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

Maksim Yegorov
Computer Science Department
Rochester Institute of Technology, NY
Email: mey5634@rit.edu

***Abstract*—Longest prefix matching has long been the bottleneck of the Bloom filter-based solutions for packet forwarding implemented in software. We propose a search algorithm that allows for a compact representation of the FIB table in cache on general purpose CPUs with an average performance target of $\mathcal{O}(\log n)$ for n-bit IP addresses.**

1. Background

To set the context, consider the fundamental task of a network router. In order to match an incoming packet to an outgoing interface link, the router inspects the packet's header to obtain the destination address. It will then consult a forwarding (Forwarding Information Base) table stored on the router. An address entry in such a table will contain, among other things, a variable length prefix (as in 129.12.30.0/20). In effect, the router will compare the destination IP against the known prefixes using the longest prefix match rule. The forwarding table itself may be occasionally updated with new prefixes received via BGP route advertisement [1].

It would appear that a standard hash table or a binary search tree could satisfy the requirements of a data structure that permits the *look-up* and *update* operations. The difficulty with adapting the well-known datastructures for Internet routing stems mainly from the sheer throughput demand imposed by today's line speed and the FIB table size. This can be easily shown on the example of a hypothetical 50Gbps core router and an Ethernet frame of 84 bytes for a minimum sized packet. The bit per second wire line speed can be framed in terms of packets per second. Specifically, for this simplified scenario, we can expect to process 75Mpps. Depending on the algorithm that we pick, this may require at least 75 million lookups per second (or an order of magnitude more). For a general purpose CPU clock speed of (for the sake of example) 4 GHz, this equates to approximately 50 CPU cycles per packet. If the router cannot keep up with the speed of arriving packets, it will drop packets.

How much work can be done in 50 CPU cycles? As a very rough approximation, consider that conventional hashing algorithms require about 10 cycles per byte of hash (40 cycles to compute a 32 bit hash). Memory latency presents a particular challenge. The access times range between 4-50 cycles for L1 and L3 CPU caches, respectively. The penalty for misses can easily double the time requirements. Main memory access will require several hundred cycles.

Clearly, this hypothetical scenario is an oversimplication. The forwarding task is only one among many processing steps that a router performs on each packet, contemporary CPUs will likely have multiple cores, router line speeds may be in the single digits or in the hundreds of Gbps, there will be a distribution of packet sizes (my estimate errs on the conservative side), leaf node routers may benefit from caching previously seen IP addresses etc. Still, the above generalization gives us a ballpark number to quickly determine if a particular datastructure is fit for the task.

In view of these numbers, it is not at all surprising that the lookup has been traditionally performed in hardware, using dedicated TCAM and SRAM circuits. There are multiple considerations that make software implementations superior to ASIC hardware based ones. The cost per transistor, power requirements, and monopoly effects, in particular, drive up the cost. The inability to patch hardware makes security updates unfeasible. There has been a renewed push recently to develop programmable routers that, on the one hand, can accommodate the data processing speeds expected of today's networks, and on the other, offer the option to implement and continuously update various parts of the network stack in software rather than hardware.

2. Related Work

Classical algorithms developed up to about 2007 have been surveyed in [2] and [3]. The data structures include trie, tree, and hash table variants.

Of particular relevance is the binary search on prefix lengths proposed in [4]. Waldvogel et al. propose a very elaborate hash table of binary search trees with logarithmic time complexity. Most of the refinements involve comparatively large databases that require at least an order of magnitude more memory than what can fit into third level cache, and are therefore only practical for hardware implementations. We believe that the core ideas of leveraging the binary search on prefixes and using memoization to avoid backtracking when the search fails can be adapted to the more compact data structures.

Dharmapurikar et al.[5] describe a longest prefix matching algorithm utilizing a probabilistic set membership check with Bloom filters. A Bloom filter is associated with each

prefix length. The destination adress is masked and matched against each of the Bloom filters, yielding a list of one or more prefix matches. The list is then checked against an off-chip conventional hash table, starting with the longest prefix match. Because of the arbitrarily small false positive rate, a single lookup in high-latency main memory is sufficient in practice.

