IPv4 route lookup on Linux

Vincent Bernat — June 21, 2017

📌 *TL;DR*

With its implementation of IPv4 routing tables using LPC-tries, Linux offers good lookup performance (50 ns for a full view) and low memory usage (64 MiB for a full view).

During the lifetime of an IPv4 datagram inside the Linux kernel, one important step is the *route lookup for the destination address* through the `fib_lookup()` function. From essential information about the datagram (source and destination IP addresses, interfaces, firewall mark, …), this function should quickly provide a decision. Some possible options are:

- local delivery (`RTN_LOCAL`),
- forwarding to a supplied next hop (`RTN_UNICAST`),
- silent discard (`RTN_BLACKHOLE`).

Since 2.6.39, Linux stores routes in a *compressed prefix tree* (commit 3630b7c050d9). In the past, a route cache was maintained but it has been removed[1] in Linux 3.6.

Route lookup in a trie

Looking up a route in a routing table is to find the *most specific prefix* matching the requested destination. Let's assume the following routing table:

```
$ ip route show scope global table 100
default via 203.0.113.5 dev out2
192.0.2.0/25
        nexthop via 203.0.113.7  dev out3 weight 1
        nexthop via 203.0.113.9  dev out4 weight 1
192.0.2.47 via 203.0.113.3 dev out1
192.0.2.48 via 203.0.113.3 dev out1
192.0.2.49 via 203.0.113.3 dev out1
192.0.2.50 via 203.0.113.3 dev out1
```

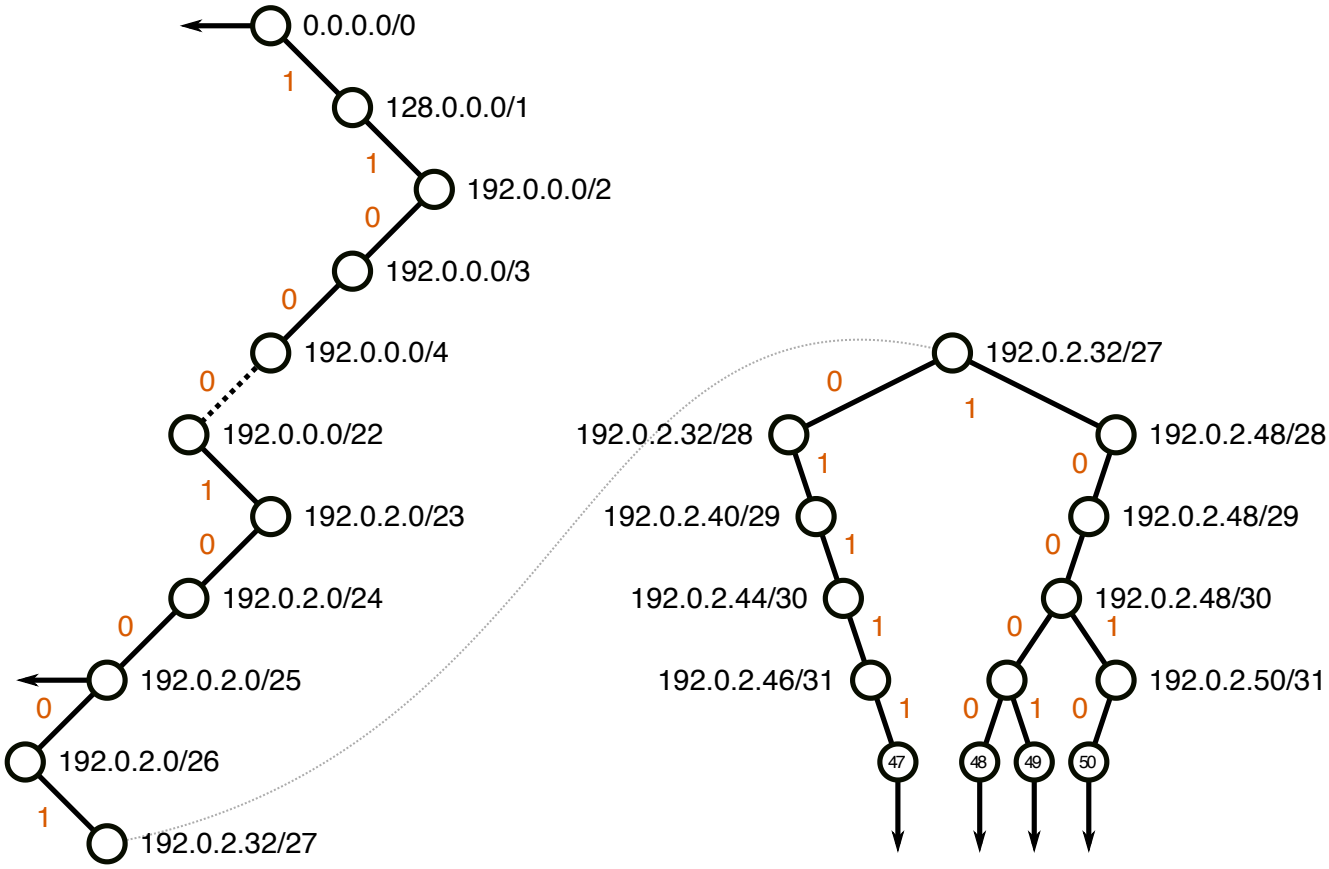
Here are some examples of lookups and the associated results:

Destination IP	Next hop
`192.0.2.49`	`203.0.113.3` via `out1`
`192.0.2.50`	`203.0.113.3` via `out1`
`192.0.2.51`	`203.0.113.7` via `out3` or `203.0.113.9` via `out4` (ECMP)
`192.0.2.200`	`203.0.113.5` via `out2`

A common structure for route lookup is the trie, a tree structure where each node has its parent as prefix.

