Performance evaluation of multiway trie for longest common prefix matching in Data Plane

Anonymous

Abstract—With the increase in the popularity of the internet, the size of routing tables are increasing. The packet forwarding from a software switch or a software router with increasing routing table size has become a big problem. This paper discusses the implementation of the multi-way trie as a solution for longest common prefix matching on a software switch. The performance of the multi-way trie is evaluated with the integration of the trie structure with the Vector Packet Processing (VPP). This will help us to analyze the throughput of the VPP when integrated with the trie.

Index Terms—IPv6; Data Plane; FD.io VPP, Longest common prefix, m-way trie.

I. INTRODUCTION

Routers are responsible for sending information packets from source to destination. These routers are configured with routing tables which have the information to find the next hop needed for the packet to successfully reach its destination. Each machine or system that is the part of the network has an assigned address. This address is unique and used to identify the machine, just like our house address. In networking, we have two addressing techniques, IPv4 and IPv6. IPv4 is a 32-bit address and is commonly used. Since the length of IPv4 is 32 bits, we can have $2^{32}$ possible IPv4 addresses. Due to the increase in the popularity of the internet services, we need to use a larger IP address range. IPv6 has $2^{128}$ possible values as its length is 128 bits. All the routing algorithms that compute the next hop in the path of a packet resides in the router. Routing devices do not store any address and instead have routing rules to compute the next hop. For this reason, they store network prefixes and not individual IPs and the lookup procedure computes the longest common prefix to determine the next hop.

In modern times, almost all companies are using commodity hardware which is very cheap. This encourages to write software-based routers and switches that allow us to take the leverage of this commodity hardware to reduce the cost. With the introduction of ubiquitous internet technologies like the cloud and the Internet of Things, the consumption of the internet has lead to the increase of routing table size. While writing software solutions we need to address this problem. The solution should include being able to search large number of entries in the table quickly irrespective of the growing table size. A switch or router has two planes: the data plane and the control plane. Conventionally both of them are implemented in the firmware of the hardware devices. In this project, the focus is to understand how the multiway trie can be used to program the data plane for IP lookup, specially for IPv6.

Cisco has implemented and introduced a framework that functions like a switch or a router with high-class production quality. This framework is called VPP, Vector Packet Processing. VPP is an open source framework and is capable of processing packets with groundbreaking high performance and can easily be deployed on commodity hardware[7]. This framework has a great level of modularity and flexibility which makes it easier for developers to integrate their module to work with VPP. The working and architecture of VPP are described in the later section. In this paper, I will discuss various filter-based and binary trie-based solutions that can be used to speed up the lookup process by integrating it with VPP. Later we will discuss the behavior of the multiway trie when integrated with VPP.

II. BACKGROUND

In this section, we will first discuss Vector Packet Processing (VPP), its architecture and working. This section also extensively describes the internal architecture of Bloom Filter and Cuckoo Filter. This will be further augmented by the understanding of the Binary trie data structure.

What is VPP?

Vector Packet Processing is a framework developed by Cisco under the project FD.io (Fast Data -Input/Output) which provides the functionality of a router/switch. As the name suggests, VPP processes a vector of packets together at a given time rather than processing an individual packet[7]. Processing a vector of packets at a time maximizes the effect of cache locality which optimizes the CPU’s instruction cache.

Architecture of VPP:

The VPP creates graph nodes for processing the vector of packets. The input node uses Intel’s DPDK technology which quickens the input processing of the incoming packets. After receiving the packet from the input node, it then decides to which graph node it should delegate the process. If it is an IPv4 type packet, then it is given to the IPv4-input node. Similarly, it handles IPv6-input and arp-input. Once received at the IPv4-input node, it is sent to the lookup graph node which runs the algorithm to lookup and determine the next hop for that packet. Similarly, the IPv6-input node will send the received packet to the IPv6-lookup graph node. Once lookup is done, the packet is sent to the write-transmit node which forwards the packet to the next hop. All these nodes are loosely coupled[7] and these nodes make the system modular. This modularity enables plugging in a new node to the graph to process a packet in
a particular fashion. Each node contains the logic to handle a particular kind of packet. This makes VPP very flexible and modular in nature. This project aims at plugging in a node that runs the filters or trie algorithms for IPv6 packets. The new node addition does not require any core or kernel level changes[7]. Figure 2 shows the ‘packet processing graph’[7] of VPP.

