Poptrie based Routing Table Lookup in Linux Kernel

MD Iftakharul Islam, Javed I Khan
Kent State University
mislam4@kent.edu, javed@cs.kent.edu

ABSTRACT

As the speed of network interface is growing rapidly, routing table lookup has become a bottleneck. A 100Gb/s NIC can receive 148.8 million packets per second. This is why Linux kernel needs to able to perform 148.8 million routing table lookup per second. Currently Linux kernel uses LC-trie for routing table lookup which cannot achieve such fast routing table lookup. Recently poptrie [1] has been proposed that can improve the routing table lookup performance to a great extent. However poptrie is implemented as an userspace library. This is why it is only useful to kernel bypass packet processing frameworks such DPDK and VPP. In this paper, we have implemented poptrie based routing table lookup in Linux kernel. This will enable us to use Linux as a high speed router. Our experimental results show that poptrie based routing table lookup improves the lookup cost from 200-850 CPU cycles to 80-100 CPU cycles.

1. INTRODUCTION

Routing table lookup is a key functionality of a router. A router needs to extract the destination IP address of incoming packets and find the destination NIC based on the longest prefix match (LPM) algorithm. For instance, Table 1 shows an example routing table of a router. If the destination IP address of an incoming packet is 169.254.198.1, then according to the longest prefix match algorithm, the packet should be forwarded to $eth4$. Again if the destination IP address of an incoming packet is 169.254.190.5, then the packet should be forwarded to $eth3$. Many of the commercial routers these days use Ternary Content Addressable Memory (TCAM) to implement the LPM algorithm. TCAM based routers however suffer from high cost and high power consumption. It is also not programmable. In modern service provider routers, we need functionalities beyond LPM. Those include network monitoring, traffic shaping, explicit throughput allocation and so on. This is why it is desirable to implement all the network functions in software rather than hardware. The objective of this paper is to implement a high-speed routing table lookup with general purpose computers. Recently we have seen poptrie [1] based routing table lookup which improves the lookup performance significantly. Poptrie however is implemented as an userspace library. This is why Operating Systems cannot utilize it. Currently it is only useful to kernel-bypass packet processing frameworks such as DPDK [2] and VPP [7]. In this paper, we have implemented poptrie based routing table lookup in Linux kernel. This will enable us to use Linux as a high speed router. It is noteworthy that Linux kernel also has an in-kernel dataplane framework XDP [8]. Poptrie based fast routing table lookup along with XDP will enable Linux to work as router without needing any kernel-bypass dataplane framework.

The remaining of the paper proceeds as follows. In Section II, we describe the challenges and previous research works that have been done on routing table lookup. In Section III, we describe how a poptrie is constructed from a routing table. In Section IV, we describe the poptrie based routing table lookup algorithm. Section V discusses the implementation and Section VI shows the experimental results. Finally Section VII concludes the paper.

Table 1: Routing table (also known as FIB table)

Prefix	Destination NIC
10.18.0.0/22	$eth1$
131.123.252.42/32	$eth2$
169.254.0.0/16	$eth3$
169.254.192.0/18	$eth4$
192.168.122.0/24	$eth5$

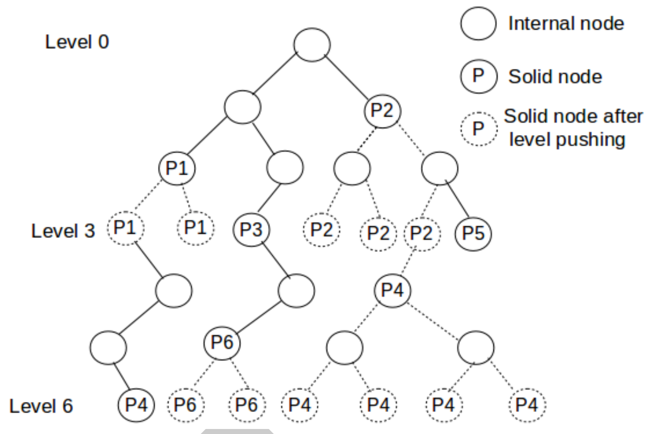
2. CHALLENGES AND RELATED WORKS

As the data rate of network interface is growing rapidly, routing table lookup in a general purpose computer has become a bottleneck. For instance, a 100Gb/s NIC produces 148.8 million packets per second. This is why, a CPU based router needs to perform 148.8 million lookups per second. Again the routing tables are also keep growing. According to RIPE [6] network coordination center, a backbone router may have about 500K routes in their routing tables. Fast routing table lookup in such a large routing table is particularly challenging.

