is what was pushed to the leaves in Lulea…
Stored next hop info for prefixes

- Store prefixes in a separate array *adjacent in memory* to the node
- Internal tree bitmap tells us where in that array to find the pointer
- Furthermore, don’t actually retrieve the next hop info until the very end of the search
  - Adds one extra memory access at the very end
Next hop pointer array in adjacent memory location
Lookup algorithm (basic idea anyway)

- Conceptually, the two bitmaps allow you to “reconstruct” the Unibit tree for a given stride (i.e. 3 bits)
- The child pointer plus Extended Paths Bitmap tell you where to find the child node
- The Internal Tree Bitmap tells you which Unibit tree nodes have prefixes
Lookup algorithm (basic idea anyway)

Prefix Database

<table>
<thead>
<tr>
<th>Prefix</th>
<th>Database</th>
<th>Child Pointer</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>*</td>
<td></td>
</tr>
<tr>
<td>P2</td>
<td>1*</td>
<td></td>
</tr>
<tr>
<td>P3</td>
<td>00*</td>
<td></td>
</tr>
<tr>
<td>P4</td>
<td>101*</td>
<td></td>
</tr>
<tr>
<td>P5</td>
<td>111*</td>
<td></td>
</tr>
<tr>
<td>P6</td>
<td>1000*</td>
<td></td>
</tr>
<tr>
<td>P7</td>
<td>11101*</td>
<td></td>
</tr>
<tr>
<td>P8</td>
<td>111001*</td>
<td></td>
</tr>
<tr>
<td>P9</td>
<td>1000011*</td>
<td></td>
</tr>
</tbody>
</table>

Root Node

Child Array

P1

P2

P3

P4

P5

P6

P7

P8

P9

Child Pointer

1.01.1000

00001101