3. Solution

3.1. Goals

We have identified two opportunities for improvement in the context of the Bloom filter-based solutions to the longest prefix matching problem. The Bloom filter (BF) data structure was originally used by Dharmapurikar et al. [5] for parallel look up implemented in hardware. By contrast, the software implementations on conventional hardware pay a hefty penalty – in computation cost and code complexity – to parallelize the look up. The default solution has been a linear search (see Algorithm 1). The time complexity of Algorithm 1 is $\mathcal{O}(n)$, where n is the number of distinct prefix lengths in the BF. We propose to improve on the linear search for Bloom filter in this paper.

Algorithm 1 Linear search for longest matching prefix

```
1:  procedure LINEARSEARCH(bf, ip, fib, maxlen)
2:      plen ← maxlen                          ▷ max prefix length
3:      while plen ≥ MINLEN do                 ▷ min prefix length
4:          tmp ← extract plen most significant bits in IP
5:          key ← encode(tmp, plen)
6:          res ← bf.contains(key,
7:                            [hash_1..hash_bf.k])
8:          if res ≠ 0 & key ∈ fib then
9:              return plen
10:         else plen ← plen − 1
11:         end if
12:     end while
13:     return PREF_DEFAULT                    ▷ default route
14: end procedure
```

Second, any scheme that utilizes a probabilistic data structure, such as the Bloom filter, to identify candidate(s) for the *longest matching prefix* (LMP) would generally need to look up the candidate(s) in a forwarding table that serves as the definitive membership test and the store of the next hop information. Current solutions typically store this information in an off-chip hash table. This operation is therefore a bottleneck of the probabilistic filter-based schemes. While we have not yet implemented or tested the proposed guided search data structure specifically for FIB storage, we conjecture that the method we propose may be broadly applicable to any key-value store application that

(a) is defined as a many-to-few kind of mapping over totally ordered keys, and
(b) tolerates (i.e. self-corrects for) a certain probability of error.

In the case of the FIB table, we propose to store the outgoing link information in a compact array. We would then insert encoded `(index, prefix)` pairs into a *guided BF* data structure (separate from the BF used to encode `(length, prefix)` pairs). For forwarding table applications, this appears feasible on today's off-the-shelf hardware, where we would typically have on the order of a million keys, with array indices (outgoing interfaces) numbering in the low hundreds.

From our preliminary analysis, both Bloom filters ($\texttt{BF}_{\texttt{fib}}$ and $\texttt{BF}_{\texttt{lmp}}$) can fit in L3 cache for the current backbone router FIB table sizes that we've had the opportunity to survey. The implementation and a cost-benefit analysis of such a FIB implementation remain to be done.

3.2. Implementation

The key observation that we draw upon is that any one of the routine tests – whether a particular bit in a Bloom filter bit array is set – contains valuable information, in that the correlation between a set bit and a prefix being a member of the set is much higher than chance (see Fig. 1). The cost of calculations performed as part of validating the membership of a given key in a BF gives us an incentive to assign meaning to specific hashing calculations. In other words, we will define a simple protocol that exploits the overhead associated with the BF hashing calculations to direct our search for the LMP.

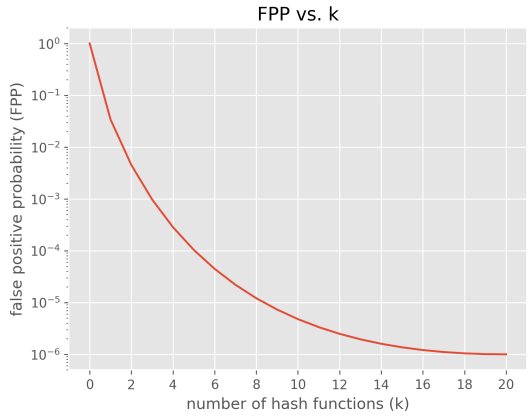


Figure 1: False positive probability vs. number of hash functions in an *optimal* BF