Research on the use of various filters or trie to compute the longest common prefix for an IP has been done previously. In this section, we will discuss those filters and binary trie.

A. Cuckoo Filter:

**Working of cuckoo filter:**

Cuckoo filter is an approximate set membership test data structure[2]. Such a data structure can be used to lookup targets in a very large datasets. Hence, this can be used in databases, cache lookup, etc. A cuckoo filter is composed of a list of buckets. Fingerprints are stored in these buckets. The idea of generating the fingerprint will be discussed later. To insert an item x in the cuckoo table, two hash functions h1(x) and h2(x) are computed. The value of h1(x) and h2(x) computed earlier will help us to determine the location in the list of buckets where the fingerprint of x can be inserted. If one of the h1(x) or h2(x) buckets is available, we compute the fingerprint of x and insert that fingerprint in the available location[2]. If both the locations are occupied by other fingerprints, purging any one of the fingerprints present at h1(x) or h2(x) are computed. The value of h1(x) and h2(x) computed earlier will help us to determine the location in the list of buckets where the fingerprint of x can be inserted. If one of the h1(x) or h2(x) buckets is available, we compute the fingerprint of x and insert that fingerprint in the available location[2]. If both the locations are occupied by other fingerprints, purging any one of the fingerprints present at h1(x) or h2(x) are computed. The value of h1(x) and h2(x) computed earlier will help us to determine the location in the list of buckets where the fingerprint of x can be inserted. If one of the h1(x) or h2(x) buckets is available, we compute the fingerprint of x and insert that fingerprint in the available location[2].

We keep doing this process until we end up fitting the displaced fingerprint in the filter or we reach a pre-decided threshold displacement value. In this scenario, the size of the cuckoo list is increased in order to fit all the elements in the buckets. The author in[2] has used a nest of buckets instead of using a single bucket at every location. The lookup procedure for cuckoo filter is intuitive. To search item ‘x’ in the cuckoo filter, first we must compute the fingerprint along with two hash function values. Lookup in the nested buckets at location computed by these hash functions. If fingerprint is present then return true, else false. This is a probabilistic data structure as it may give false positive output at truth values. Cuckoo is quite efficient as its size is very small compare to the Bloom filter.

**Cuckoo filter with VPP for IPv6:**

IPv6 in binary format is used to compute the network prefix with given prefix value. Once network prefix is computed, CRC(Cyclic Redundancy Check) hash is applied to it to compute the hashed fingerprint. First 16 bits of the CRC hash are considered as the fingerprint. Then, hash1 and hash2 are computed to determine the location of buckets where this fingerprint can be stored. It will be inserted in the similar fashion as standard cuckoo hashing discussed earlier. For lookup, it starts from largest prefix for a given IP and true output of cuckoo is checked against the VPPs FIB(Forwarding Information Base) table. The reason to start from the largest prefix i.e. 128 is to obtain the longest common prefix match. This way the lookup is done for every IPv6 packet received in the cuckoo graph node. This helps to reduce the number of lookups in VPP. Since VPP lookup table i.e. FIB table is in main memory and cuckoo filter resides in cache, it helps to speed up the entire lookup process.

B. Bloom filter

Bloom filter is another set membership query data structure just like Cuckoo filter. Unlike Cuckoo filter, Bloom stores bits of 0s and 1s rather than storing the fingerprint[3]. Also Bloom filter has more number of hash function computations.
**Working of Bloom filter:**

In Bloom filter, an integer array is defined with a predefined size. To insert 'x' in bloom filter, hash values are computed by applying different k-hash functions. Each value of hash will tell the location in the array. At each of these locations, '1' is inserted to mark the presence of x. Similarly, for lookup the same hash functions are applied to the lookup query. For every hash output, array is checked for 1’s. If a '0' is found it will ensure that the queried data is absent in Bloom filter.

If '1' is found it may be there, as it is a probabilistic data structure. In the below section bloom filters working for the specific problem i.e. IP lookup is explained.

### Bloom filter with VPP for IPv6:

To insert a network prefix in the Bloom filter, binary value for the given IPv6 is computed. The corresponding binary value is used to obtain the network prefix with the given prefix value. The 7 hash functions are applied to obtain respective 7 locations to insert 1’s in the filter array. Similarly all the entries are inserted in the Bloom filter. At the time of lookup we are searching for longest common prefix. Then starting from highest prefix value (starting from 128), find the network prefix with this highest value. Then one by one all 7 hash functions are applied. If a '0' is encountered in any of the hash locations computed we stop there and conclude that prefix length is not present in the FIB. If all hash locations give '1', then only we look up that IP with that prefix in FIB. This way the number of FIB lookups can be reduced.