Lookup with a simple trie

The following trie encodes the previous routing table:



Simple routing trie for a small routing table. For readibility, the trie has been cut in two parts. The nodes with an arrow contain an actual route entry.

For each node, the prefix is known by its path from the root node and the prefix length is the current depth.

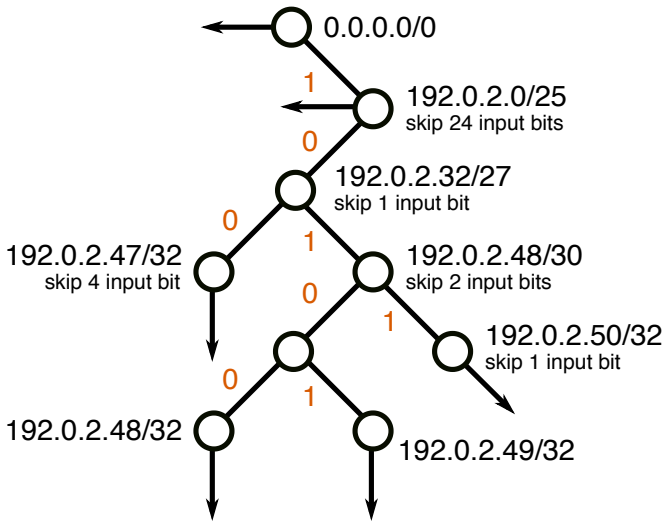
A lookup in such a trie is quite simple: at each step, fetch the n^{th} bit of the IP address, where n is the current depth. If it is 0, continue with the first child. Otherwise, continue with the second. If a child is missing, backtrack until a routing entry is found. For example, when looking for `192.0.2.50`, we will find the result in the corresponding leaf (at depth 32). However for `192.0.2.51`, we will reach `192.0.2.50/31` but there is no second child. Therefore, we backtrack until the `192.0.2.0/25` routing entry.

Adding and removing routes is quite easy. From a performance point of view, the lookup is done in constant time relative to the number of routes (due to maximum depth being capped to 32).

Quagga is an example of routing software still using this simple approach.

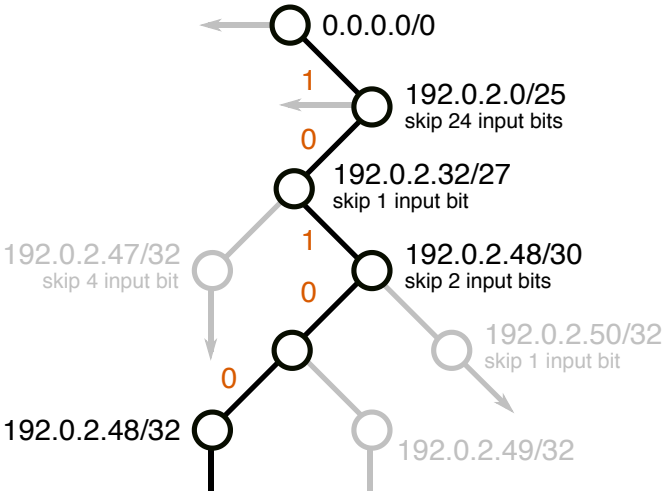
Lookup with a path-compressed trie

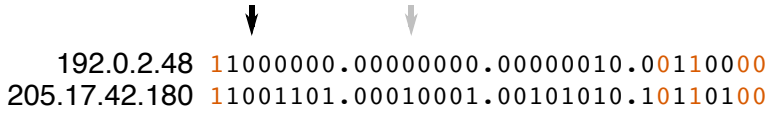
In the previous example, most nodes only have one child. This leads to a lot of unneeded bitwise comparisons and memory is also wasted on many nodes. To overcome this problem, we can use *path compression*: each node with only one child is removed (except if it also contains a routing entry). Each remaining node gets a new property telling how many input bits should be skipped. Such a trie is also known as a *Patricia trie* or a radix tree. Here is the path-compressed version of the previous trie:



Patricia trie for a small routing table. Some nodes indicate how many additional input bits to skip.

Since some bits have been ignored, on a match, a final check is executed to ensure all bits from the found entry are matching the input IP address. If not, we must act as if the entry wasn't found (and backtrack to find a matching prefix). The following figure shows two IP addresses matching the same leaf:





Lookup of two IP addresses in a Patricia trie. Both IP share the same checked bits but differ on the skipped ones.

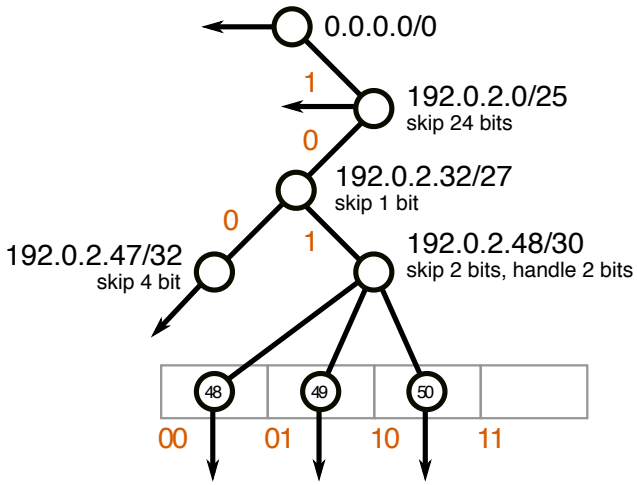
The reduction on the average depth of the tree compensates the necessity to handle these false positives. The insertion and deletion of a routing entry is still easy enough.