Algorithm 2 contains the pseudocode to *build* the data structure amenable to such a guided search. The underlying idea is to pre-compute in advance the search path for an IP that could possibly match a given prefix – at the time of inserting the prefix into the BF. Algorithm 3 suggests a procedure to *look up* an IP in a data structure built by Algorithm 2. The algorithms assign specific meaning to the *first* hashing function to direct our search left or right in a binary search tree. In addition, we reserve n hashing functions ($n \in \{5, 7\}$ for IPv4 and IPv6 tables in our experiments) to encode the best matching prefix as a bit

sequence. The n bits, when decoded, will index the prefix length in a compact array of the prefix lengths contained in the router's FIB table (e.g., $0 \rightarrow 0$, $1 \rightarrow 8$, etc. for the IPv4 table used in our experiments).

Both the *build* and the *look up* procedures assume an (optimal) binary search tree to guide the search. Such an optimal tree could be constructed for a given router if the historic traffic data and its correlation with the fraction of the address space covered by each prefix length were known. In the absence of such information, we can conservatively assume random traffic and a balanced binary search tree (as in the classic binary search algorithm).

The *build* invokes the *look up* to identify the best matching (shorter) prefix in a BF constructed to date for the (longer) prefix about to be inserted. Therefore, we will first sort the prefixes, before inserting them into the BF in the ascending order.

Algorithm 2 Build a BF to enable guided search for LMP

```
1:  procedure INSERT(bf, pref, fib, bst)
2:      bmp ← Lookup(                          ▷ best match prefix
3:                   bf,
4:                   pref,
5:                   fib,
6:                   bst)
7:      curr ← bst                                  ▷ start at root
8:      count_hit ← 0                    ▷ times branched right
9:      while curr ≠ null do                          ▷ not leaf
10:         if pref.len < curr.plen then
11:             curr ← curr.left
12:         else if pref.len = curr.plen then
13:             key ← encode(pref, pref.len)
14:             bf.ins(key,
15:                    [hash_1..hash_bf.k])
16:             break
17:         else
18:             tmp ← curr.plen most signif bits in pref
19:             key ← encode(tmp, curr.plen)
20:             bf.ins(key, [hash_1])   ▷ signal right
21:             count_hit ← count_hit + 1
22:             hashes ← filter(              ▷ hash funcs
23:                     bmp,                ▷ encode bmp
24:                     hash_count_hit,   ▷ start hash func
25:                     n)                      ▷ num bits
26:             bf.insert(key, hashes)
27:             curr ← curr.right
28:         end if
29:     end while
30: end procedure
```

Algorithm 3 defaults to linear search when a bit that would be unset under the perfect hashing assumption is found set in the actual BF. The dead end scenario can ensue in the course of:

1) the search being directed *right*, where (in hindsight) it should have proceeded *left*;
2) the decoded best matching prefix length is incorrect – either logically impossible (nonsensical prefix length, or prefix length longer than or equal to $last_{hit}$) or failing the BF look up on one of the remaining hash functions;
3) the case of false positive: The prefix is not found in FIB.

In any one of these cases, Algorithm 3 defaults to the linear look up scheme, starting with the longest match to date ($last_{hit}$ in Algorithm 3 pseudocode). Given the number of prefixes to be stored in the BF, we can tune the BF parameters (bit array size m, number of hash functions k) to provide an optimal balance between the size of the data structure in memory (i.e., design the BF to fit in CPU cache), on the one hand, and the rate at which the guided search would default to linear search and the FIB table look up rate, on the other. The cost benefit analysis is a function of the size of L3 cache available, the penalty for off-chip memory hits and misses, the computational cost per byte of hash, and the like – and can be established through grid search and tuned for the target hardware (and traffic, if the details are available).

The time complexity of Algorithm 3 is $\Omega(\log n)$, where n is the number of distinct prefix lengths in the BF. Each search will scale the full height of the binary search tree, stopping at a leaf, then jumping from the most recent $\texttt{hash}_1$ match to the *best matching prefix*, occasionally defaulting to a linear search over the lower prefix lengths. The number of defaults can in principle be controlled in the same way as the false positive rate can be tuned for the standard BF. In practice, the degree to which the default rate can be minimized is limited by the practical considerations of the available CPU cache size.

