Scalable High Speed IP Routing Lookups

Marcel Waldvogel, George Varghese and Jon Turner

Washington University
Computer Science Department

http://www.arl.wustl.edu/~jst
Motivation

- Rapid growth of internet increasing demands for higher performance routing.
- Address lookup is one key component of packet forwarding in routers.
  » given IP address, determine which output link is best choice for reaching that address
  » hierarchical address structure of IP addresses means that addresses that can be reached by similar routes often start with the same sequence of bits
  » this allows routing table compression by combining entries for groups of addresses that have common prefixes
  » turns lookup problem into best-matching prefix
- Standard lookup algorithms require several $\mu$s per lookup.
- New algorithms can do lookup in 100-200 ns.
The Address Lookup Problem

- Address in packet is compared to stored prefixes, starting with leftmost bit.
- Prefix that matches largest number of address bits is desired match.
- Packet is forwarded to the specified next hop.
- Next hop fields change as a result of topology changes and traffic changes.
- Set of prefixes changes infrequently.

<table>
<thead>
<tr>
<th>prefix</th>
<th>next hop</th>
</tr>
</thead>
<tbody>
<tr>
<td>10*</td>
<td>7</td>
</tr>
<tr>
<td>01*</td>
<td>5</td>
</tr>
<tr>
<td>110*</td>
<td>3</td>
</tr>
<tr>
<td>1011*</td>
<td>5</td>
</tr>
<tr>
<td>0001*</td>
<td>0</td>
</tr>
<tr>
<td>01011*</td>
<td>7</td>
</tr>
<tr>
<td>00010*</td>
<td>1</td>
</tr>
<tr>
<td>001100*</td>
<td>2</td>
</tr>
<tr>
<td>1011001*</td>
<td>3</td>
</tr>
<tr>
<td>1011010*</td>
<td>5</td>
</tr>
<tr>
<td>0100110*</td>
<td>6</td>
</tr>
<tr>
<td>01001100*</td>
<td>4</td>
</tr>
<tr>
<td>10110011*</td>
<td>8</td>
</tr>
<tr>
<td>10011000*</td>
<td>10</td>
</tr>
<tr>
<td>01011001*</td>
<td>9</td>
</tr>
</tbody>
</table>
Address Lookup Using Tries

- Prefixes stored in binary trie.
- Green nodes denote terminal nodes for prefixes.
- Search for address, by using address bits to traverse path from route.
- Remember most recent green node visited.
- If search ends at leaf, then exact match.
- If search ends because no matching branch, go back to last green node.
- Number of memory accesses proportional to address length (32 in IPv4, 128 in IPv6).
Compressed Tries

- Space requirements for trie can grow as product of number of prefixes and address length.
- Compressed trie eliminates all non-termination nodes with single child.
  - number of nodes less than twice number of prefixes
  - reduction in tree depth can improve average search time
  - complicates the node data structures and traversal code
- This approach was common standard until recently.
- Lookup times of several microseconds
  - lookup time largely determined by number of memory accesses
Multibit Tries

- Multibit tries match several bits at once.
- Longest prefix length stored where search ends.
- Greatly speeds search at cost of more memory.
  - 8 bit stride gives 4 memory accesses steps for IPv4; first two are typically in cache
  - may double memory usage over trie
  - optimize space usage for given depth using variable length stride and dynamic programming
- Like standard tries, supports in-place modification.
**Length-Based Search**

- Organize prefixes by length.
- Use hashing to check if i-prefix of address is in row i.
- To find longest match, start at bottom row and work up.
- # of hash ops equals address length in worst-case.

|    | 0*     | 11* 01* | 000* 001* 100* | 0001* 1011* 1110* | 00001* 01010* 01011* 10000* 10001* | 000101* 010110* 101101* 111100* | 0000000* 0101011* 1011011* 1111010* 1111011* | 01010110* 01011101* 10110000* 10110001* 10110101* 11110001* | 010101011* 111101001* | 0101010111* 1011000111* 1111010001* | 0101010100* 10110000010* 11110100010* | 010101011001* 101100011011* 111101000110* |
|----|--------|---------|----------------|-------------------|------------------------------------|------------------------------------|--------------------------------------------------|-----------------------------------------------|-----------------------------------------------|-----------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------|