Many routing systems are using Patricia trees:

- OpenBSD
- NetBSD
- FreeBSD
- GoBGP (through go-radix)
- Linux for IPv6, see "IPv6 route lookup on Linux"

Lookup with a level-compressed trie

In addition to path compression, *level compression*[2] detects parts of the trie that are *densily populated* and replace them with a single node and an associated vector of 2^k children. This node will handle k input bits instead of just one. For example, here is a level-compressed version our previous trie:



LPC trie. Two levels are merged into a single one using a 4-element vector. The last element of this vector is empty.

Such a trie is called LC-trie or *LPC-trie* and offers higher lookup performances compared to a radix tree.

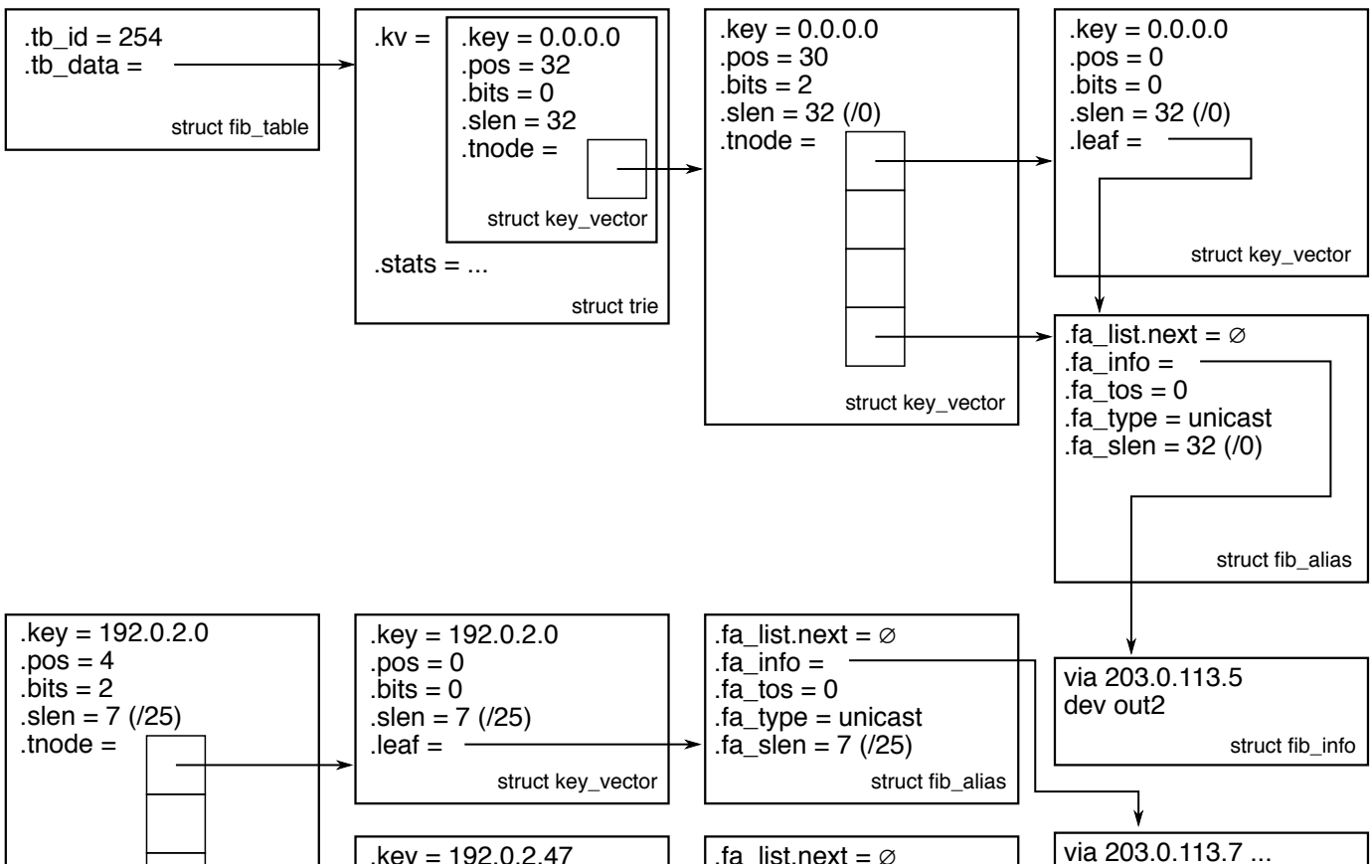
An heuristic is used to decide how many bits a node should handle. On Linux, if the ratio of non-empty children to all children would be above 50% when the node handles an additional bit, the node gets this additional bit. On the other hand, if the current ratio is below 25%, the node loses the responsibility of one bit. These values are not tunable.

