Cache-aware data structures for packet forwarding tables on general purpose CPUs: Milestone 3

Maksim Yegorov

2018-04-14 Sat 06:04 PM

presentation highlights

1. What is the longest prefix match problem?
2. What do I propose to do?
   - theme: improve upon linear search
   - algorithm: IP lookup in the table using guided search
   - algorithm: build the lookup table using guided search
   - experiments
3. Suggestions for the future.

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
...
```

deceptively simple

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

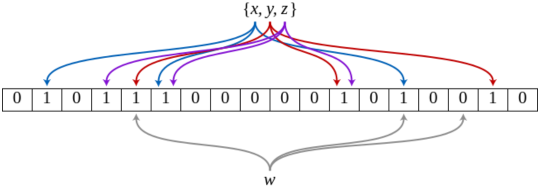


Figure 1: Bloom filter (source: Wikipedia)

```
*> 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 → look right
  - miss → look left
- check $\log(N)$ hashes during the search (in case of a balanced tree, N is the number of valid prefix lengths)
- details on build/lookup algorithms below

(optimal) binary search tree

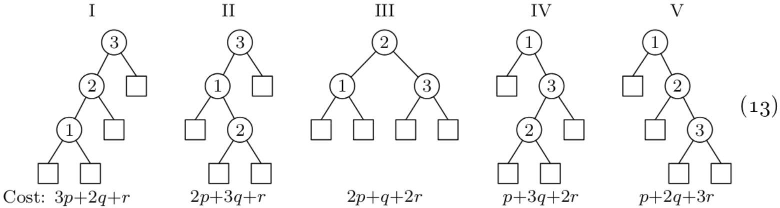


Figure 2: weighted tree cost; source: Knuth

use case for BST optimization

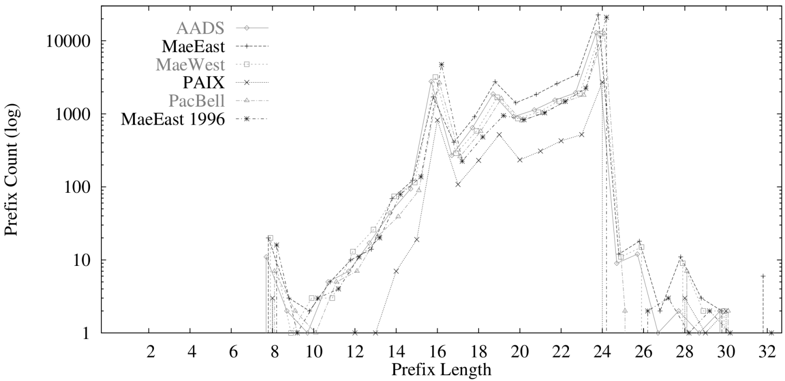


Figure 3: prefix length distribution in backbone routers ca. 2000 (source: Waldvogel et al.)

use case for BST optimization (cont.)

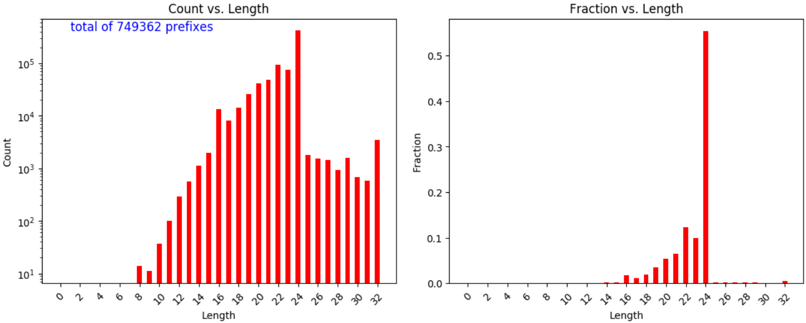


Figure 4: prefix length distribution today: count (Route-Views.Oregon-ix.net)

use case for BST optimization (cont.)

