11.1 “Faster IP Lookups Using Controlled Prefix Expansion” by Srinivasan & Varghese

This paper presents a fast way for IP lookups and updates using transformation techniques.

11.1.1 Basic Model

![Diagram showing CPU, CACHE, and DRAM with times and capacities]

**Figure 11.1.** Router Implemented in Software on a PC

IP address lookup requires the longest matching prefix lookup.

**Performance Metrics:**

- 1st metric: # of access to DRAM
- 2nd metric: account for cache + DRAM access

**Recall:**

Internet Lookup Problem: Given IP address, find the longest matching prefix in a routing table and return the interface number.

<table>
<thead>
<tr>
<th>Prefix</th>
<th>Interface #</th>
</tr>
</thead>
<tbody>
<tr>
<td>P_0</td>
<td>I_0</td>
</tr>
<tr>
<td>P_1</td>
<td>I_1</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

**Table 11.1.** Routing Table

2 issues:

- lookup
- dynamic mapping between prefixes and interfaces
11.1.2 Review: 1-Bit Trie - basic data structure

![Diagram of a 1-bit Trie]

**Figure 11.2.** Binary (1-bit) Trie

**Problem:**
- waste time/space => might do DRAM access at each level

=> shrink the tree, collapse path

11.1.3 Basic Idea

- Create Tries (Patricia Tries) with strings on each arch

Couple ways to think about this:

- **A Stride** of a node is the length of the strings labeling its outputs.

  ![Diagram of a 1-bit Trie with stride]

  => no reason to be consistent, just make sure that each node records what stride it is.

- **Fixed stride:** All nodes at one level have the same stride.
  
  01 101 10 1101

- **Variable stride:** no restrictions!

  Problem: lookups in fixed stride trie can only store IP address at the nodes.

  Example:
11.1.4 Solution: “Control Prefix Extension”

- Expand any prefix that would end up in the middle of a stride to the next stride length.

Expansion:
111* want to expand to length 5:
11100*
11101*
11110*
11111*

Problem:
Example 1:
0*
01*

0* expands to:
- 00*
- 01* - but already have this => erroneous

Have to keep track of what you already have (previously declared prefixes)

Example 2:
0* -> 17 => 00*->17, 01*->17
01* -> 35

00* -> 17
01* -> 35

Want to delete 01*, but want the result to be:
00*->17
01*->17

Cost trade-off:
- efficient memory lookup (# of lookups decreased)
- wasted space

Solution: Decide the depth of the tree, then design a tree with the depth at most that.
Example: (Figure 1 and Figure 2 from the paper)

<table>
<thead>
<tr>
<th>Original</th>
<th>Expanded (3 levels)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P_5=0^*$</td>
<td>00*(P_5)</td>
</tr>
<tr>
<td>$P_1=10^*$</td>
<td>01*(P_5)</td>
</tr>
<tr>
<td>$P_2=111^*$</td>
<td>10*(P_1)</td>
</tr>
<tr>
<td>$P_3=11001^*$</td>
<td>11*(P_4)</td>
</tr>
<tr>
<td>$P_4=1^*$</td>
<td>11100*(P_2)</td>
</tr>
<tr>
<td>$P_6=1000^*$</td>
<td>11101*(P_2)</td>
</tr>
<tr>
<td>$P_7=100000^*$</td>
<td>11110*(P_2)</td>
</tr>
<tr>
<td>$P_8=1000000^*$</td>
<td>11111*(P_2)</td>
</tr>
</tbody>
</table>

Table 11.2. Controlled Expansion of the Original Database
Insertion is complicating. For each node build 1-bit trie to record the difference. From the example above:

\[ P_2 = 11||1^* \]
\[ P_3 = 11||001^* \]

Add \( P_9 = 1100^* \) \( \Rightarrow \) 11000*, 11001*
Add $P_{11} = 1110 \Rightarrow 11100^*, 11101^*$

Pack sparse nodes
- loose bits as an offset, might have to do something else, but might fit into cache

Leaf-push

Optimization
- How do you choose stride lengths?

Idea: Dynamic Programming
- choose how many lengths you want
- start with the highest (since need to cover all)
- recurse