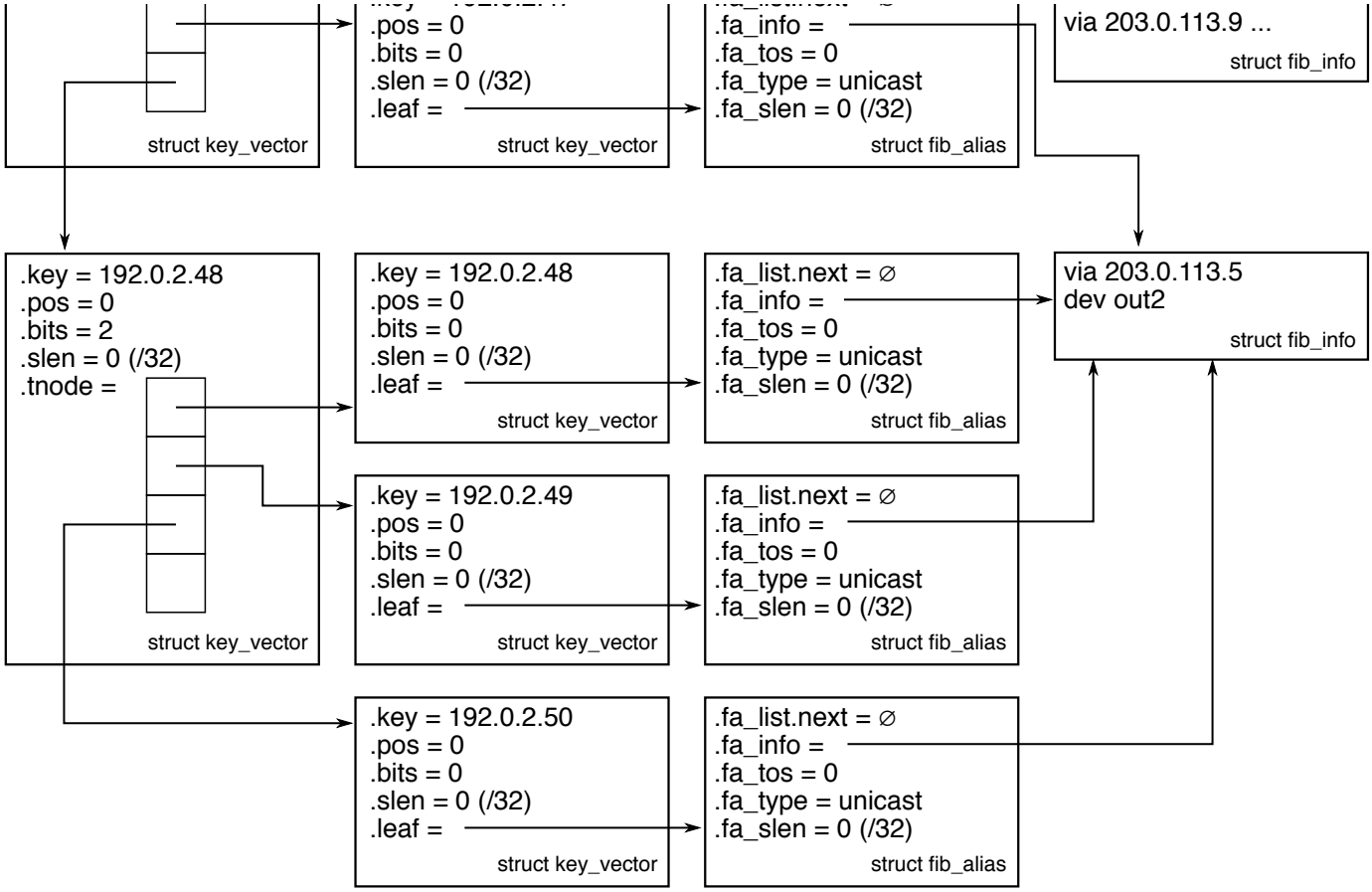
Insertion and deletion becomes more complex but lookup times are also improved.

Implementation in Linux

The implementation for IPv4 in Linux exists since 2.6.13 (commit 19baf839ff4a) and is enabled by default since 2.6.39 (commit 3630b7c050d9).

Here is the representation of our example routing table in memory:[3]





In-memory representation of an LPC trie in Linux

There are several structures involved:

- `struct fib_table` represents a routing table,
- `struct trie` represents a complete trie,
- `struct key_vector` represents either an internal node (when `bits` is not zero) or a leaf,
- `struct fib_info` contains the characteristics shared by several routes (like a next-hop gateway and an output interface),
- `struct fib_alias` is the glue between the leaves and the `fib_info` structures.

The trie can be retrieved through `/proc/net/fib_trie`:

```
$ cat /proc/net/fib_trie
Id 100:
  +-- 0.0.0.0/0 2 0 2
     |-- 0.0.0.0
        /0 universe UNICAST
     +-- 192.0.2.0/26 2 0 1
        |-- 192.0.2.0
           /25 universe UNICAST
```

```
        |-- 192.0.2.47
           /32 universe UNICAST
        +-- 192.0.2.48/30 2 0 1
           |-- 192.0.2.48
              /32 universe UNICAST
           |-- 192.0.2.49
              /32 universe UNICAST
           |-- 192.0.2.50
              /32 universe UNICAST
[...]
```

For internal nodes, the numbers after the prefix are:

