Cache-aware data structures for packet forwarding tables on general purpose CPUs: Milestone 1

Maksim Yegorov

2018-02-26 Mon 10:29 PM

context

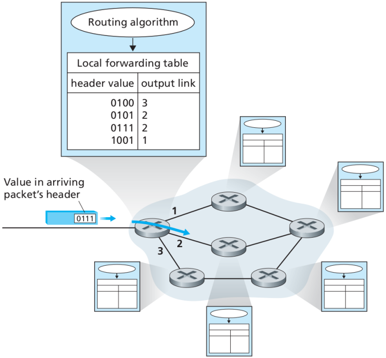

Figure 1: forwarding; from Kurose & Ross, *Computer Networking*

problem: demand > capacity

- process packets at line speed or drop packets
- FIB table exhibits polynomial growth
- target programmable routers on conventional CPUs where:
  - DRAM latency almost unchanged for decades
  - workhorse data structures must fit in limited CPU cache
  - limited budget of CPU cycles per packet

classes of proposed data structures

- tries: too much overhead to fit in cache; best solutions patented
- binary search on prefix ranges (Lampson, Srinivasan, Varghese) and prefix lengths (Waldvogel)
- filters (Bloom, cuckoo)

filters

Opportunities for improvement:

- lots of lookups required to find longest prefix match:
  - 8..32 prefix lengths (IPv4)
  - 19..128 prefix lengths (IPv6)
- lookup in off-chip hash table is prohibitively expensive (TBD)

take inspiration from binary search

Try to **nudge filter lookup in the right direction** by quasi-false-positives:

Algorithm 1 investigate: hypothetical heuristic

Input: BF, IP, $8 < i < 32$

Output: j (next prefix length to consider) or i (done)

```
h ← hash(IP[0:i])
if BF.lookup(h) = 1 then
    investigate j > i
    if BF.match(IP[0:i]) then
        return i
    end if
else
    investigate j < i
end if
```

false positive probability vs. number of hashes

- false positive probability is exceedingly small (put otherwise, probability $>>$ chance if any single hash matches)
- empirical fact: uncommon that table includes several prefixes within same subsequence

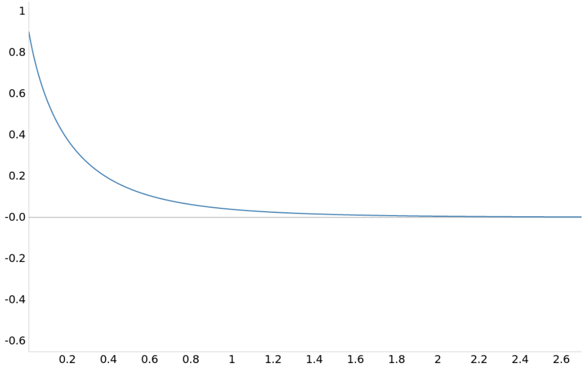

Figure 2: false positive probability vs num hash functions

more plausible heuristic

Proceed in logarithmic stride:

1. When populating the filter, apply $hash_1$ to subprefixes in search path (at most, $\log(N)$ hashes: e.g. for a $prefix_{32}$ of length 32, apply $hash_1$ to $prefix_{\{16,24,28,30\}}$)
2. apply up to $k - \log(N)$ remaining hash functions to the *prefix* (e.g. apply $hash_{2..(k-4)}$ to $prefix_{32}$ in this example)
3. start lookup on $prefix_{16}$ (or perhaps most frequent prefix length) and climb if at least non-zero match
4. remember:
   - longest match to date,
   - computed hashes

edge cases to consider

- iff given prefix is in fact a set member, we can always ascertain this in $\log(N)$ steps
- but what is the effect of false positives? may need to backtrack

summary

Recall what we set out to do:

- lots of lookups required to find longest prefix match:
  - 8..32 prefix lengths (IPv4)
  - 19..128 prefix lengths (IPv6)

summary

- approach lookup in FIB filters in a way similar to hill climbing and binary search based approaches
- need to experiment, see the effect on false positives, tweak params as needed, see if can parallelize search
- on a different note: still need to think about off-chip table lookup