Implementing Poptrie in Lua and DynASM

Max Rottenkolber <max@mr.gy>

Tuesday, 26 March 2019

When routing IP packets it is common to do a longest prefix match (LPM) lookup on the destination address to find a suitable next hop for a packet. Networks are denoted by address prefixes which are matched against the address fields of packets in order to determine where they are headed or where they come from.

Poptrie (http://conferences.sigcomm.org/sigcomm/2015/pdf/papers/p57.pdf) is a high-performance LPM lookup algorithm designed for general purpose computers. What makes it perform well on regular CPUs you ask? Well, it lays out the look up structure in a way that minimizes memory accesses, which are notoriously slow. If we are to route packets at high speeds in software we need to keep our business in the *muffer* (https://twitter.com/rygorous/status/1100600336958382081)—otherwise known as the cache.

One reason to choose Poptrie is that it is sufficiently general to work on prefixes of arbitrary lengths. Hence, it is suitable for both IPv4 and IPv6 addresses, unlike say *DIR-24-8* (http://tiny-tera.stanford.edu/~nickm/papers/Infocom98_lookup.pdf) which targets prefixes if up to 32 bits exclusively. It would be neat if we can use a single, highly optimized lookup routine for all our routing needs (and possibly other use cases as well).

A rough overview of Poptrie (go read the paper!)

At its heart, Poptrie uses a compressed *trie* (https://en.wikipedia.org/wiki/Trie) structure to hold the *forwarding information base* (FIB), which is networking-speak for a mapping from prefixes (keys) to next hops (values). In this compressed trie each node holds edges for the next $k = 6$ bits of the prefix which in turn point to either child nodes or leaves.

Poptrie nodes are efficiently encoded using a trick based on the *popcnt* (https://www.felixcloutier.com/x86/popcnt) instruction which counts the number of bits set in a bit vector. Each node consists of two base offsets—one into an array of nodes, and another into an array of leaves—as well as two 64-bit vectors that hold compressed information about the edges of the node. Let us call them the node and the leaf vectors.

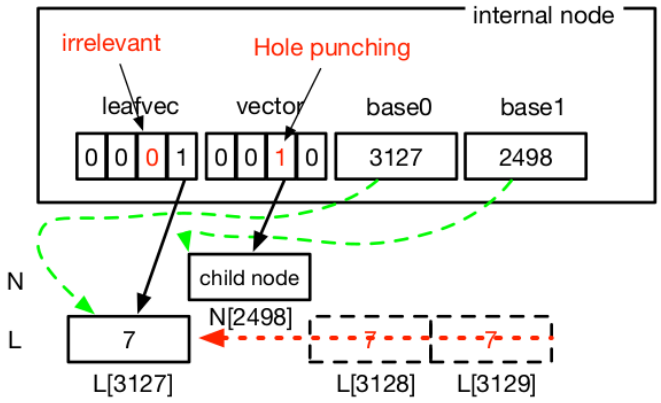
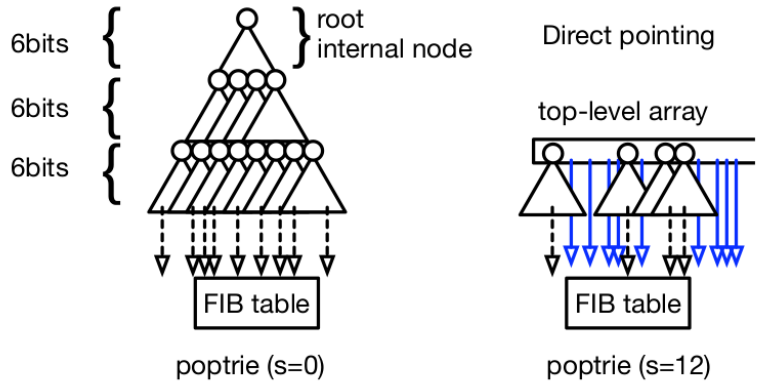


Figure 3 from the Poptrie paper showing the encoding of a node.

Since $k = 6$ bits can represent 64 distinct values, each of the vectors has one bit of information for every possible value of the next k-bit chunk of the prefix. A set bit in the corresponding position of the node vector indicates that the edge points to a node, and the number of bits set up that position indicates the child node's offset from the node base.

