Sources

- George Varghese, “Network Algorithmics”, Morgan Kauffmann, December 2004

IP forwarding

• Input: a source IP address

• Output: an output port

• Defines the link on which the packet must be sent

IP dst: 129.82.103.121

Port 1
Port 2
Port 3
... 
Port 16
IP forwarding - II

- Internally, router determines output port by looking up a routing table.

Routing table:
- 49.* Port 3
- 129.82.* Port 2
- 151.1.1.* Port 12

IP.dst: 129.82.103.121
Longest prefix matching

- There are $4B+$ possible 32-bit IP addresses
  - Not really (due to reserved/private IP address spaces), but still...
- There are $2^{128}$ possible 128-bit IPv6 addresses
- No data structure can possibly encode individual (destination IP, port) for each possible destination
Longest prefix matching - II

• What do we do then? Longest prefix matching

• Observation: IP addresses are not randomly assigned
  • Organizations receive blocks of IP address space
  • Each block defined by IP+mask: e.g. 129.82.103.0/24
Longest prefix matching - III

• Organizations receive blocks of IP address space
  
  • Each block defined by IP+mask: e.g. 129.82.103.0/24
    
    • Machines in this block will be assigned addresses with the 129.82.103 prefix, e.g., 129.82.103.1, 129.82.103.24, 129.82.103.11, etc....

• Implication: IP addresses which share the same route also tend to share the same prefix
Longest prefix matching - IV

- Route aggregation example
Longest prefix matching - V

• Why “longest”?
  • Given a destination IP address, there may be multiple entries in the routing table with matching prefixes of different length
  • In this case, the most specific (longer) prefix is chosen
• Example:

IP.dst: 129.82.103.121
Longest prefix matching - VI

- Longest prefix matching - general description:
  - Identify all rules whose prefix matches the destination IP address from the current packet
  - Among all the results, pick the rule with the longest prefix

Routing table

<table>
<thead>
<tr>
<th>IP Address</th>
<th>Port</th>
</tr>
</thead>
<tbody>
<tr>
<td>129.82.103.*</td>
<td>Port 1</td>
</tr>
<tr>
<td>129.82.103.121</td>
<td>Port 6</td>
</tr>
</tbody>
</table>
Longest prefix matching - VII

- Problems with longest prefix matching:
  - Routing tables may contain hundreds of thousands of rules
  - Need to find all rules that match a certain IP, pick the one with the longest prefix
  - Need to do it fast! (1 Tbps w/ 512B packets: 250M+ packets/s
  - Impossible to use linear search
  - Routing “tables” are not really tables
    - Need optimized data structure!
Efficient longest prefix matching

• Router designers have looked at a number of approaches to efficient longest prefix matching:
  
  • Hardware-based approaches (specialized memory)
    
    • TCAMs
  
  • Software-based approaches (SRAM + optimized data structures)
    
    • Tries
    • Lulea Tries
    • Tree bitmaps
    • …
TCAMs

- **TCAM**: Ternary Content-Access Memory

- **Content-Access Memory**: a hardware memory which is indexed by keys instead of memory addresses
  - Think of it as an hardware key-value store
  - Given a key, returns the corresponding value

- **Ternary**: not all key bits need to be specified in each entry
  - Some bits can be set to “don't care”
  - Those parts of the key always match
TCAM - example

- Keys: 3-bit wide
- Values: 4-bit wide

<table>
<thead>
<tr>
<th>Key</th>
<th>Keys</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>*</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

**Example:**

- For the key `101`, the corresponding values are `000000`.
How is this useful?

• Route prefixes can be directly stored as keys
  • The part of the prefix which is not covered by the mask is set as “don’t care”

• Output ports stored as values

• Given a search key, among all active entries pick the one with the least “don't care” bits
## TCAM+LPM - Example

<table>
<thead>
<tr>
<th>Key</th>
<th>Routing table</th>
</tr>
</thead>
<tbody>
<tr>
<td>129</td>
<td>100000010101001001100110</td>
</tr>
<tr>
<td>82</td>
<td>101010011</td>
</tr>
</tbody>
</table>

The example shows how TCAM+LPM works in routing tables, highlighting the matching process and mask usage.
TCAM - pros and cons

• **Advantages:**
  
  • Fast! Can search or update in 1 memory cycle (~10ns)
  
  • Can handle large routing tables (~100000 prefixes)

• **Disadvantages:**
  
  • Area-hungry (12 transistors per bit - compare to 6 transistors per bit required by regular SRAM)
  