Jonathan Turner 3/24/98 7
## Binary Search with Markers

<table>
<thead>
<tr>
<th>Search Order</th>
<th>Prefixes</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0* 1*</td>
</tr>
<tr>
<td>2</td>
<td>11* 01*</td>
</tr>
<tr>
<td>3</td>
<td>0.1* 1.* 4.* 5.* 7.* 2.*</td>
</tr>
<tr>
<td>4</td>
<td>0.1* 5.1* 7.0* 0.0* 2.1* 4.0*</td>
</tr>
<tr>
<td>5</td>
<td>0.01* 2.10* 2.11* 4.00* 4.01*</td>
</tr>
<tr>
<td>6</td>
<td>05.* 26.* 55.* 74.* 00.* 25.* 75.* 27.* 54.*</td>
</tr>
<tr>
<td>7</td>
<td>00.0* 25.1* 55.1* 75.0* 75.1* 27.0* 54.0* 55.0* 74.0*</td>
</tr>
<tr>
<td>8</td>
<td>25.10* 27.01* 54.00* 54.01* 55.01* 74.01*</td>
</tr>
<tr>
<td>9</td>
<td>253.* 751.* 543.* 750.* 252.* 540.*</td>
</tr>
<tr>
<td>10</td>
<td>253.1* 543.1* 750.1*</td>
</tr>
<tr>
<td>11</td>
<td>252.00* 540.10* 750.10* 253.00* 543.01* 750.11*</td>
</tr>
<tr>
<td>12</td>
<td>2531.* 5432.* 5433.* 7506.*</td>
</tr>
</tbody>
</table>

- Add marker strings to guide search to possibly better matches.
- For each prefix $y$, include $\text{bmp}_S(y)$ in entry (not shown).
Binary Search Algorithm

Let $S$ be set of original strings of length $\leq W$, including empty string, $\varepsilon$. Let $S'$ be set, augmented with markers and let $S_i, S'_i$ be subsets of length $i$.

function search($x$)
    $B := \text{bmp}_S(\varepsilon)$;  // best matching prefix of $\varepsilon$ is $\varepsilon$
    $lo := 0; hi := W$;
    while $lo \leq hi$ do
        $i := (lo+hi)/2$;
        $y := x[1..i]$;  // selects first $i$ bits
        if $y \in S'_i$ then
            $B := \text{bmp}_S(y)$;
            $lo := i$;
        else
            $hi := i-1$;
        fi;
    od;
    return $B$;
end;
Initializing the Data Structure

let \( S'_i := S_i \) for all \( i \);
for \( y \) in \( S'_i \), let \( \text{bmp}(y) := y \);
build(0,W);

function build(lo,hi)
    if \( lo > hi \) then return fi;
    i := \((lo+hi)/2\);
    build(lo,i-1);
    Q := set of length \( i \) prefixes of strings in \( S_{i+1}..S_{hi} \);
    \( S'_i := S'_i \cup Q \);
    for each \( y \) in \( S'_i \) do
        \( \text{bmp}(y) := \text{search}(y) \);
    od;
    build(i+1,hi);
end;
Avoiding Unnecessary Probes

- When match is found, need only search rows that contain extensions of matched string.
- Store rows worth searching with every string.
- Probe rows in “binary search order.”
- Different markers needed in this case.
- Omit, redundant probe rows.

<table>
<thead>
<tr>
<th>1</th>
<th>0*/ 2</th>
<th>1*/ 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>11*</td>
<td>01*</td>
</tr>
<tr>
<td>3</td>
<td>0.*/ 4,5</td>
<td>1.<em>/ 4.</em>/ 5</td>
</tr>
<tr>
<td>4</td>
<td>0.1*/ 5.1*</td>
<td>7.0*</td>
</tr>
<tr>
<td>5</td>
<td>0.01*</td>
<td>2.10*</td>
</tr>
<tr>
<td>6</td>
<td>05.<em>/ 26.</em></td>
<td>55.*/ 7,8</td>
</tr>
<tr>
<td>7</td>
<td>00.0*/ 25.1*</td>
<td>8</td>
</tr>
<tr>
<td>8</td>
<td>25.10*/ 27.01*</td>
<td>54.00*/ 54.01*/ 55.01*/ 74.01*</td>
</tr>
<tr>
<td>9</td>
<td>253.*/ 10,12</td>
<td>751.*</td>
</tr>
<tr>
<td>10</td>
<td>253.1*/ 543.1*/ 750.1*/ 11,12</td>
<td>540.1*/ 11</td>
</tr>
<tr>
<td>11</td>
<td>252.00*/ 540.10*/ 750.10*/ 750.11*/ 12</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>2531.*</td>
<td>5432.*</td>
</tr>
</tbody>
</table>
General Search Algorithm