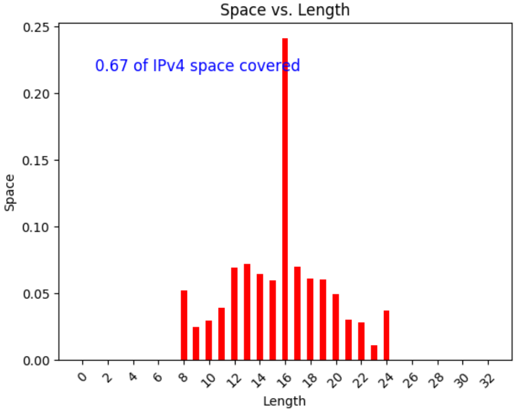


Figure 5: prefix length distribution today: space covered (Route-Views.Oregon-ix.net)

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

(optimal) binary search tree

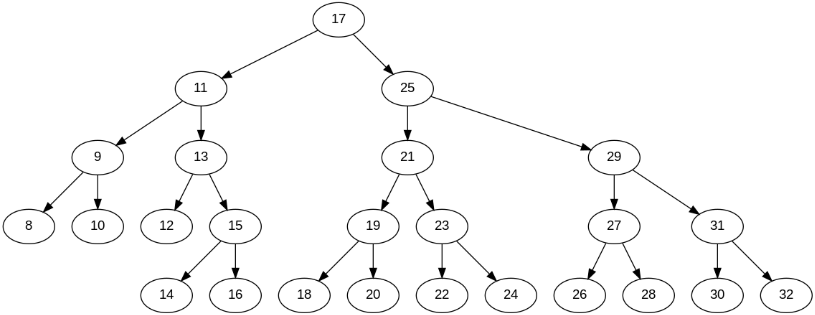


Figure 6: balanced tree: IPv4

(optimal) binary search tree (cont.)

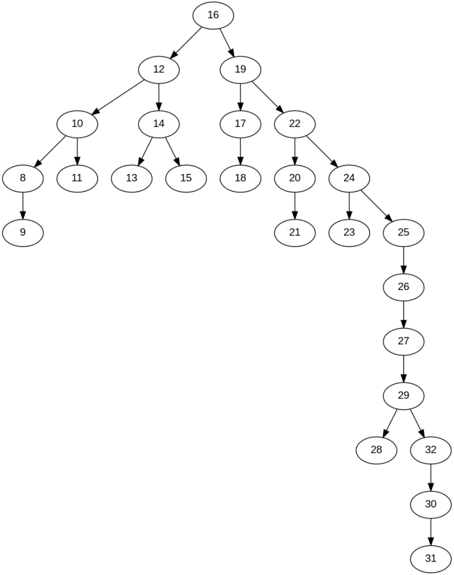


Figure 7: weighted by space covered: IPv4

(optimal) binary search tree (cont.)

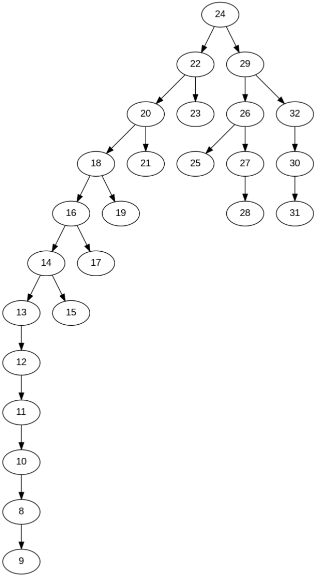


Figure 8: weighted by relative count of prefix lengths: IPv4

algorithm: IP lookup in the table

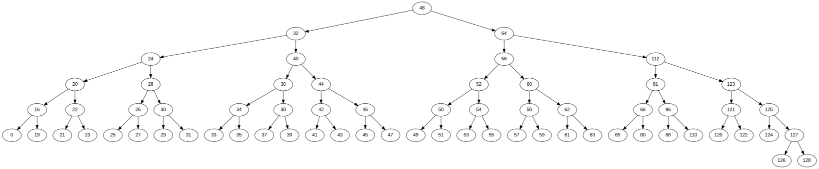


Figure 9: tree traversal

algorithm: IP lookup in the table (cont.)

- Bloom filter uses $hash_{1..k}$ for set membership ($k \approx 10 - 20$)
- $hash_1$ serves as a marker reserved for guiding the search:
  - `miss` → look left, else goto most recent $prefix_{hit}$, else $prefix_{default}$
  - `hit` → 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 → default path
  - if all of $hash_{count_{hit}..k}$ succeed → 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.)