4. Experiments

4.1. Design of Experiments

Table 1 summarizes the experiments that we have run. The goal has been to compare the performance of the linear and guided search schemes in terms that

(a) are common to both algorithms, and
(b) account for the bulk of CPU and memory access time irrespective of implementation.

In particular, any filter-based implementation will involve repeated testing if a given bit is set in the filter's bit vector, invoking a non-cryptographic hash function, and looking up a candidate prefix match in a FIB table (whether implemented in a BF or a hash table). In the case of the guided BF search, we may also be interested to know, how infrequently the BF (with given parameter settings) defaults to linear search. Obviously, the frequency of defaulting will be also reflected in the other metrics. Consequently, we have instrumented our BF implementation to collect the statistics on the bit lookup, hashing, and FIB table lookup function invocations. The profiling results will be reported per packet (equivalently, per destination IP).

Algorithm 3 Guided search for LMP

```
1:  procedure LOOKUP(bf, ip, fib, bst)
2:      last_hit ← −1                          ▷ last plen that yielded hit
3:      count_hit ← 0                          ▷ times branched right
4:      curr ← bst                             ▷ start at root
5:      while curr ≠ null do                   ▷ not leaf
6:          tmp ← curr.plen most significant bits of IP
7:          key ← encode(tmp, curr.plen)
8:          res ← bf.contains(tmp, [hash_1])
9:          if res = 1 then
10:             count_hit ← count_hit + 1
11:             last_hit ← curr.plen
12:             curr ← curr.right
13:         else
14:             curr ← curr.left
15:         end if
16:     end while                              ▷ reached leaf
17:     if last_hit = −1 then
18:         return PREF_DEFAULT                ▷ default route
19:     end if
20:     tmp ← last_hit most significant bits of IP
21:     key ← encode(tmp, last_hit)
22:     bmp ← bf.contains(                     ▷ decode best match
23:                   key,
24:                   [hash_{count_hit}..hash_{count_hit+n-1}],
25:                   decode=true)
26:     if bmp = 2^n − 1 | bmp < last_hit then
27:         key ← encode best match prefix, as usual
28:         res ← bf.contains(
29:                       key,
30:                       [hash_{count_hit+n}..hash_{bf.k}])
31:         if res = 1 & key ∈ fib then
32:             return bmp
33:         else
34:             return LinearSearch(           ▷ defaulting
35:                           bf, ip,
36:                           fib, last_hit−1)
37:         end if
38:     end if
39: end procedure
```

An off-chip FIB table look up may be framed in terms of CPU cycle cost and set equivalent to some number of lookups in the bitarray in L3 cache. This is a major factor in the cost-benefit analysis when tuning the BF hyperparameters, such as the optimal *false positive probability* setting. Because these metrics depend on the system and implementation, we do not convert them to a common denominator.

The performance of each scheme will depend on the traffic passing through the router in question. The traffic may be more or less correlated with the prefixes in the table. We benchmark both search schemes on three synthetically produced traffic data sets:

1) Random traffic: IP addresses where chosen randomly out of the whole address space. Because of the vast size of the IPv6 address space, without a default route, the absolute majority of such randomly selected addresses would not be matched to a prefix.
2) Traffic is correlated with the fraction of address space covered by the prefixes of a given length. For example, for the IPv4 BGP table that our experiments are based on, 16-bit long prefixes cover close to 1/4 of the total address space. Using reservoir sampling, with approximately 25% probability we generate an address from the range of the subnets that span the `/16` prefixes in the table.
3) Traffic is correlated with the frequency distribution of prefix lengths in the BGP table. For example, 24-bit long prefixes account for approximately 60% of all prefixes in the table. We will randomly generate an address spanned by the `/24` subnets with an approximately 60% probability.