While we could just invert the node vector and then count the set bits to get an offset into the leaves array, we expect the trie to be sparse and leaves to be plenty—a "no match" is represented by a leaf with value zero. A node with a single edge would imply 63 additional zero leaves in the leaves array. Enter *leaf compression*. In order minimize the size of the leaves array, the leaf vector has bits at the corresponding positions set if the edge points to a leaf, and only if the leaf's value differs from the value of the preceding leaf. I.e., adjacent identical leaves are deduplicated.



(a) Without Direct Pointing. (b) Direct Pointing.

Figure 5 from the Poptrie paper illustrating the *direct pointing* optimization.

Finally, a *direct pointing* optimization is prepended to the lookup. It trades off memory space for decreased depth of the trie which translates to reduced branching and pointer chasing. This is the same mechanism as is also employed in *DIR-24-8* and *DXR* (http://www.nxlab.fer.hr/dxr/). In this scheme an array of size 2^s holds references to the nodes and leaves for the first s bits of prefixes (the Poptrie paper suggests $s = 18$). In a sense this table of roots allows us to fast forward the lookup to the chunk at s and proceeding with the regular trie search from there.

Structure

Besides the lookup routine there is quite bit of logic required to build the Poptrie lookup structure in the first place. An incremental way of tackling the problem is to first build a simple binary trie to hold the routing information. Following networking-speak (rather inaccurately) our code calls this the RIB (for *routing information base*). After we built

the RIB we can convert it to an equivalent, compressed Poptrie structure, the FIB.

Note that no particular consideration was given towards optimizing either building the RIB or converting it to a FIB because we do not necessarily have to perform these operations in the data-plane. Instead, we make sure that the final FIB that is used for lookups is easily serializable and hence portable. I.e., our Poptrie object constructor can be used to create an empty RIB left for us to populate and convert to a FIB, but we can also do that in a separate process and instantiate it with the resulting binary blobs (node and leaves arrays, and the direct map).

Currently, *lib.poptrie* (https://github.com/eugeneia/snabb/blob/poptrie/src/lib/poptrie.lua) has the following interface:

- `new`(*init*) — Instantiate a new Poptrie object, either empty or from a pre-built FIB. Additionally, *init* is a map that allows you to tweak things like *s* or whether you want to enable the *direct pointing* optimization at all.
- `Poptrie:add`(*key*, *length*, *value*) — Add prefix *key* (a `uint8_t *`) of *length* to the RIB, mapping it to *value* (a `uint16_t` with zero being reserved to indicate "no match")
- `Poptrie:build`() — Derive the FIB from the RIB (overwrite the previous FIB)
- `Poptrie:lookup32/64/128`(*key*) — Look up *key* and return the associated value. These routines expect 32-bit, 64-bit, and 128-bit keys respectively.

There is currently no interface to remove individual prefixes or even clear the RIB, although there is no particular reason that prevents this from being implemented.

```
-- Add key/value pair to RIB (intermediary binary trie)
-- key=uint8_t[?], length=uint16_t, value=uint16_t
function Poptrie:add (key, length, value)
   assert(value)
   local function add (node, offset)
      if offset == length then
         node.value = value
      elseif extract(key, offset, 1) == 0 then
         node.left = add(node.left or {}, offset + 1)
      elseif extract(key, offset, 1) == 1 then
         node.right = add(node.right or {}, offset + 1)
      else error("invalid state") end
      return node
   end
   self.rib = add(self.rib or {}, 0)
end
```

The recurring theme of the code is bootstrapping from "simple, slow, and obviously (ha!) correct" to "more complex but fast". For instance we start with an innocent Lua implementation of a binary trie, and transition to tricky but safe Lua code that builds the compressed Poptrie structure. I am not going to go into that hairy piece of code here, but recommend to checkout the `Poptrie:build_node` function for those that are interested. Let's just say I am happy I was able to do this in a high-level language.

```
-- Map f over keys of length in RIB
function Poptrie:rib_map (f, length, root)
   local function map (node, offset, key, value)
      value = (node and node.value) or value
      local left, right = node and node.left, node and node.right
      if offset == length then
         f(key, value, (left or right) and node)
      else
         map(left, offset + 1, key, value)
         map(right, offset + 1, bor(key, lshift(1, offset)), value)
      end
   end
   return map(root or self.rib, 0, 0)
end
```