  • Power-hungry (because of the parallel compare between keys and all entries)
  
  • Arbitration delay (~5 ns to pick winning entry)
  
  • Require dedicated chip

• **Can we have an alternative approach not requiring specialized memory?**
LPM - memory-based approaches

- Easiest: use generic data structure
  - E.g. potentially slow, unpredictable latency
  - Most conventional structures are designed for exact matching
- Next: use generic data structure + cache
  - Problem: IP lookups have poor memory locality
  - Why? Core routers process tens of thousands of network flows at the same time
  - Different flows have different destination IPs, which cause different memory locations to be accessed
LPM - poor memory locality

### Routing Table

<table>
<thead>
<tr>
<th>Keys</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>IP.dst: 129.82.103.121</td>
<td></td>
</tr>
<tr>
<td>IP.dst: 144.92.9.21</td>
<td></td>
</tr>
<tr>
<td>IP.dst: 130.192.181.193</td>
<td></td>
</tr>
<tr>
<td>IP.dst: 131.234.142.33</td>
<td></td>
</tr>
<tr>
<td>IP.dst: 144.92.9.21</td>
<td></td>
</tr>
<tr>
<td>IP.dst: 130.192.181.193</td>
<td></td>
</tr>
<tr>
<td>IP.dst: 129.82.103.121</td>
<td></td>
</tr>
<tr>
<td>IP.dst: 129.82.103.121</td>
<td></td>
</tr>
<tr>
<td>IP.dst: 131.234.142.33</td>
<td></td>
</tr>
</tbody>
</table>

...
Latency

- Latency is also a problem!
  - DRAM latency: ~30ns; SRAM latency ~3ns
  - For large routing tables, SRAM may be too expensive - so need to use DRAM
  - Slow, and caching does not help!
  - Can coalesce multiple accesses to alleviate issue
Take away point

• Generic data structures/memory hierarchy have poor performance on IP lookup operations (either too slow, or too large)

• Need specialized data structures for efficiency

• We are going to see a few:
  • Tries
  • Tree bitmaps
Lookup tries

- Trie: tree data structure used to store values associated with a **set of strings**

- Each node represents a prefix of one or more strings in the set

- Allows compact encoding, particularly when many strings share compact prefixes
Lookup trie - example

Example: need to store the following keys and values:

- CSU: 3
- Colorado: 15
- Colostate: 11
IP lookup tries

- IP addresses are string of bits
- Many of them share common prefixes
- Tries are ideally suited for the task!
- Also helps with longest prefix matching
  - Lower-priority rules closer to root; higher priority rules close to leaves
Unibit tries

- Simplest possible trie-based approach
- Each node “consumes” 1 bit of the destination IP address
- Each node which stores the last bit of a prefix in the routing table also stores a pointer to the corresponding port number
In Figure 11.7, this is done by using a text string (i.e. “01”) to represent the pointers that would have been followed in the one-way branch. Thus in Figure 11.7, two trie nodes (containing two pointers apiece) in the path to P3 have been replaced by a single text string of 2 bits. Clearly, no information has been lost by this transformation. (As an exercise, determine if there is another path in the trie that can similarly be compressed.)

To search for the longest matching prefix of a destination address \( D \), the bits of \( D \) are used to trace a path through the trie. The path starts with the root and continues until search fails by ending at an empty pointer or at a text string that does not completely match. While following the path, the algorithm keeps track of the last prefix encountered at a node in the path. When search fails, this is the longest matching prefix that is returned.

For example, if \( D \) begins with 1110, the algorithm starts by following the 1-pointer at the root to arrive at the node containing P4. The algorithm remembers P4 and uses the next bit of \( D \) (a 1) to follow the 1-pointer to the next node. At this node, the algorithm follows the 1-pointer to arrive at P2. When the algorithm arrives at P2, it overwrites the previously stored value (P4) by the newer prefix found (P2). At this point, search terminates, because P2 has no outgoing pointers.

On the other hand, consider doing a search for a destination \( D' \) whose first 5 bits are 11000. Once again, the first 1 bit is used to reach the node containing P4. P4 is remembered as the last prefix encountered, and the 1 pointer is followed to reach the rightmost node at height 2. The algorithm now follows the third bit in \( D' \) (a 0) to the text string node containing “01.” Thus we remember P9 as the last prefix encountered. The fourth bit of \( D' \) is a 0, which matches...
Unibit tries - example II

- Need to lookup 10000011

- (Using 8-bit addresses for simplicity)

P7 wins over P8 (longest prefix)
Considerations on unibit tries

• Convenient for understanding…

• …not so useful in practice

• **Slow!** Up to 32 memory accesses per lookup

• **Idea:** improve efficiency by looking up *multiple bits in each trie node*
Multibit tries

- 1 bit per node is too slow!