If the traffic passing through a given device could be observed and correlated with the prefix distribution, we would be able to customize the binary search tree to reduce the search time, when ammortized over aggregate traffic. This would involve customizing the optimal binary search tree (e.g. using Knuth's dynamic programming algorithm) through differential weights assigned to each prefix length. The optimal weights would be a function of two parameters: the fraction of traffic correlated with each prefix length and the optimal balance ratio among the height of each tree branch – to avoid gross imbalance in a scheme when each traversal scales the full height of a branch from root to leaf.

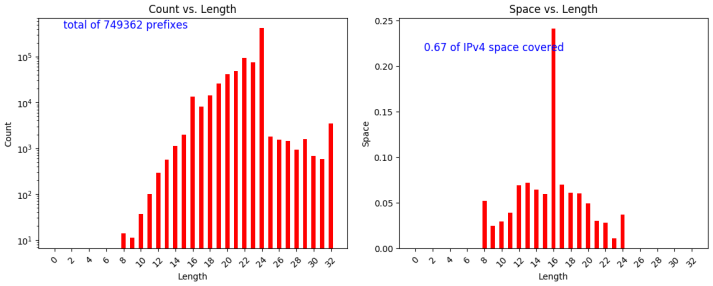


Figure 2: Prefix distribution in the Route-Views.Oregon-ix.net IPv4 table

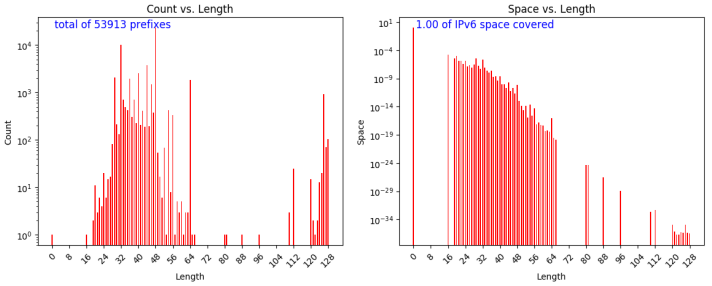


Figure 3: Prefix distribution in the Route-Views.Oregon-ix.net IPv6 table

In the absence of data on traffic associated with the BGP table that we use in our experiments, we merely observe

TABLE 1: Experiment Matrix

traffic	protocol	
	IPv4	IPv6
random	✓	✓
by prefix address space	✓	
by prefix frequency	✓	✓

the effect of each traffic extreme on the performance of the guided vs. linear schemes. In all experiments, we use the balanced binary tree (equivalently, binary search) with the implication that prefix lengths in any one branch of the balanced tree are equally likely to match any one IP.

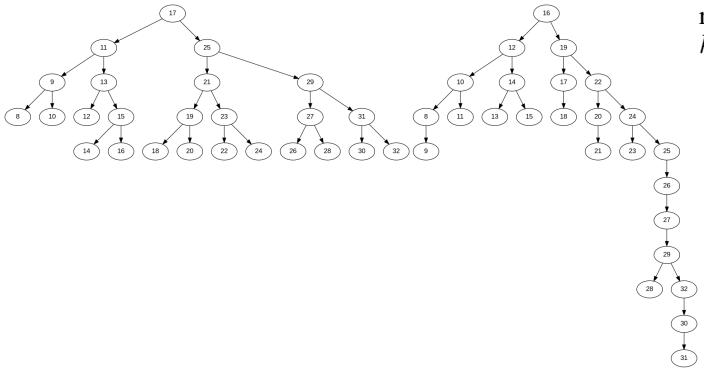


Figure 4: Balanced BST (used in experiments) vs. BST optimized exclusively for address share (not used), both based on the Route-Views.Oregon-ix.net IPv4 table

We also contrast the performance of the linear vs. guided search schemes benchmarked on IPv4 and IPv6 tables. We use two traffic pattern extremes:

1) random traffic, where most if not all IPv6 queries go to default route;
2) traffic correlated with the prefix length frequency distribution, using prefixes themselves as traffic.

Finally, we observe the effect of the BF hyperparameters: the bit vector size (m) – or equivalently, the percentage of bits set – and the count of hash functions (k) on the performance of the linear vs. guided search schemes.

4.2. Discussion

We summarize the results of the experiments with different IPv4 traffic types in Table 2. In general, the *guided search* requires about half the number of accesses on a per packet basis compared to *linear search*.