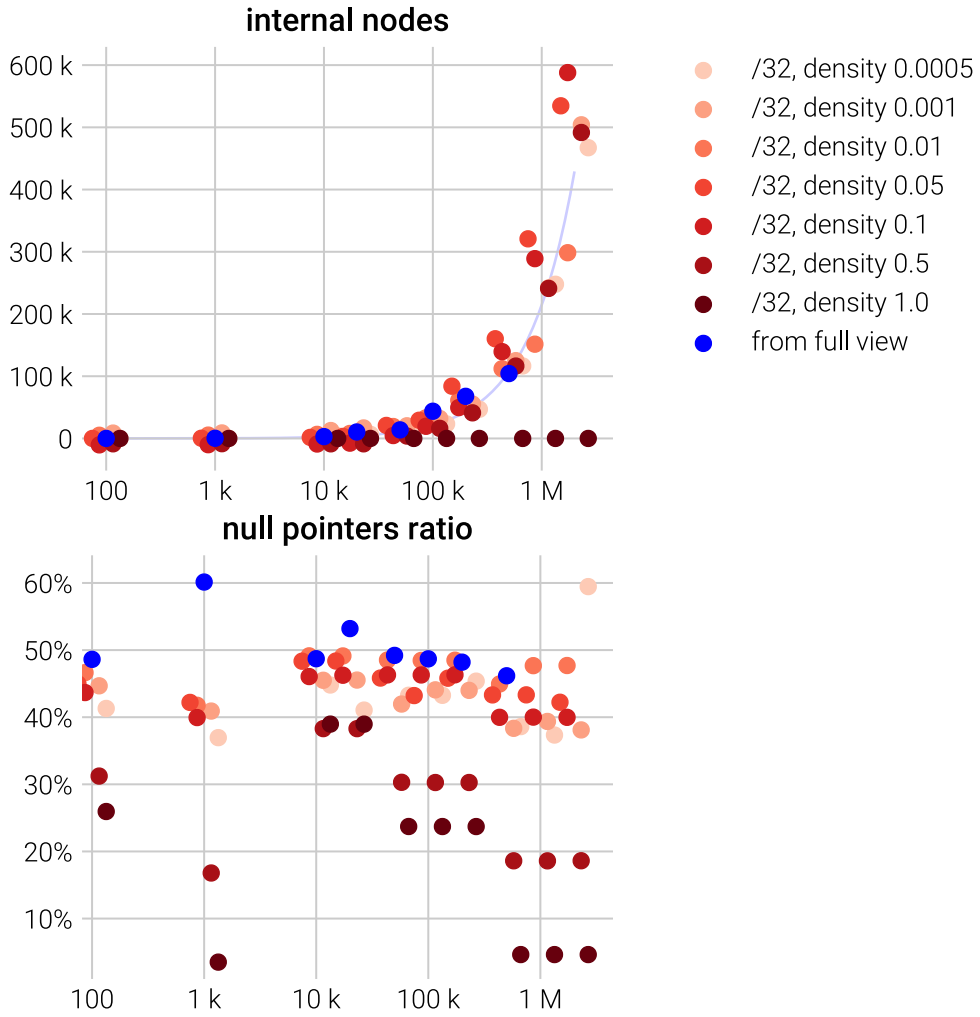
1. the number of bits handled by the node,
2. the number of full children (they only handle one bit),
3. the number of empty children.

Moreover, if the kernel was compiled with `CONFIG_IP_FIB_TRIE_STATS`, some interesting statistics are available in `/proc/net/fib_triestat`:[4]

```
$ cat /proc/net/fib_triestat
Basic info: size of leaf: 48 bytes, size of tnode: 40 bytes.
Id 100:
        Aver depth:     2.33
        Max depth:      3
        Leaves:         6
        Prefixes:       6
        Internal nodes: 3
          2: 3
        Pointers: 12
Null ptrs: 4
Total size: 1  kB
[...]
```

When a routing table is very dense, a node can handle many bits. For example, a densily populated routing table with 1 million entries packed in a /12 can have one internal node handling 20 bits. In this case, route lookup is essentially reduced to a lookup in a vector.

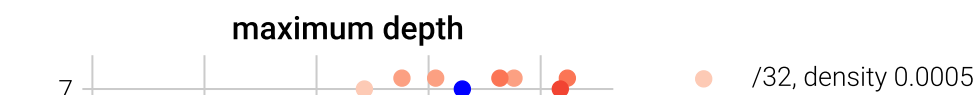
The following graph shows the number of internal nodes used relative to the number of routes for different scenarios (routes extracted from an Internet full view, /32 routes spreaded over 4 different subnets with various densities). When routes are densily packed, the number of internal nodes are quite limited.