- What about $B > 1$ bits per node? \textbf{(Multibit tries)}
  - #memory accesses cut to $32/B$ (B is called \texttt{stride})

- Advantage: faster (less latency)

- Disadvantage: what do we do with prefixes whose length is not a multiple of N?
  - E.g. prefix 10101*** but stride $B = 3$
Multibit tries - II

• Problem: how can we support every prefix length with multibit tries?

  • Use a technique called **prefix expansion**

  • For each stride, generate all possible prefixes

    • Simple, but…

    • Generates entries even for prefixes that are not in the routing table
Prefix expansion example

Routing Table

<table>
<thead>
<tr>
<th>Prefix</th>
<th>Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>101*</td>
<td>P1</td>
</tr>
<tr>
<td>111*</td>
<td>P2</td>
</tr>
<tr>
<td>11001*</td>
<td>P3</td>
</tr>
<tr>
<td>1*</td>
<td>P4</td>
</tr>
<tr>
<td>0*</td>
<td>P5</td>
</tr>
<tr>
<td>1000*</td>
<td>P6</td>
</tr>
<tr>
<td>100000*</td>
<td>P7</td>
</tr>
<tr>
<td>100*</td>
<td>P8</td>
</tr>
<tr>
<td>110*</td>
<td>P9</td>
</tr>
</tbody>
</table>

FIGURE 11.9 Expanded trie (which has two strides of 3 bits each) corresponding to the prefix database of Figure 11.8.

Thus each trie node element is a record containing two entries: a stored prefix and a pointer.

Trie search proceeds 3 bits at a time. Each time a pointer is followed, the algorithm remembers the stored prefix (if any). When search terminates at an empty pointer, the last stored prefix in the path is returned.

For example, if address D begins with 1110, search for D starts at the 111 entry at the root node, which has no outgoing pointer but a stored prefix (P2). Thus search for D terminates with P2. A search for an address that starts with 100000 follows the 100 pointer in the root (and remembers P8). This leads to the node on the lower right, where the 000 entry has no outgoing pointer but a stored prefix (P7). The search terminates with result P7. Both the pointer and stored prefix can be retrieved in one memory access using wide memories (P5b).

A special case of fixed-stride tries, described in Gupta et al. [GLM98], uses fixed strides of 24, 4, and 4. The authors observe that DRAMs with more than 2²⁴ locations are becoming available, making even 24-bit strides feasible.

11.5.2 Variable-Stride Tries

In Figure 11.9, the leftmost leaf node needs to store the expansions of P3 = 11001*, while the rightmost leaf node needs to store P6 (1000*) and P7 (100000*). Thus, because of P7, the rightmost leaf node needs to examine 3 bits. However, there is no reason for the leftmost leaf node to examine more than 2 bits because P3 contains only 5 bits, and the root stride is 3 bits. There is an extra degree of freedom that can be optimized (P13).

In a variable-stride trie, the number of bits examined by each trie node can vary, even for nodes at the same level. To do so, the stride of a trie node is encoded with the pointer to the node. Figure 11.9 can be transformed into a variable-stride trie (Figure 11.10) by replacing the leftmost node with a four-element array and encoding length 2 with the pointer to the leftmost node.
Can we reduce storage?

- Nodes do not need to be all of the same size
- For example, in previous example leftmost node only stores P3 which is a 2-bit prefix
- Optimization: variable-stride nodes
Optimal node size

• Optimal node size in a variable-stride trie is a tricky problem

• Srinivasan and Varghese: dynamic programming algorithm

• Complexity $O(N \times W^2 \times h)$ where $N$: #prefixes, $W$: address width, $h$: desired maximum height of tree (= maximum number of memory accesses)

• In practice, optimization runs in ~1 s and leads to ~5x size reduction (compared to fixed-stride trie) on 40k-prefix routing table
Compressed tries

• Variable-stride tries offer a compromise: the higher the tree, the more fine-tuned node size is (i.e. less wasted space in each node)

• However, the higher the tree, the higher the latency (higher tree = more memory accesses)

• Can we improve this trade off (i.e. short tree, but no wasted space?)