Search performance vs. utilization ratio (% full) is documented in Table 3. From Figure 5 it can be seen, that the linear vs. guided search relative performance levels off when the bitarray is approximately 10% full. For a table of approx. 750,000 entries, 10% full guided BF requires 10.3MB and can therefore fit in L3 cache on today's general purpose CPUs. The performance for a compact 2.57MB guided filter is still an improvement on linear search results, even though the search defaults to linear 68% of the time. The effect of

TABLE 2: IPv4 linear vs. guided search performance by traffic type; guided BF: $n = 749362$ elements, $k = 10$ hash funcs, $size = 2.57$MB, $length = 21548036$ bits, 33.3% full; linear BF: $fpp = 10^{-4}$, $n = 749362$ elements, $k = 14$ hash funcs, $size = 1.71$MB, $length = 14365358$ bits, 51.8% full

traffic	**metric (per packet)**	**linear**	**guided**
random	bit lookup	49.3	22.1
	hashing	20.0	11.2
by prefix address space	bit lookup	48.0	22.3
	hashing	17.4	10.3
by prefix frequency	bit lookup	33.3	17.7
	hashing	10.3	8.6

TABLE 3: IPv4 guided search performance by utilization ratio (random traffic); guided BF: $n = 749362$ elements, $k = 10$ hash funcs

% bit vector full (size)	**metric**	**count per packet**
33.3% full (2.6MB)	bit lookup	22.1
	hashing	11.2
	FIB lookup	0.7
	default rate	68%
9.6% full (10.3MB)	bit lookup	14.0
	hashing	8.0
	FIB lookup	0.7
	default rate	30%
4.0% full (25.7MB)	bit lookup	12.5
	hashing	7.4
	FIB lookup	0.7
	default rate	22%

defaults is reflected in the increased number of bit lookups and hash computations.

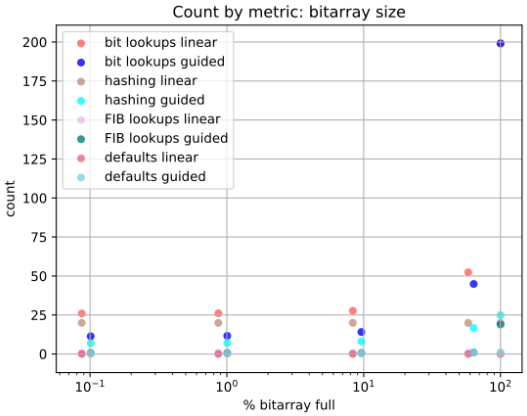


Figure 5: IPv4 search performance for guided vs. linear search; $n = 749362$ elements, $k = 10$ hash funcs

The number of bit lookups asymptotically approaches approximately 8. For the IPv4 table covering approximately 67% of the address space, 33% of lookups will require 4 lookups to arrive at the default route, while the remaining 67% of lookups will traverse the full height of the balanced tree (5 lookups), in addition to decoding the 5-bit *best matching prefix* sequence for a total of 10 lookups. Similarly, assuming the bit vector is sparse enough to never default

to linear search, there would be about 5.3 hash lookups in the limit. Four hash computations are required to yield the default route for 33% of lookups and 6 hash computations are required for the remaining 67% of lookups: 5 hashes to traverse the height of the tree, in addition to a single hash computation to decode the 5-bit sequence). This calculation also illustrates, why $k = 10$ hash functions were chosen for the experiments.

Last but not least, the proposed search algorithm is particularly effective for the IPv6 address space. This is to be expected with the 128-bit IP addresses that require only six bitarray lookups to confirm the default route with the guided search scheme (Figure 6).

Figure 7 compares the performance for two traffic patterns. At one extreme, there is the default-only traffic that is optimally suited for the guided search scheme. At the other extreme, there is the case of no-default traffic directly correlated with the share of each prefix length (here we reuse prefixes as the traffic). It will be seen that for any traffic pattern the guided scheme outperforms the linear search by an order of magnitude.

