Cache-aware data structures for packet forwarding tables on general purpose CPUs: Milestone 2

Maksim Yegorov

2018-03-22 Thu 01:25 PM

presentation highlights

1. What is the longest prefix match problem?
2. What do I propose to do?
   - theme: improve upon linear search
   - algorithm: optimal binary search tree (Knuth)
   - algorithm: IP lookup in the table
   - algorithm: build the lookup table
   - testing program
3. Goals for milestone 3

context

- task: match destination IP `1.2.3.4` from an incoming packet
- Classless Inter Domain Routing (CIDR) tradeoff: use limited IP address space efficiently at the cost of lookup complexity
- router forwarding table

```
   Network           Next Hop
...
*> 1.2.0.0/16        203.133.248.254
...
*> 1.2.3.0/24        202.12.28.1
...
```

context (cont.)

Constraints:

- process packets at line speed or drop packets
- target programmable routers on conventional CPUs

Solutions:

- tries
- hash tables
- filters (Bloom, cuckoo)

Bloom filter

- Original network applications of the BF made use of dedicated hardware that enables searching all prefix lengths in parallel
- Later adaptations in software use linear search

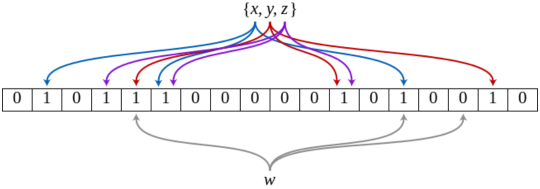

Figure 1: Bloom filter (source: Wikipedia)

```
*> 1.2.0.0/16       x
*> 1.2.3.0/24       y
 > 1.2.3.4          w
```

guided search

- BF involves multiple independent hash functions to insert or look up a prefix
- Use the first hash function, $hash_1$ to serve as a marker:
  - hit $\rightarrow$ look right
  - miss $\rightarrow$ look left
- for subprefixes in the "happy" search path, check up to $\log(N)$ hashes during the search (in case of a balanced tree, N is the number of valid prefix lengths)
- will talk more about signaling later

(optimal) binary search tree

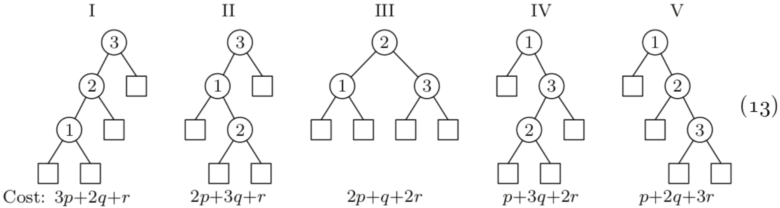

Figure 2: weighted tree cost; source: Knuth

(optimal) binary search tree (cont.)

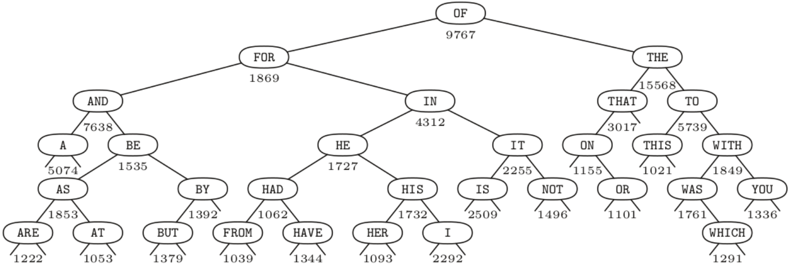

Figure 3: example optimal BST; source: Knuth

use case for BST optimization

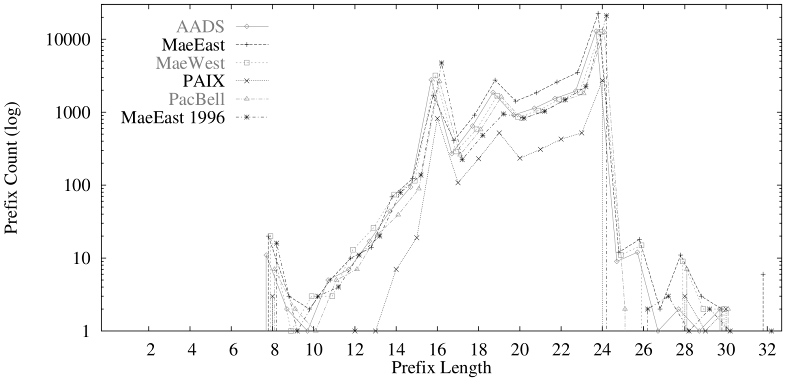

Figure 4: prefix length distribution in backbone routers ca. 2000 (source: Waldvogel et al.)

(optimal) binary search tree: algorithm sketch (Knuth)

Optimal substructure: each subtree T' of an optimal BST T must itself be optimal.