- challenge: guided search fails on false positives
- solution: keep the bit array sparse → probability $>>$ chance if any single hash matches
- on false positive, default to linear search of prefix lengths $< \mathit{prefix}_{hit}$

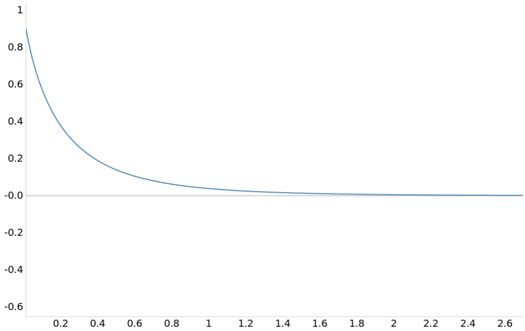


Figure 10: FPP vs num hash functions for "optimal" BF

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
- insert prefixes in the sorted order, as follows:
  - find BMP, if exists, for given prefix via lookup in `BF` built to date
  - traverse the BST in search of the prefix length, insert $hash_1$ markers to signal *branch right* up to and inclusive of the prefix length, increment $count_{hit}$
  - insert prefix into `BF` using $hash_{1..k}$ (can't be reduced to allow defaulting to linear search if all else fails)
  - 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 $hash_{count_{hit}..count_{hit}+n}$
    - side note: if ancestor is itself a BMP → previously encoded with $hash_{1..k}$
  - take care to increment $count_{hit}$ going down the tree, and keep it handy

testing program: linear vs. guided search

1. metrics: average number of lookups/hashing per IP
2. collect interesting stats:
   - 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:
   - 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

results: % bitarray full and number of hash functions

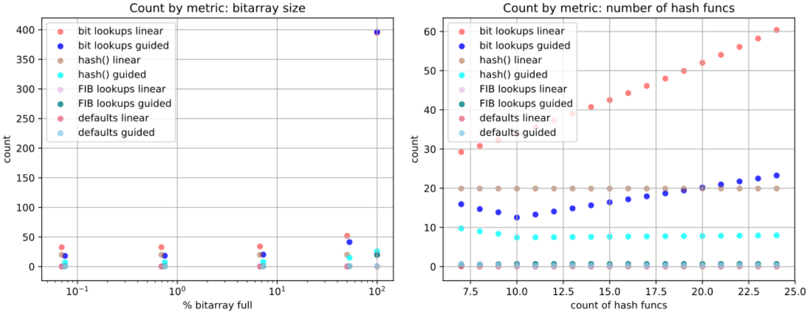
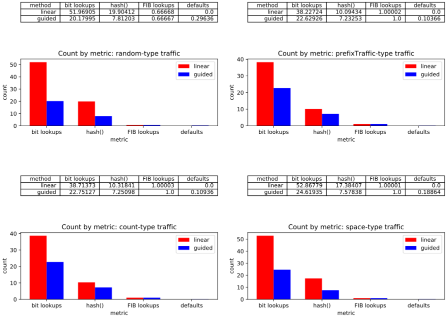


Figure 11: IPv4 trends

results: traffic type



method	bit lookups	hash()	FIB lookups	defaults
linear	51.96905	19.90412	0.66668	0.0
guided	20.17995	7.81203	0.66667	0.29636

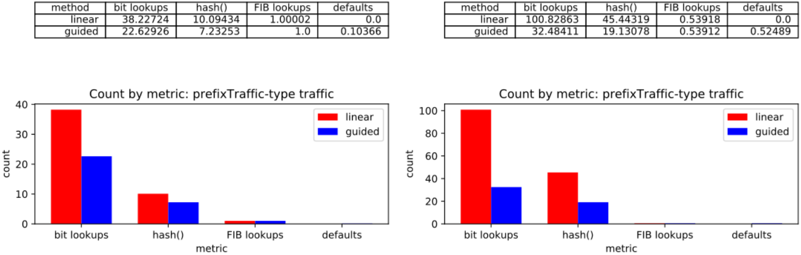
method	bit lookups	hash()	FIB lookups	defaults
linear	38.22724	10.09434	1.00002	0.0
guided	22.62926	7.23253	1.0	0.10366

method	bit lookups	hash()	FIB lookups	defaults
linear	38.71373	10.31841	1.00003	0.0
guided	22.75127	7.25098	1.0	0.10936

method	bit lookups	hash()	FIB lookups	defaults
linear	52.86779	17.38407	1.00001	0.0
guided	24.61935	7.57838	1.0	0.18864