Nodes and leaves are allocated on a contiguous chunk of memory. When a chunk is exhausted, a new chunk twice the size of the previous

chunk is allocated. The old chunk is copied into the new chunk, and the next node or leaf is allocated in the new extra space. The old chunk becomes garbage and is eventually collected. This works because all references to leaves or nodes are relative to the pointer to their backing chunk, so you can just swap out chunks by updating the pointers to the nodes and leaves arrays respectively.

```
function Poptrie:allocate_node ()
   if self.direct_pointing then
      -- When using direct_pointing, the node index space is split into half in
      -- favor of a bit used for disambiguation in Poptrie:build_directmap.
      assert(band(self.node_base, Poptrie.leaf_tag) == 0, "Node overflow")
   end
   while self.node_base >= self.num_nodes do
      self:grow_nodes()
   end
   self.node_base = self.node_base + 1
   return self.node_base - 1
end
```

It remains visible in the code that it was constructed incrementally. At first `lib.poptrie` did not implement any of the optimizations described in the paper, which where then added one after another. The leftovers of this scaffolding are toggles that you can switch off to access intermediary versions of the algorithm.

```
-- Compress RIB into Poptrie
function Poptrie:build ()
   self:clear_fib()
   if self.direct_pointing then
      self:build_directmap(self.rib)
   else
      self:build_node(self.rib, self:allocate_node())
   end
end
```

Finally, we generate hand-optimized assembler routines for looking up prefixes of various sizes using DynASM. More on that in the next section.

```
local asm_cache = {}

function Poptrie:configure_lookup ()
   local config = ("leaf_compression=%s,direct_pointing=%s,s=%s")
      :format(self.leaf_compression, self.direct_pointing, self.s)
   if not asm_cache[config] then
      asm_cache[config] = {
         poptrie_lookup.generate(self, 32),
         poptrie_lookup.generate(self, 64),
         poptrie_lookup.generate(self, 128)
      }
   end
   self.asm_lookup32, self.asm_lookup64, self.asm_lookup128 =
      unpack(asm_cache[config])
end
```

Credit where credit is due: a lot of the design choices were influenced by Pete Bristow's earlier work on *LPM implementations* (https://github.com/snabbco/snabb/tree/v2018.09/src/lib/lpm) for *Snabb* (https://github.com/snabbco/snabb). Notably, the requirement that the compressed FIB is easily copied between processes was inherited from his design. Likewise, the idea of bootstrapping from a simple trie was apparent from looking at the prior LPM implementations.

Lookup

There are three flavors of lookup routines. First there is a simple `Poptrie:rib_lookup` routine to look up prefixes in the RIB. It is a dead simple recursive binary trie search, and useful as a reference since the Poptrie lookup routines should return the exact same results when querying the compressed FIB.

Let us quickly recap how the Poptrie lookup works:

- **1.** If we use *direct pointing* look up the first s bits in the direct table. Check if the entry denotes a leaf by checking if bit 32 (the leaf tag) of the value is set. If the entry is a leaf then return early with its value, which is obtained by masking off the leaf tag. Otherwise start the lookup at the node denoted by the entry.
- **2.** Check if the edge denoted by the next k bit segment of the prefix points to a node. If so fetch the next node and repeat this step with the child node and the next prefix segment.

- **3.** Fetch the leaf denoted by the *k* bit segment and return its value.

Our first implementation is written in Lua. It is not particularly fast, but supports all combinations of optimizations turned on or off and works on prefixes of any length. It is mainly used as a runable reference for comparison with the x86 assembly implementation.

```
|.define leaves,  rdi -- pointer to leaves array
|.define nodes,   rsi -- pointer to nodes array
|.define key,     rdx -- key to look up
|.define dmap,    rcx -- pointer to directmap
|.define index,   r8d -- index into node array
|.define node,    r8  -- pointer into node array
|.define offset,  r9d -- offset into key
|.define offsetx, r9  -- (offset as qword)
|.define v,       r10 -- k or s bits extracted from key
|.define v_dw,    r10d -- (v as dword)
|.define vec,     r11 -- 64-bit vector or leafvec
```

The *DynASM generator* (https://github.com/eugeneia/snabb/blob/poptrie/src/lib/poptrie_lookup.dasl) for the optimized x86 lookup routines on the other hand is supposed to generate the best code possible, and keeps its working data set in registers. *Pete Cawley* (https://www.corsix.org/) helped a lot by reviewing the first version of the code and suggesting a bunch of x86 tricks that justify calling the resulting code "optimized". :-)