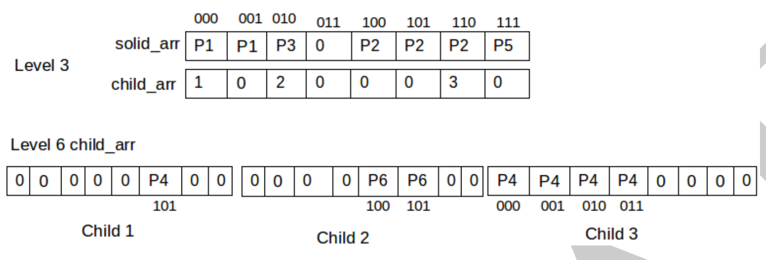
Although performance of general purpose CPU have increased significantly over last decade, DRAM latency remains unchanged at roughly 50 ns. This is why, routing

Prefix (in binary)	Next-hop
00*	P1
1*	P2
010*	P3
1100*	P4
111*	P5
000101*	P4
01010*	P6

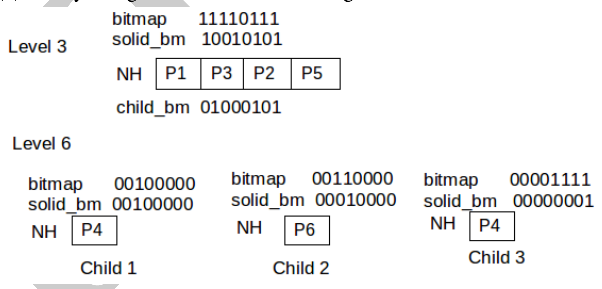
(a) Routing table



(b) Binary tree generated from the routing table



(c) Level 3 and level 6 of the binary tree are represented as arrays



(d) Poptrie constructed from the array

Figure 1: A Poptrie is constructed from routing table

table lookup algorithms need to exploit CPU cache in order to achieve the desired performance. It is required that routing table lookup data structure is small enough to fit in CPU cache. Storing a big routing table (such as one with 500K routes) in CPU cache is particularly challenging. The lookup operation also should require minimal CPU cycle.

Routing table lookup on general purpose computer however is a long standing problem. Linux currently uses LC-trie [5] for routing table lookup. It however exhibits very poor cache behavior for routing table lookup [1, 9]. Cisco routers, on the other hand, use Tree Bitmap [3] for routing table lookup. Tree Bitmap however cannot achieve desired lookup performance needed for 100Gb/s NICs [1, 9]. Recently we have seen several proposal that shows significant improvement in lookup performance. These include DXR [10], SAIL [9] and Poptrie [1]. DXR [10] is range search algorithm utilized by multicore processor. SAIL [9] on the other hand divides the routing table into multiple bitmap-array. Although SAIL often performs very fast lookup, it requires large on-chip memory (around 44 MB) [1]. This is why, it often runs into cache-misses resulting degraded lookup-performance. Poptrie [1] solve this problem by designing a very memory efficient data structure which is small enough to fit in a on-chip memory (CPU-cache). Poptrie also requires very small number of CPU cycles for routing table lookup. However poptrie is designed as an userspace library. This is why it is only useful to kernel-bypass packet processing frameworks succh as DPDK, VPP and so on. In this paper, we have implemented Poptrie in Linux kernel. Our experimental results show that it outperforms LC-trie [5] which is currently being used in Linux kernel.