References

[1] J. Kurose and K. Ross, *Computer Networking: A Top-Down Approach*, 6th ed. Addison Wesley, 2013.

[2] M. A. Ruiz-Sanchez, E. W. Biersack, and W. Dabbous, "Survey and taxonomy of ip address lookup algorithms," *IEEE Network Magazine*, vol. 15, no. 2, pp. 8–23, March 2001.

[3] G. Varghese, *Network Algorithmics*. Morgan Kaufmann, 2007.

[4] M. Waldvogel, G. Varghese, J. Turner, and B. Plattner, "Scalable high-speed ip routing lookups," *Proc. ACM SIGCOMM '97 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, pp. 25–38, October 1997.

[5] S. Dharmapurikar, P. Krishnamurthy, and D. Taylor, "Longest prefix matching using bloom filters," *IEEE/ACM Transactions on Networking*, vol. 14, no. 2, pp. 397–409, April 2006.

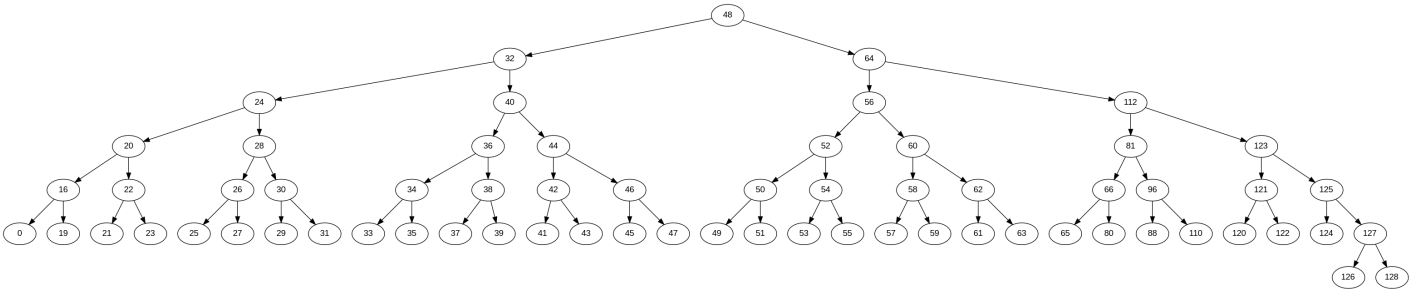
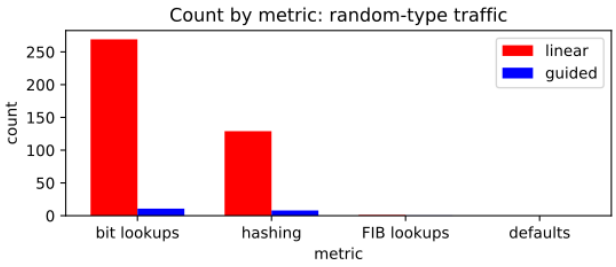


Figure 6: Binary search tree used for the IPv6 prefix search

method	bit lookups	hashing	FIB lookups	defaults
linear	269.2	129.0	1.1	0.0
guided	10.8	8.0	0.2	0.1



method	bit lookups	hashing	FIB lookups	defaults
linear	176.9	84.3	1.1	0.0
guided	19.2	8.5	1.0	0.1

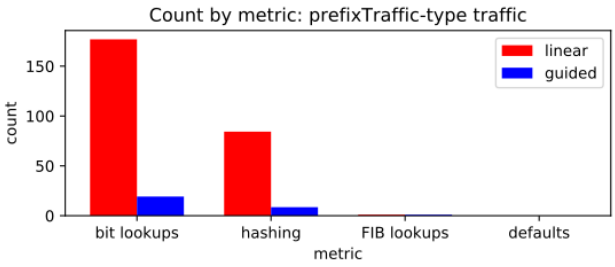


Figure 7: IPv6 search performance: random traffic that always defaults to 0-prefix length vs. traffic correlated with the share of each prefix length (no defaults); guided BF: $n = 53913$ elements, $k = 14$ hash funcs, $size = 2.05$MB, $length = 17238428$ bits, 4.4% full