Figure 12: IPv4 trends: traffic type

results: IPv4 vs IPv6 with prefix traffic

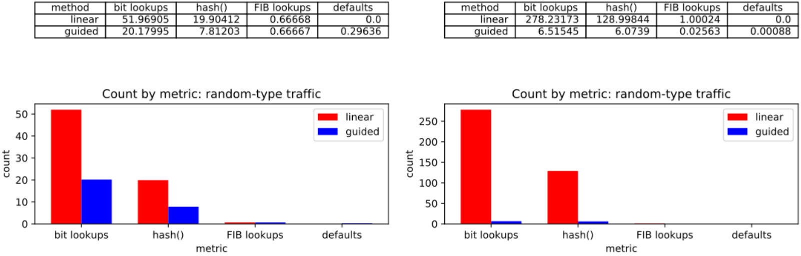


method	bit lookups	hash()	FIB lookups	defaults
linear	38.22724	10.09434	1.00002	0.0
guided	22.62926	7.23253	1.0	0.10366

method	bit lookups	hash()	FIB lookups	defaults
linear	100.82863	45.44319	0.53918	0.0
guided	32.48411	19.13078	0.53912	0.52489

Figure 13: IPv4 vs IPv6: highly non-random traffic

results: IPv4 vs IPv6 with random traffic

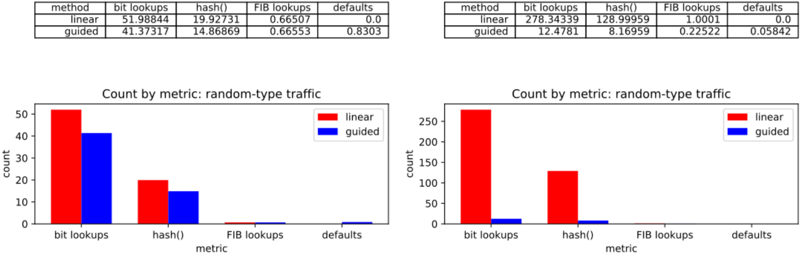


method	bit lookups	hash()	FIB lookups	defaults
linear	51.96905	19.90412	0.66668	0.0
guided	20.17995	7.81203	0.66667	0.29636

method	bit lookups	hash()	FIB lookups	defaults
linear	278.23173	128.99844	1.00024	0.0
guided	6.51545	6.0739	0.02563	0.00088

Figure 14: IPv4 vs IPv6: random traffic

results: IPv4 vs IPv6 with ~2.6MB BF and k=20

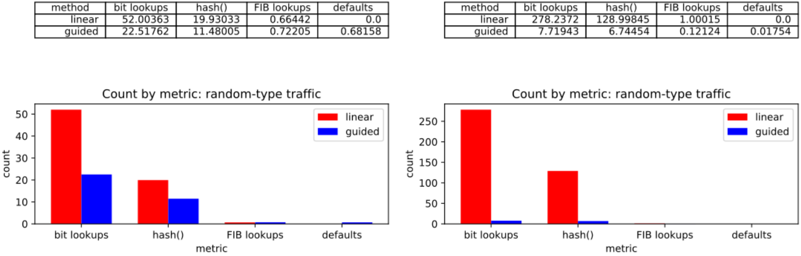


method	bit lookups	hash()	FIB lookups	defaults
linear	51.98844	19.92731	0.66507	0.0
guided	41.37317	14.86869	0.66553	0.8303

method	bit lookups	hash()	FIB lookups	defaults
linear	278.34339	128.99959	1.0001	0.0
guided	12.4781	8.16959	0.22522	0.05842

Figure 15: IPv4 vs IPv6: compact with k=20

results: IPv4 vs IPv6 with ~2.6MB BF and k=10



method	bit lookups	hash()	FIB lookups	defaults
linear	52.00363	19.93033	0.66442	0.0
guided	22.51762	11.48005	0.72205	0.68158

method	bit lookups	hash()	FIB lookups	defaults
linear	278.2372	128.99845	1.00015	0.0
guided	7.71943	6.74454	0.12124	0.01754

Figure 16: IPv4 vs IPv6: compact with k=10

summary

- implemented the algorithms
- ran experiments
- if have time, will implement in C for the absolute throughput metrics (packets (IPs) per second, not meaningful for current `Python` prototype)
- key insight: some datastructures (even highly compact) have the excess capacity to piggyback on their overhead and use it as a protocol for a simple message encoding
- apply the same method to optimize the off-chip hash table lookup? This is a many-to-one case (now with outgoing link indices in place of prefix lengths)