3. POPTRIE CONSTRUCTION

This section describes how a poptrie is constructed from a routing table. Figure 1a shows a routing table containing prefixes and next-hops. The routing table is then represented as a binary tree as in the Figure 1b. Here all the nodes that have a next hop are called solid nodes. All the nodes that do not have a next hop are called internal nodes. Note that, all the solid nodes of level $0-2$ are pushed to level 3 and all the solid nodes of level $4-5$ are pushed to level 6. This is called *level-pushing* which was initially proposed in SAIL [9]. In this example, we construct the poptrie based on level 3 and level 6. However in actual implementation, we use level 6, 12, 18, 24, 30 and 36.

The nodes in level 3 and level 6 are represented as arrays in Figure 1c. The $solid\_arr$ contains the next-hop and $child\_arr$ contains the child ID for the next level (in this case, level 6). SAIL [9] uses similar data structure for routing table lookup. Here we use $solid\_arr_i$ and $child\_arr_i$ to represent the next-hop and child array of level i. Given an IP address a, we use $a_{(i,j)}$ to denote the integer value of the bit string of a from the i-th bit to the j-th bit. For instance, if $a = 11010..b$, then $a_{(0,2)} = 6$ (the integer value of three left-most bit of the IP address). The lookup process in the arrays in Figure 1c works as follows: for a given

IP address a, $solid\_arr_3[a_{(0,2)}]$ will contain the next-hop for the the IP address. The lookup process also checks if $child\_arr_3[a_{(0,2)}] > 0$; if yes, then the lookup process will continue the same process with the arrays in level 6. Let us assume that the destination IP address of an incoming-packet packet is 01010111...0b. In this case, $a_{(0,2)} = 2$. As a result the look process initially assumes that $solid\_arr_3[2]$ contains the next-hop for the IP address. But the lookup process also checks if there is a longer prefix. As $child\_arr_3[2] > 0$; it will continue the lookup process with the arrays in level 6. $child\_arr_3[2] = 2$ indicates that it needs to lookup in child 2 (in level 6). As $a_{(3,5)} = 5$, $solid\_arr_6[5]$ of child 2 will contain the next-hop for the IP address (in this case, it is $P6$).

The main drawback of array based lookup is that it requires large arrays to be stored in CPU cache. As CPU caches are very limited in size, it often results in cache-miss which degrades lookup performance. Poptrie allows us to represent the arrays with bitmaps that reduces the memory requirement significantly. The arrays in Figure 1c are represented as Poptrie in Figure 1d. Here $bitmap$ is constructed by setting i-th bit 1 if $solid\_arr[i] \neq 0$. For instance, $\forall_i solid\_arr_3[i] \neq 0$ except $i \neq 3$. This is why, equivalent $bitmap$ is 11110111.

Note that level pushing results same prefix appears multiple times in the $solid\_arr$. For instance, there are two $P1$ and three $P2$ in $solid\_arr_3$. The problem is exacerbated if we build tree based on level 6, level 12, level 18 and so on. This consumes nontrivial amount of memory. Poptrie solves this problem by having an array where redundant elements and empty elements (that are 0) are eliminated. Array NH in Figure 1d is constructed from $solid\_arr$ by removing redundant elements and empty elements. Poptrie also construct an another bitmap $solid\_bm$ for indicating start of consecutive prefixes. For instance, $solid\_arr_3$ has four consecutive prefixes, $P1$, $P3$, $P2$, and $P5$ that starts at index 0, 2, 4 and 7 respectively. This is why those bits are set in $solid\_bm$. $solid\_bm$ is used to calculate the index to $solid\_arr$ for a give IP address. For instance, if $a_{(0,2)} = 5$, then $solid\_arr[POPCNT(11111b \,\&\, solid\_bm)]$ will contain the next-hop for the IP address. Finally $child\_bm$ is constructed by setting i-th bit if $child\_arr[i] \neq 0$.

4. POPTRIE IMPLEMENTATION

An IP address is 32 and 64 bit long in IPv4 and IPv6 respectively. Here we construct the Poptrie for IPv4 routing table. However the same idea is also applicable to IPv6. We construct poptrie for level 6, 12, 18, 24, 30 and 36. This is why bitmaps in the poptrie are 64 (2^6) bit long. This allows us to store them in 64-bit CPU register. Poptrie uses *POPCNT* instruction, a special CPU instruction that counts the number of bits set to 1. For instance, $POPCNT(11100101b) = 5$. Most of the modern CPUs can implement $POPCNT$ in $2-4$ clock cycle [4].