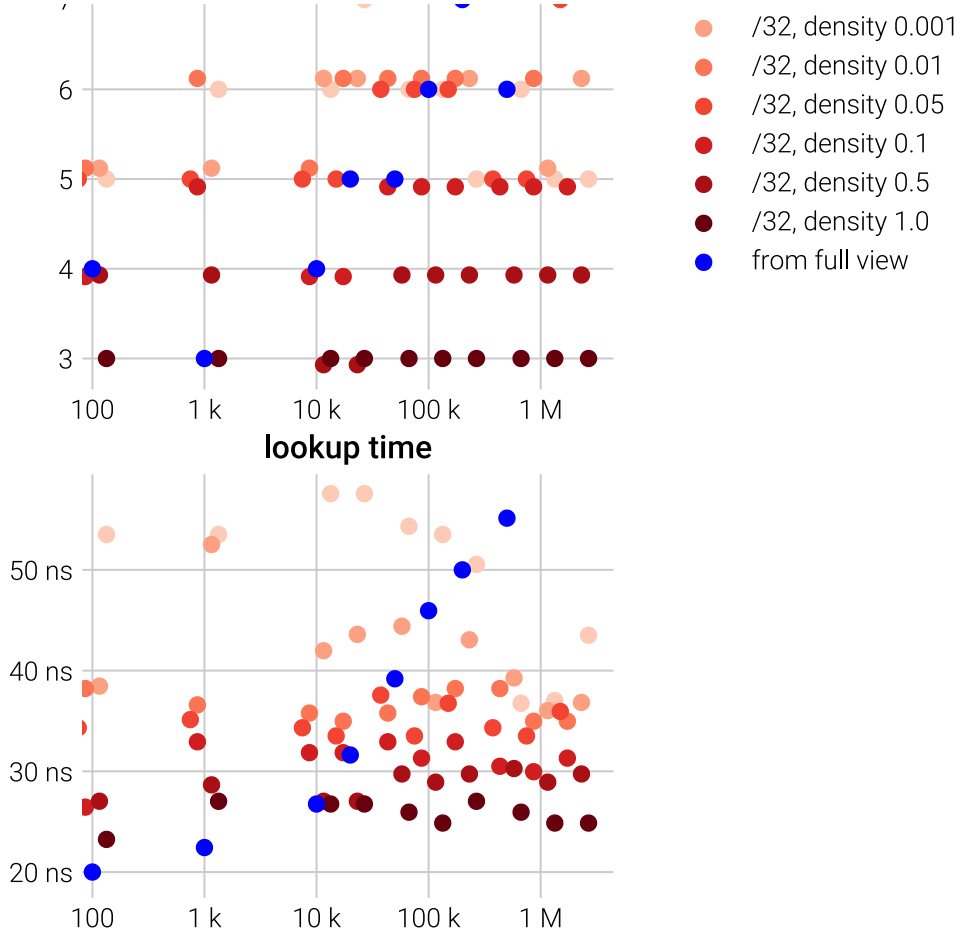


Number of internal nodes and null pointers depending on the number of routes. The x-axis scale is logarithmic. The blue line is a linear regression.

Performance

So how performant is a route lookup? The maximum depth stays low (about 6 for a full view), so a lookup should be quite fast. With the help of a small kernel module, we can accurately benchmark[5] the `fib_lookup()` function:





Maximum depth and lookup time depending on the number of routes inserted. The x-axis scale is logarithmic.

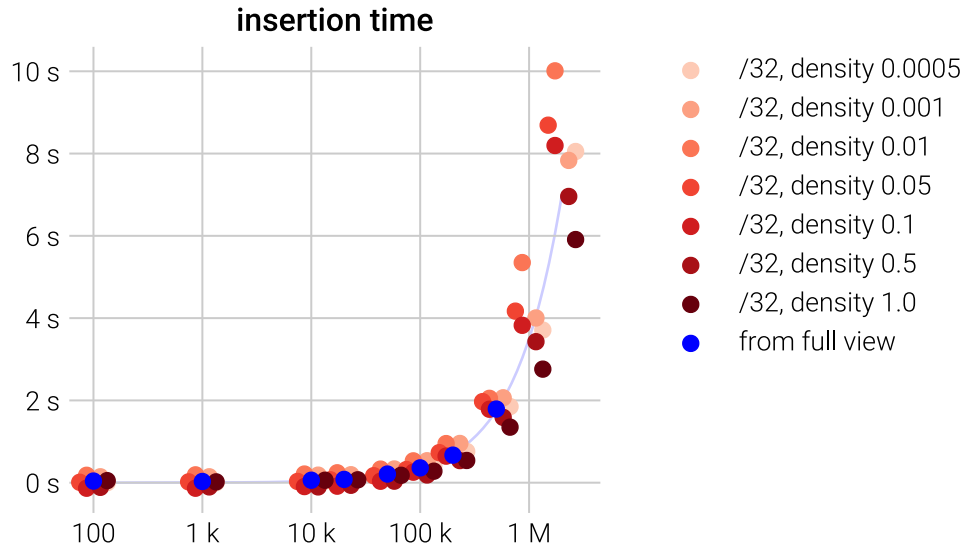
The lookup time is loosely tied to the maximum depth. When the routing table is densily populated, the maximum depth is low and the lookup times are fast.

When forwarding at 10 Gbps, the time budget for a packet would be about 50 ns. Since this is also the time needed for the route lookup alone in some cases, we wouldn't be able to forward at line rate with only one core. Nonetheless, the results are pretty good and they are expected to scale linearly with the number of cores.

The measurements are done with a Linux kernel 4.11 from *Debian unstable*. I have gathered performance metrics accross kernel versions in "Performance progression of IPv4 route lookup on Linux."

Another interesting figure is the time it takes to insert all these routes into the kernel. Linux is also quite efficient in this area since you can insert 2