function search(x)
    P := ProbeRows(ε);
    B := bmp_ε(ε);
    * lo := 0; hi := W;
    while P is not empty do
        i := P[1]; P := P[2 ..]; // removes first item from list
        y := x[1..i];
        if y ∈ S'_i then
            B := bmp_y;
            P := ProbeRows(y);
            * lo := i;
        else
            * hi := i-1;
        fi;
    return B;
end;
Correctness Issues

- Search algorithm is correct only if data structure is initialized properly (duh!).
  - markers at all the right locations
  - lists of probe sites combine to ensure that right match is found

- Key elements of correctness argument
  - if $x$ is any search string and $y \in S'_i$ is a prefix of $x$ then
    \[
    \text{bmp}_S(x) = \text{bmp}_S(y) \text{ or } | \text{bmp}_S(x) | \in \text{scope}_S(y)
    \]
    where \( \text{scope}_S(y) \) is the complete list of rows that contain extensions of $y$
  - the “transitive closure” of \( \text{ProbeRows}(y) \) is the “relevant” part of \( \text{scope}_S(y) \)
Initializing Data Structure

let $S'_i := S_i$ for all $i$;
for $y$ in $S'_i$, let $bmp(y) := y$; $ProbeRows(y) := \varepsilon$;
$P :=$ any subsequence of $[W..1]$ that includes the smallest $i > 0$ for which $S_i \neq \emptyset$;
built($\varepsilon, P, 0, W$);

function build($y, P, lo, hi$)
    if $P = \emptyset$ then return $f$;
    $i := P[1]$;
    build($y, P[2..], lo, i-1$);
    $Q :=$ set of length $i$ prefixes of strings in $S_{i+1} .. S_{hi}$ that extend $y$;
    for each $z$ in $Q$, let $scope(z)$ be lengths of prefixes in $S_{i+1} .. S_{hi}$ that extend $z$,
in decreasing order;
    for each $z \in Q$ do
        $ProbeRows(z) :=$ any subsequence of $scope(z)$ including smallest value;
        if $z \notin S'_i$ then $Q := Q \cup \{z\};$ $bmp(z) := search(z);$ $f$;
        build($z, ProbeRows(z), i+1, hi$);
    od;
end;
IP Packet Filtering

- For any IP switch port, may specify a list of packet filters
  - generally specified as <src_adr, dest_adr, src_port, dst_port, proto>
  - source and destination addresses may be prefixes
  - ports may be single port or range of port numbers
  - protocol may be specified or unspecified
- An incoming packet is to be processed in accordance with first filter that it matches.
  - may specify discarding, binding to a VC with reserved bandwidth, routing to an encryption processor, etc.
  - packet filters most often installed by network managers for security and high level management
  - ISPs can use to implement virtual private networks for corporations
  - with signaling-driven filter installation, can be used to provide QoS
- Efficient filter matching for large numbers of filters remains difficult at high speeds.
- Lack of IP signaling mechanisms limits utility.
Conclusions

- Algorithm based on binary search can reduce number of memory accesses from $W$ to $\log_2 W$.
  - implies just 40% increase in search time when we go from IPv4 to IPv6, vs. 400% increase using tries and similar methods
- Markers add significantly, but not dramatically to the storage.
- Adding explicit probe site information can reduce number of probes still further; minimal extra storage required.
- Initialization of data structure is tricky to get right but algorithmically simple and efficient - $O$ (sum of prefix lengths).
- Order in which rows are searched can be adjusted to minimize worst-case search time.
- Other new algorithms: multiple-bits-at-a-time tries; systematic prefix expansion to reduce number of prefix lengths