Note that to use the optimized lookup routines you must ensure that RIB and FIB contain no prefixes longer than the keys you will look up. I.e., you would call `Poptrie:lookup32` on a FIB with prefixes of length less than or equal to 32 bits.

```
...
-- v = extract(key, offset, k=6)
if keysize == 32 then
   if BMI2 then
      | shrx v_dw, dword [key], offset
   else
      | mov ecx, offset
      | mov v_dw, dword [key]
      | shr v, cl
   end
elseif keysize == 64 then
   if BMI2 then
      | shrx v, [key], offsetx
   else
      | mov ecx, offset
      | mov v, [key]
      | shr v, cl
   end
elseif keysize == 128 then
   | mov ecx, offset
   | mov v, [key]
   | mov vec, [key+8]
   | test cl, 64
   | cmovnz v, vec
   | shrd v, vec, cl
else error("NYI") end
...
```

The above excerpt shows how the generator is parameterized on prefix/key size as well as on the availability of certain new x86 features such as *BMI2* (https://en.wikipedia.org/wiki/Bit_Manipulation_Instruction_Sets#BMI2_(Bit_Manipulation_Instruction_Set_2)). That is, we assemble different routines for each key size, and different code depending on which features the executing CPU supports at runtime. This `ifdef`-esque code is certainly not pretty to look at, but we would rather not leave any performance on the table here.

Property testing

Hopefully the routines that build the initial binary trie are so simple as to obviously have no bugs. Assuming we can verify `Poptrie:add` and

`Poptrie:rib_lookup` by means of unit tests and staring at them, we can property test our tower from there.

We randomly generate RIBs and then make sure that our Poptrie lookup routines return identical results when querying the corresponding FIB. The Lua version of the Poptrie lookup is useful to compare with the assembly routines. If both return identical, incorrect results then there is most likely a bug in the FIB building logic. If only one of them is incorrect then the issue is probably with either implementation.

```
local seed = lib.getenv("SNABB_RANDOM_SEED") or 0
for keysize = 1, 128 do
   print("keysize:", keysize)
   -- ramp up the geometry below to crank up test coverage
   for entries = 1, 3 do
      for i = 1, 10 do
         -- add {direct_pointing=true} to test direct pointing
         for _, config in ipairs{ {} } do
            ... -- assert that FIB lookup = RIB lookup
```

While the testing logic is quite simple (e.g., no test case reduction) we are happy to realize that we apparently were able to find all bugs with RIBs of just up to three entries and minimal prefix lengths (1, 7, 33, 65) to trigger edge cases. Since we are checking the simplest cases first and only iteratively increase test case complexity we end up with minimal examples to reproduce bugs. Neat!