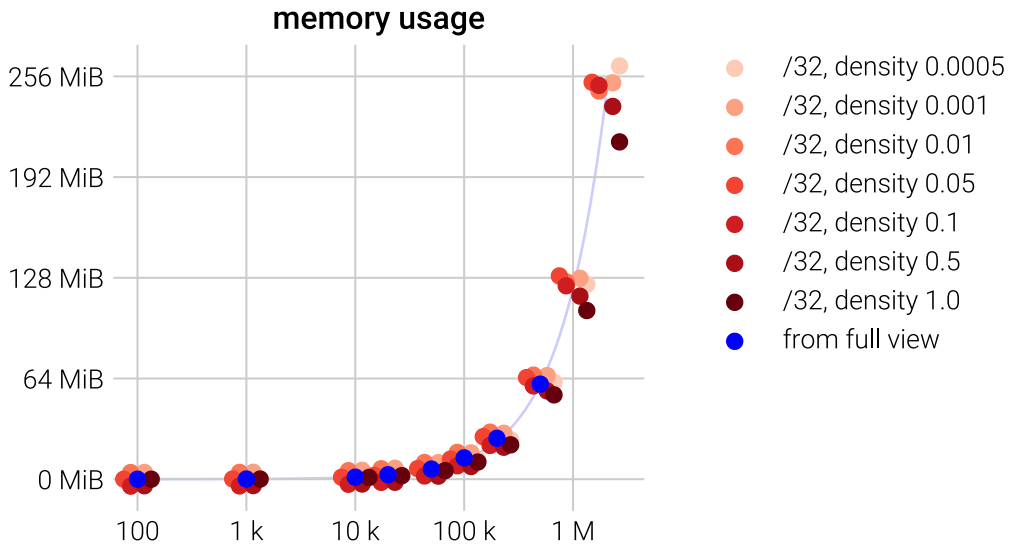
million routes in less than 10 seconds:



Insertion time (system time) for a given number of routes (linear). The x-axis scale is logarithmic. The blue line is a linear regression.

Memory usage

The memory usage is available directly in `/proc/net/fib_triestat`. The statistic provided doesn't account for the `fib_info` structures, but you should only have a handful of them (one for each possible next-hop). As you can see on the graph below, the memory use is linear with the number of routes inserted, whatever the shape of the routes is.



Memory usage of the LPC-trie depending on the number of routes inserted (linear). The x-axis cale is logarithmic. The blue line is a linear regression.

The results are quite good. With only 256 MiB, about 2 million routes can be stored!

Routing rules

Unless configured without `CONFIG_IP_MULTIPLE_TABLES`, *Linux supports several routing tables* and has a system of configurable rules to select the table to use. These rules can be configured with `ip rule`. By default, there are three of them:

```
$ ip rule show
0:      from all lookup local
32766:  from all lookup main
32767:  from all lookup default
```

Linux will first lookup for a match in the `local` table. If it doesn't find one, it will lookup in the `main` table and at last resort, the `default` table.

Builtin tables

The `local` table contains routes for local delivery:

```
$ ip route show table local
broadcast 127.0.0.0 dev lo proto kernel scope link src 127.0.0.1
local 127.0.0.0/8 dev lo proto kernel scope host src 127.0.0.1
local 127.0.0.1 dev lo proto kernel scope host src 127.0.0.1
broadcast 127.255.255.255 dev lo proto kernel scope link src 127.0.0.1
broadcast 192.168.117.0 dev eno1 proto kernel scope link src 192.168.117.
55
local 192.168.117.55 dev eno1 proto kernel scope host src 192.168.117.55
broadcast 192.168.117.63 dev eno1 proto kernel scope link src 192.168.11
7.55
```

This table is populated automatically by the kernel when addresses are configured. Let's look at the three last lines. When the IP address `192.168.117.55` was configured on the `eno1` interface, the kernel automatically added the appropriate routes:

- a route for `192.168.117.55` for local unicast delivery to the IP address,
- a route for `192.168.117.255` for broadcast delivery to the broadcast address,
- a route for `192.168.117.0` for broadcast delivery to the network address.

When `127.0.0.1` was configured on the loopback interface, the same kind of routes were added to the `local` table. However, a loopback address receives a special treatment and the kernel also adds the whole subnet to the `local` table. As a result, you can ping any IP in `127.0.0.0/8`:

```
$ ping -c1 127.42.42.42
PING 127.42.42.42 (127.42.42.42) 56(84) bytes of data.
64 bytes from 127.42.42.42: icmp_seq=1 ttl=64 time=0.039 ms

--- 127.42.42.42 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 0.039/0.039/0.039/0.000 ms
```

The `main` table usually contains all the other routes:

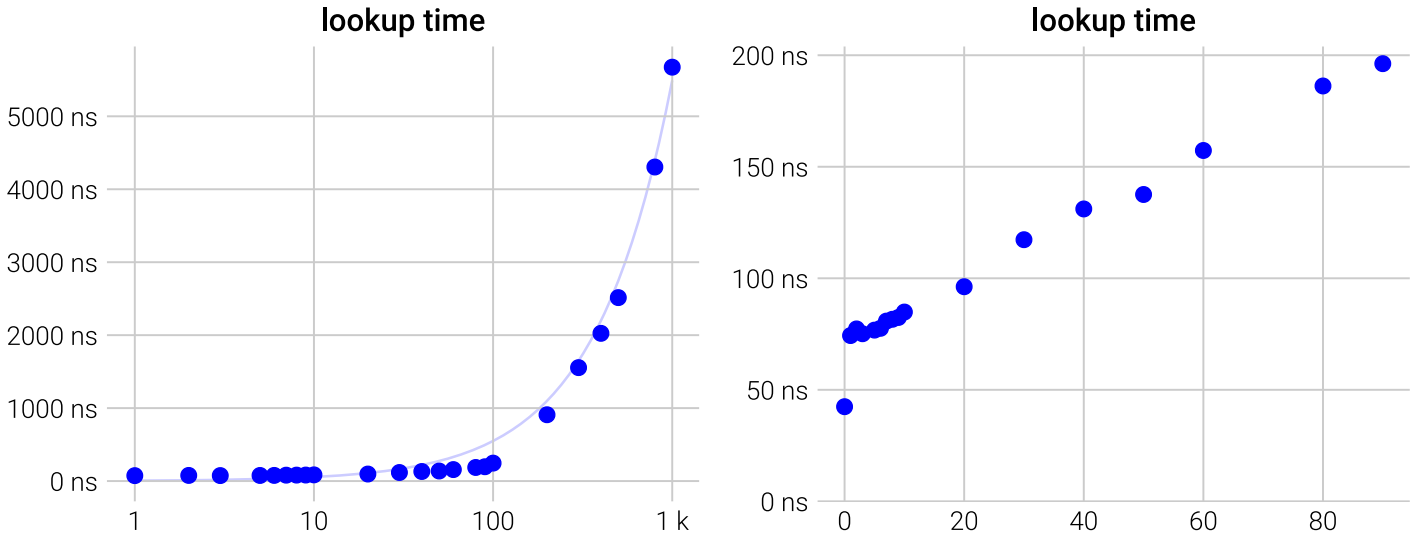
```
$ ip route show table main
default via 192.168.117.1 dev eno1 proto static metric 100
192.168.117.0/26 dev eno1 proto kernel scope link src 192.168.117.55 metr
ic 100
```