When adding a root node to optimal BST subtrees, the total tree cost becomes:

$$e_{i,j} = w_r + (e_{i,r-1} + w_{i,r-1}) + (e_{r+1,j} + w_{r+1,j})$$

Seek the root that minimizes the tree cost:

$$r_{i,j} = \underset{i \leq r \leq j}{\operatorname{argmin}} \{e_{i,r-1} + w_{i,j} + e_{r+1,j}\}$$

what's an optimal weighting for prefix distribution?

- ultimately depends on the traffic (unknown)
- experiment with:
  - balanced tree
  - weighting based on the fraction of IP address space covered by prefixes of given length
  - weighting based on the fraction of prefixes in table of given length

algorithm: IP lookup in the table

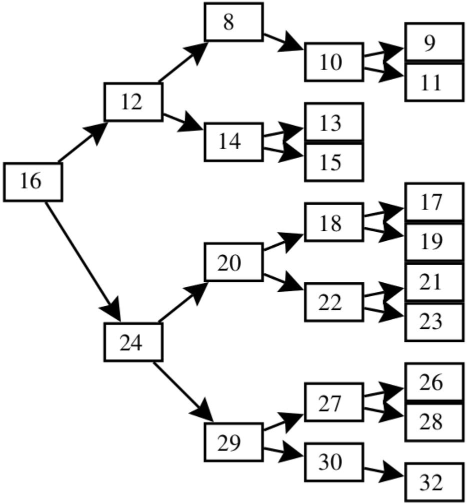

Figure 5: tree traversal

algorithm: IP lookup in the table (cont.)

- Bloom filter uses $hash_{1..k}$ for set membership ($k \approx 20$)
- $hash_1$ serves as a marker reserved for guiding the search:
  - `miss` $\rightarrow$ look left, else goto most recent $prefix_{hit}$, else $prefix_{default}$
  - `hit` $\rightarrow$ remember this prefix length, increment $count_{hit}$ & look right
- if `hit` with `nil` right child or if returning to most recent $prefix_{hit}$, decode $hash_{count_{hit}..count_{hit}+n}$ to calculate *best matching prefix* for $prefix_{hit}$, finish matching with $hash_{n+1..k}$
- edge cases:
  - if none of $hash_{count_{hit}..count_{hit}+n}$ succeed $\rightarrow$ default path
  - if all of $hash_{count_{hit}..k}$ succeed $\rightarrow$ found match
- n is 5 for IPv4, 6 for IPv6 (bits to encode valid prefix lengths, implicitly assuming $n \geq count_{hit}$)

algorithm: IP lookup in the table (cont.)

- guided search fails on false positives
- false positive probability is exceedingly small (put otherwise, probability >> chance if any single hash matches)
- on false positive, default to linear search of prefix lengths $\leq prefix_{hit}$

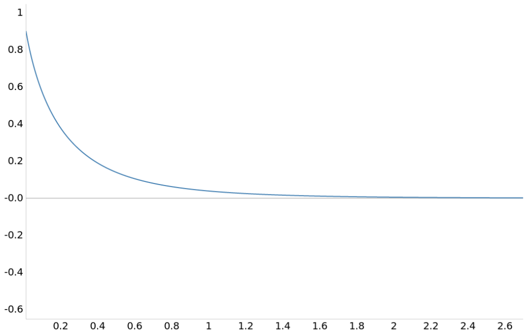

Figure 6: false positive probability vs num hash functions (k)

algorithm: build the lookup table

- idea: precompute traversal paths at insertion
- compute optimal BST to direct the search
- sort prefixes prior to inserting into the Bloom filter, used to find the *best matching prefix* for given prefix
- insert prefixes in the sorted order, as follows:
  - find BMP for given prefix
  - traverse the BST in search of the prefix length, insert $hash_1$ markers for *branch right* up to and inclusive of the prefix length, increment $count_{hit}$
  - insert prefix into Bloom filter using $hash_{1..k}$ (can't be reduced to allow for linear search all else failing)
  - encode pointers (binary encoding for *best matching prefix* length) at each ancestor with marker (i.e. each ancestor where we branched right) on the path to this prefix, using $hash_{count_{hit}..count_{hit}+n}$
    - edge case: ancestor is itself a BMP $\rightarrow$ previously encoded with $hash_{count_{hit}..k}$
  - take care to increment $count_{hit}$ going down the tree, decrement when going up the tree

testing program: linear vs. guided search

1. metrics: average number of matches per IP + packets (IPs) per second
2. false positive rate:
   - increase in false positive rate because of inserted pointers
   - what fraction of prefixes in the table have a shorter *best matching prefix* (other than default)
3. effects of simulated traffic pattern:
   - randomly generated traffic over all address space (with default route)
   - randomly generated from address space covered by prefixes (no default route)
   - traffic correlated with prefix table (prefix length distribution, fraction of address space covered by each prefix)
   - effects of optimal binary search tree (balanced vs. weighted by covered address space vs. weighted by fraction of prefix length)
4. performance on IPv4 vs IPv6 prefixes

summary

- tightened the algorithms since last presentation
- defined the testing program
- working on implementation

milestone 3

- complete implementation
- analyze results from experiments