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*—TBD...**

1. TODO

Factor out some repeated routines from the algorithm(s) – e.g. the encoding of the (length, prefix) pair in preparation for searching in BF.

2. Background

TBD...

3. Related Work

TBD...

4. Solution

4.1. 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. [1] for parallel lookup implemented in hardware. By contrast, the software implementations on conventional hardware pay a hefty penalty – in computation cost and code complexity – to parallelize the lookup. 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.

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 – 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 suspect that the method we propose may be broadly applicable to any key-value store application that

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
```

(a) is defined as a many-to-few kind of mapping, 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.

4.2. 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.

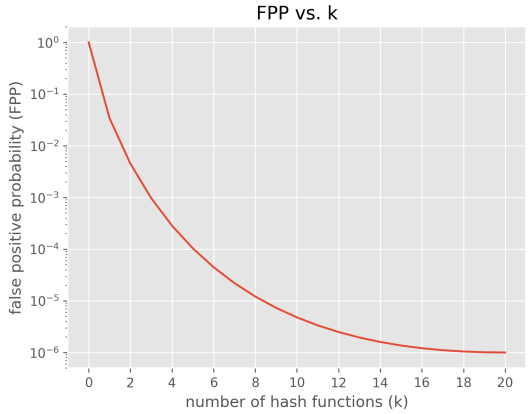

Figure 1: False positive probability vs. number of hash functions in an *optimal* BF

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 traversal 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 *lookup* 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 ($n = 5$ for IPv4, $n = 6$ for IPv6) 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 *lookup* procedures assume an (optimal) binary search tree to guide any search. Such an optimal tree could be constructed for a given router if the historic traffic data and its correlation with the fraction of 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* and the *lookup* procedures are mutually recursive in that the *build* invokes the *lookup* to identify the best matching (shorter) prefix in a BF constructed to date for a (longer) prefix about to be inserted. Therefore, we will first sort the prefixes, before inserting them into the BF in the ascending order.

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*;

Algorithm 2 Build a BF to enable guided search for LMP

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

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 lookup 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 lookup 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 lookup 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 scan the full height of the binary search tree,

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
```

stopping at a leaf, then jumping from the most recent $\texttt{hash}_1$ match to the *best matching prefix*, occasionally defaulting to linear search. 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.

5. Experiments and Discussion

TBD...

References

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

[2] M. Waldvogel, G.Varghese, J. Turner, B. Plattner, *Scalable high-speed prefix matching*, ACM Trans. Comput. Syst., vol. 19, no. 4, pp. 440–482, 2001.