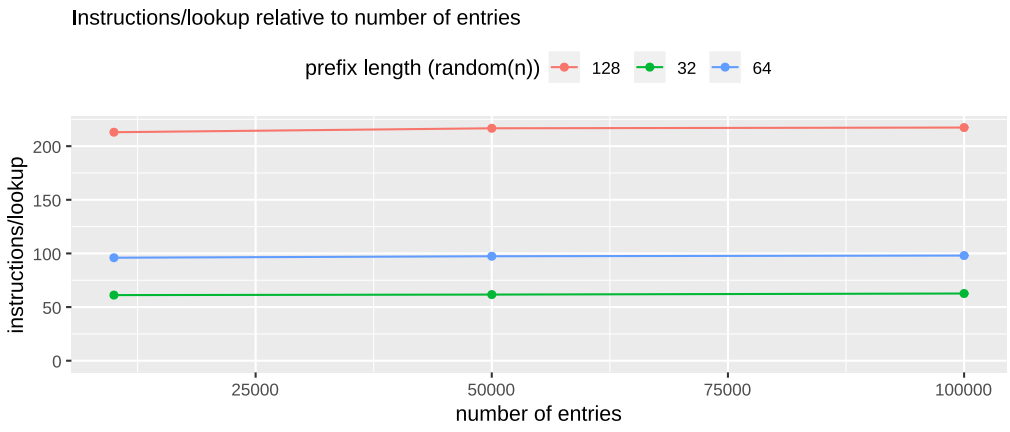
Benchmarking

The library also includes a micro-benchmark to help quantify changes and optimizations. To help quantify micro-optimizations in the x86 code it appears most useful to ensure the benchmark executes in L3 cache in order to highlight how the code performs on the CPU pipeline. Cache misses and the ensuing memory latency mask pressure on the pipeline and could cause us to miss valuable optimizations.

```
PMU analysis (numentries=10000, numhit=100, keysize=32)
build: 0.2241 seconds
lookup: 7922.54 cycles/lookup 17583.86 instructions/lookup
lookup32: 61.34 cycles/lookup 99.96 instructions/lookup
lookup64: 60.50 cycles/lookup 99.95 instructions/lookup
lookup128: 70.93 cycles/lookup 118.62 instructions/lookup
build(direct_pointing): 0.2007 seconds
lookup(direct_pointing): 1281.85 cycles/lookup 3033.97 instructions/lookup
lookup32(direct_pointing): 33.12 cycles/lookup 62.59 instructions/lookup
lookup64(direct_pointing): 33.46 cycles/lookup 62.59 instructions/lookup
lookup128(direct_pointing): 34.72 cycles/lookup 66.80 instructions/lookup
```

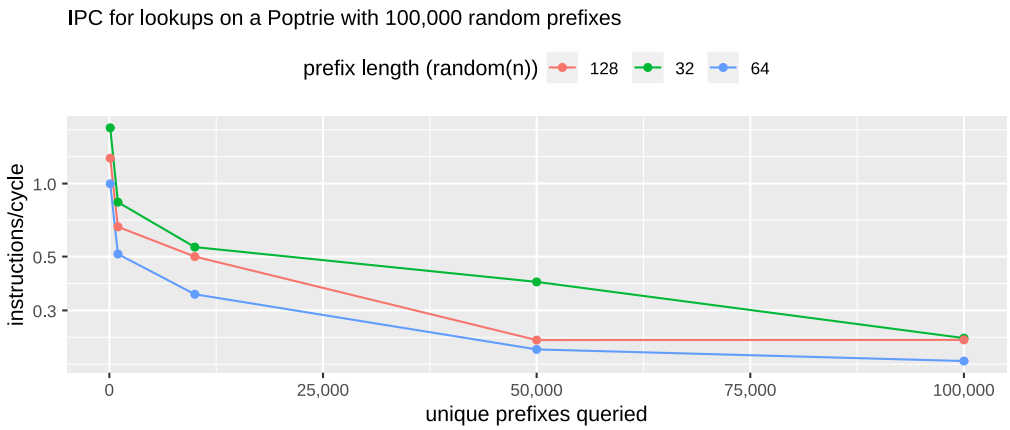
Micro-benchmark results with the default parameters on a laptop. Executes within cache.

The number of instructions needed per lookup increases linearly with prefix length, but is mostly unaffected by the total number of entries in the Poptrie. That is, as long as the set of prefixes you query stay in cache. It seems difficult to make general statements about how memory latency impacts performance because it really depends on the runtime profile of your application when it processes a real world workload.



Plot showing instructions/lookup vs number of Poptrie entries for multiple prefix lengths on a Xeon E3-1246v3.

However, it is clear that memory access latency will become the bottleneck depending on the size of your routing table and traffic profile. You can tweak the `numhit` variable which controls how many distinct entries will be looked up during the benchmark. Doing so shows that the instructions executed per cycle ratio (IPC) takes a dive once the live data outgrows the cache.



Plot showing IPC vs number unique prefixes queried in a Poptrie with 100,000 prefixes on a Xeon E3-1246v3.

Testing in *Vita* (https://github.com/inters/vita) has shown practically equivalent performance when compared to the previously used *DIR-24-8* implementation. This is good news for Vita because we need to support IPv6 routes, and it looks like we can do that with a single algorithm without a performance regression in IPv4 routing. Vita might not be a particularly representative test case for other applications though, since it typically uses very small routing tables.

Next Steps

Luke Gorrie brought up the possibility parallelizing lookups to amortize memory latency. That is, if we could somehow perform multiple lookups in parallel we could do useful work while waiting for the next nodes to be fetched from memory. At least in theory this could help us keep a more steady IPC ratio with random accesses on big routing tables. Maybe someone wants to take a stab at vectorizing the Poptrie lookup routine? :-)