• Idea: use compression techniques to reduce node size
Compressed tries: Lulea

• Lulea algorithm (Degermark et al., 1997): due to prefix expansion, the same rule is oftentimes repeated multiple times in a node

• Example:

• Can compress using bitmap+
  compressed sequence

• Bitmap: 1 represents new rule,
  0 same rule

• Example becomes 1100 and P7,P6
Why does this work?

Rules are large (even if the trie just stores output port per each prefix, I will still need 16 bits per rule)

Bitmaps are small (1 bit per rule)

Instead of repeating same rule multiple times, the rule is just specified once - the bitmap takes care of the rest!
• Problem: trie nodes may need to store both rules and pointers to other nodes

• Complicates compression

• Solution: leaf pushing

• Every time a node would contain both a rule and a pointer, the rules are pushed to the children node
11.5.2 Variable-Stride Tries

Trie search proceeds 3 bits at a time. Each time a pointer is followed, the algorithm remembers P8. This leads to the node on the lower right, where the 000 entry has no outgoing pointer but a stored prefix (P7). The search terminates with result P7. Both the pointer and the stored prefix (if any) are available, making even 24-bit strides feasible.

Figure 11.9 can be transformed into a variable-stride trie (Figure 11.10) by replacing the locations are becoming two entries: a stored prefix and a pointer.

A special case of fixed-stride tries, described in Gupta et al. [GLM98], uses fixed strides of 24, 4, and 4. The authors observe that DRAMs with more than 2 leftmost node with a four-element array and encoding length 2 with the pointer to the leftmost node to examine more than 2 bits because P3 contains only 5 bits, and the root stride is 3 bits. However, there is no reason for the leftmost leaf node to store P6 (1000*) and P7 (100000*). Thus, because of P7, the rightmost leaf node needs to examine 3 bits.
Lulea - V

- Lulea compression with leaf pushing:

Original uncompressed node:

<table>
<thead>
<tr>
<th>000</th>
<th>001</th>
<th>010</th>
<th>011</th>
<th>100</th>
<th>101</th>
<th>110</th>
<th>111</th>
</tr>
</thead>
<tbody>
<tr>
<td>P5</td>
<td>P5</td>
<td>P5</td>
<td>P5</td>
<td>ptr1</td>
<td>P1</td>
<td>ptr2</td>
<td>P2</td>
</tr>
</tbody>
</table>

Bitmap:

- 1
- 0
- 0
- 0
- 1
- 1
- 1
- 1

Rules:

- P5
- P1
- P2

Savings (assuming 16-bit rules):

- Original: 2*8 = 16B
- Compressed: 1+5*2 = 11B
- Reduced node size by ~30%
Tree bitmaps

• What’s the problem with Lulea?
  • Small, fast, but big headache in regards to updates to the data structure
  • Due to leaf pushing, changes to the routing table may involve updates to large numbers of nodes, especially for short prefixes
  • Can we have a space- and latency-efficient data structure without leaf pushing?
• Tree bitmap
Tree bitmaps - II

• **Problem with Lulea**: must deal with **pointers** (to other nodes) and **rules**, but only defines 1 bitmap per node.

• **Tree bitmap core idea**: use **two bitmaps per node**; one bitmap compresses the **rule list**, the other one the **node pointer list**.

• **No leaf pushing** -> must keep track of prefix length in node.
  
  • Bitmap stores 1 bit for each possible prefix of length up to the current one (compare to Lulea, whose bitmaps stores $2^B$ bits, one for each possible value of the current stride).
Tree bitmaps example

### Input rules

<table>
<thead>
<tr>
<th>Prefix</th>
<th>Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>101*</td>
<td>P1</td>
</tr>
<tr>
<td>111*</td>
<td>P2</td>
</tr>
<tr>
<td>11001*</td>
<td>P3</td>
</tr>
<tr>
<td>1*</td>
<td>P4</td>
</tr>
<tr>
<td>0*</td>
<td>P5</td>
</tr>
<tr>
<td>1000*</td>
<td>P6</td>
</tr>
<tr>
<td>100000*</td>
<td>P7</td>
</tr>
<tr>
<td>100*</td>
<td>P8</td>
</tr>
<tr>
<td>110*</td>
<td>P9</td>
</tr>
</tbody>
</table>