4.1 Look up Process

Algorithm 1 shows the look up process in a Poptrie. The process starts by extracting 6 bytes from the least significant bits of the destination IP address (Line 5). It then creates a bitmap (bm) and bit mask ($mask$) based on the extracted value (Line 8 and 9). For instance, if $a_{(0,5)} = 12$, then $bm = 1000000000000b$ and $mask = 111111111111b$. The bm and $mask$ are used to find the corresponding next hop and child node. Line $1 - 142$ checks if there is a solid node for the extracted value. If there is, it finds the next-hop. It then checks if there is a child node for the IP address (Line 17). If there is a child node, it repeats the same process with the child node. If however it does not find the child node, it indicates that the next-hop found in Line 14 is the desired next-hop. Note that, Poptrie does not need to backtrack in order to find the longest-prefix. This is because in each step we check if there is a solid node; if yes, then we save the corresponding next-hop. This next-hop will be replaced if we find a solid node with longer prefix.

Algorithm 1 Routing Table Lookup

Input: Poptrie root $root$
Input: Destination IP address of the packet ip
Output: Next-hop $nexthop$

```
 1: procedure LOOKUP($ip$)
 2:     $N \leftarrow root$
 3:     while True do
 4:         /*Extract 6 bytes from $ip$ */
 5:         $index \leftarrow ip \,\&\, 63$
 6:
 7:         /*construct bitmap and bitmask */
 8:         $bm \leftarrow 1ULL << index$
 9:         $mask \leftarrow bm - 1$
10:
11:         /*Find corresponding solid node*/
12:         if $N.bitmap \,\&\, bm$ then
13:             $i \leftarrow POPCNT(N.solid\_bp \,\&\, mask)$
14:             $nexthop \leftarrow N.NH[i]$
15:
16:         /*Find corresponding child node*/
17:         if $N.child\_bm \,\&\, bm$ then
18:             $i \leftarrow POPCNT(N.child\_bm \,\&\, mask)$
19:             $N \leftarrow N.child\_arr[i]$
20:             /*Need to extract next 6 bytes*/
21:             $ip \leftarrow ip >> 6$
22:             Continue
23:         return $nexthop$
```

5. REFERENCES

[1] H. Asai and Y. Ohara. Poptrie: A compressed trie with population count for fast and scalable software IP routing table lookup. In *ACM SIGCOMM 2015*.

[2] Data Plane Development Kit (DPDK). `http://dpdk.org/`.

[3] W. Eatherton, G. Varghese, and Z. Dittia. Tree bitmap: hardware/software IP lookups with incremental updates. *ACM*

SIGCOMM, 34(2):97–122, 2004.

[4] A. Fog et al. Instruction tables: Lists of instruction latencies, throughputs and micro-operation breakdowns for Intel, AMD and VIA CPUs. *Copenhagen University College of Engineering*, 97:114, 2011.

[5] S. Nilsson and G. Karlsson. IP-address lookup using LC-tries. *IEEE Journal on selected Areas in Communications*, 17(6):1083–1092, 1999.

[6] RIPE Network Coordination Centre. `https://www.ripe.net/`.

[7] Vector Packet Processing (VPP). `https://wiki.fd.io/view/VPP`.

[8] XDP (eXpress Data path). `http://prototype-kernel.readthedocs.io/en/latest/networking/XDP/introduction.html`.

[9] T. Yang, G. Xie, Y. Li, Q. Fu, A. X. Liu, Q. Li, and L. Mathy. Guarantee IP lookup performance with FIB explosion. In *ACM SIGCOMM 2014*.

[10] M. Zec, L. Rizzo, and M. Mikuc. DXR: towards a billion routing lookups per second in software. *ACM SIGCOMM Computer Communication Review*, 42(5):29–36, 2012.