The `default` route has been configured by some DHCP daemon. The connected route (`scope link`) has been automatically added by the kernel (`proto kernel`) when configuring an IP address on the `eno1` interface.

The `default` table is empty and has little use. It has been kept when the current incarnation of advanced routing has been introduced in Linux 2.1.68 after a first tentative using "classes" in Linux 2.1.15.[6]

Performance

Since Linux 4.1 (commit 0ddcf43d5d4a), when the set of rules is left unmodified, the `main` and `local` tables are merged and the lookup is done with this single table (and the `default` table if not empty). Moreover, since Linux 3.0 (commit f4530fa574df), without specific rules, there is no performance hit when enabling the support for multiple routing tables. However, as soon as you add new rules, some CPU cycles will be spent for each datagram to evaluate them. Here is a couple of graphs demonstrating the impact of routing rules on lookup times:



Lookup time for a given number of routing rules. Only last routing rule matches. The first graph uses a logarithmic scale while the second uses a linear scale. The blue line is a linear regression.

For some reason, the relation is linear when the number of rules is between 1 and 100 but the slope increases noticeably past this threshold. The second graph highlights the negative impact of the first rule (about 30 ns).

A common use of rules is to create *virtual routers*: interfaces are segregated into domains and when a datagram enters through an interface from domain *A*, it should use routing table *A*:

```
# ip rule add iif vlan457 table 10
# ip rule add iif vlan457 blackhole
# ip rule add iif vlan458 table 20
# ip rule add iif vlan458 blackhole
```

The blackhole rules may be removed if you are sure there is a default route in each routing table. For example, we add a blackhole default with a high metric to not override a regular default route:

```
# ip route add blackhole default metric 9999 table 10
# ip route add blackhole default metric 9999 table 20
# ip rule add iif vlan457 table 10
# ip rule add iif vlan458 table 20
```

To reduce the impact on performance when many interface-specific rules are used, interfaces can be attached to VRF instances and a single rule can be used to select the appropriate table:

```
# ip link add vrf-A type vrf table 10
# ip link set dev vrf-A up
# ip link add vrf-B type vrf table 20
# ip link set dev vrf-B up
# ip link set dev vlan457 master vrf-A
# ip link set dev vlan458 master vrf-B
# ip rule show
0:      from all lookup local
1000:   from all lookup [l3mdev-table]
32766:  from all lookup main
32767:  from all lookup default
```

The special `l3mdev-table` rule was automatically added when configuring the first VRF interface. This rule will select the routing table associated to the VRF owning the input (or output) interface.

VRF was introduced in Linux 4.3 (commit 193125dbd8eb), the performance was greatly enhanced in Linux 4.8 (commit 7889681f4a6c) and the special routing rule was also introduced in Linux 4.8 (commit 96c63fa7393d, commit 1aa6c4f6b8cd). You can find more details about it in the kernel documentation.

Conclusion

The takeaways from this article are:

- route lookup times hardly increase with the number of routes,
- densily packed /32 routes lead to amazingly fast route lookups,
- memory use is low (128 MiB par million routes),
- no optimization is done on routing rules.

For IPv6, have a look at "IPv6 route lookup on Linux."

. . .

1. The routing cache was subject to reasonably easy to launch denial of service attacks. It was also believed to not be efficient for high volume sites like Google but I have first-hand experience it was not the case for moderately high volume sites. ↩

2. "IP-address lookup using LC-tries," IEEE Journal on Selected Areas in Communications, 17(6):1083-1092, June1999. ↩

3. For internal nodes, the `key_vector` structure is embedded into a `tnode` structure. This structure contains information rarely used during lookup, notably the reference to the parent that is usually not needed for backtracking as Linux keeps the nearest candidate in a variable. ↩

4. One leaf can contain several routes (`struct fib_alias` is a list). The number of "prefixes" can therefore be greater than the number of leaves. The system also keeps statistics about the distribution of the internal nodes relative to the number of bits they handle. In our example, all the three internal nodes are handling 2 bits. ↩

5. The measurements are done in a virtual machine with one vCPU. The host is an Intel Core i5-4670K running at 3.7 GHz during the experiment (CPU governor was set to performance). The benchmark is single-threaded. It runs a warm-up phase, then executes about 100,000 timed iterations and keeps the median. Timings of individual runs are computed from the TSC. ↩

6. Fun fact: the documentation of this first tentative of more flexible routing is still available in today's kernel tree and explains the usage of the "default class". ↩