### Trie node

#### Rules

<table>
<thead>
<tr>
<th>Prefix</th>
<th>Bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>*</td>
<td>0</td>
</tr>
<tr>
<td>0*</td>
<td>1</td>
</tr>
<tr>
<td>1*</td>
<td>1</td>
</tr>
<tr>
<td>00*</td>
<td>0</td>
</tr>
<tr>
<td>01*</td>
<td>0</td>
</tr>
<tr>
<td>10*</td>
<td>0</td>
</tr>
<tr>
<td>11*</td>
<td>0</td>
</tr>
<tr>
<td>000*</td>
<td>0</td>
</tr>
<tr>
<td>001*</td>
<td>0</td>
</tr>
<tr>
<td>010*</td>
<td>0</td>
</tr>
<tr>
<td>011*</td>
<td>0</td>
</tr>
<tr>
<td>100*</td>
<td>1</td>
</tr>
<tr>
<td>101*</td>
<td>1</td>
</tr>
<tr>
<td>110*</td>
<td>1</td>
</tr>
<tr>
<td>111*</td>
<td>1</td>
</tr>
</tbody>
</table>

#### Pointer array

<table>
<thead>
<tr>
<th>Prefix</th>
<th>Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>P5</td>
<td>P4</td>
</tr>
<tr>
<td>P8</td>
<td>P1</td>
</tr>
<tr>
<td>P9</td>
<td>P2</td>
</tr>
</tbody>
</table>

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>0</td>
<td>ptr1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>001</td>
<td>0</td>
<td>ptr2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>010</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>011</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>100</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>101</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>110</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>111</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Notes on implementation

- Each lookup is a trie traversal

- **Pipelined for efficiency:** while traversal of level N for the current lookup can be done in parallel to traversal of level N-1 for the next lookup

- Dedicated memory/node lookup ASIC for each trie level
PLUG

PLUG: Flexible Lookup Modules for Rapid Deployment of New Protocols in High-Speed Routers (L. De Carli et al., SIGCOMM 2009)
Background

• Context: packet processing in high-speed routers

• Requirements: real-time processing, high throughput (40Gbps -> 100Gbps -> 1 Tbps)
Background - II

• What’s the problem? We know how to build an efficient IP forwarding engine! Well…

• …IP lookup is not the only operation that routers need to perform
  • Ethernet lookup
  • Packet classification
  • …

• Challenge: implementing all these operations efficiently within the same hardware
Most operations performed by routers require time-consuming lookups in various specialized data structures.

IP lookup using tries, Ethernet forwarding using hash tables, etc...

Typically, implemented using ASICs:
- Complicate to design
- Expensive
- Inflexible

Can they be replaced by a single module?
Background - IV

- Idea: exploit commonalities between accelerators

- Orderly combination of steps
- Each step includes dedicated computation & memory
- Predictable communication patterns
PLUG (Pipelined LookUp Grid)

- Efficient execution of algorithmic steps with local data+computation
- Algorithms (IP lookup etc.) as **dataflow graphs** (DFGs)
- Massively parallel HW efficiently executes DFG applications
Why dataflow graphs?

• A dataflow graph implements a program representation based on data dependencies

• Each node in the graph represents a small sequence of instructions; if the output of the node is used by one or more other nodes, the graph will have edges going from the current node to the ones that consume its output

• Very well suited for expressing certain types of data structure lookups
Lookups as DFGs

Ethernet lookup:
hash table w/ 4 entries per bucket

IP forwarding:
multi-level lookup trie
PLUG hardware

- Massively parallel HW architecture
- Matrix of elements (tiles), each including memory and computation, connected by on-chip network

Tile

Point-to-point links

- DFG nodes → tiles; edges → network links
Case study
IPv4 forwarding

- Three-level multibit trie w/ 16/8/8 stride
- 3 pages per level: 2 for rule array, 1 for children ptrs

Throughput 1G lookup/s
Latency 90 ns
Power 0.35 W
Some results

- Applications:
  - IPv4/IPv6: PLUG vs NetLogic NL9000
  - Regexp matching: PLUG vs Tarari T10
Other

- Ethernet forwarding (hash table)
- Packet classification (decision trees) (Vaish et al., ANCS 2011)
- Ethane (precursor to OpenFlow; hash tables)
- Seattle (DHT-based forwarding; hash tables + trees)
- Alternative designs: LEAP (Harris et al., ANCS 2012); SWSL (Kim et al., ANCS 2013)
That’s all for today

• Next lecture:
  • We are moving up to layer 4 (TCP congestion avoidance and control)
  • Our TA will also introduce the first class project