As discussed earlier, FIB table resides in main memory and main memory lookup is a costly operation. Filter based approach reduces the number of main memory lookups. As the size of Bloom filter is less to fit it in cache, it speeds up the entire process. From the experiments, it has been found that hash computation is an extra overhead in case of Bloom filter as the number hash computations are more in than compared to Cuckoo. Hence, Cuckoo filter preforms better when number of required hash computation is more in Bloom. Bloom filter can prove to be better than Cuckoo in the particular cases when first or second hash function of Bloom return '0' for most cases.

#### C. Binary trie

The Binary trie is a simple implementation of trie data structure with only two children nodes. Working with a binary string, every '0' corresponds to left and '1' to right child. To insert data in Binary trie, we convert the input IPv6 into binary representation and compute its network prefix with the given prefix value. This computed network prefix will be inserted in the trie following the same rule of going left for '0' and right for '1'. We keep decrementing the prefix value by 1 as we go down in the trie. As the prefix value reaches '0', we mark it as leaf by setting a flag for this node. Same methodology will be applied for lookup of the longest common prefix for an IP. In lookup we keep traversing the trie by applying DFS and longest common prefix is found as we reach the null node or a leaf node. The performance of binary trie is lesser than other filters because the size of the trie created is too big to fit in the cache.

### III. Methodology

This project focuses on implementing and evaluating the performance of the multiway trie for IPv6 lookup with VPP. In a m-way trie, i.e. multiway trie, each node can have $2^m$ number of children nodes based on the value of m. For example, if m is 2, each node can have $2^2 = 4$ children, for m = 5, each node in the trie can have $2^5$ or 32 children. The value of m decides the number of bits of IPv6 address to be considered in a single stride. We compute the decimal value of the considered bit stride and check if the corresponding reference already exists as a child node. If it exist, we traverse to that node and repeat the process for the next stride of bits from the binary IPv6 address. If node is not present at the decimal reference location, we add a child node to the current node.

While considering every stride we keep decreasing the value of prefix length by 1 for each bit of the binary IPv6 address. During this process if the prefix length becomes 0, there are two possible scenarios, one where exactly m number of bits of binary IPv6 are considered in the last stride and the prefix reaches 0 (i.e. the case where prefix length is multiple of m in m-way trie). In this case, we compute the decimal value of the bit stride, add the child node and mark the last node as leaf. In the second scenario, where the prefix is not the multiple of m, things become a little bit complex. In this scenario, while considering the last m bits, the value prefix becomes 0 before considering all m bits. This is the case when considering last n bits (n<m) where the value of prefix becomes 0. To handle this, we add (m-n) zeros at the end of the considered n bits. Computing the decimal representation of these prepared bits and finally adding a child node to that decimal reference location. By marking this new node as true leaf, a problem arises that the system will be unaware of the fact that n number of prefix bits are considered for a case.

To counter this problem, a new flag is being set which stores the broken prefix length which is n. In the first scenario, discussed earlier, this broken prefix length will have value -1. By doing this, while traversing the system can figure out that this is not a true leaf but it is a broken leaf node prepared by considering n bits of the IPv6 address. Also, the parent of the newly added node should set a bit showing that one of it’s child node is a broken prefix node where the broken prefix value can be found.

**Insertion:** Insertion creates the m-way trie on which the lookup function processes the various IPv6 addresses and is further explained in the example below. The insertion algorithm is described in Algorithm 1.

The example below shows the steps to insert an IPv6 address in a 3-way trie. To insert the IPv6 7a2c:0015::11 in a 3-way trie, compute the 128 bit binary representation of the given IPv6 address. The binary representation of the example IPv6 will be,
Now we compute the network prefix using the given prefix length which is 11. The network prefix of the above computed binary IPv6 address will be,

```
0111101000100000000000000000000000000000000000000
```

To insert this IP in a 3-way trie, take first 3 bits, compute the decimal representation of first three bits which is 3 (011), and then add a child node at position 3. We subtract 3 from the prefix. Move the current node pointer to this newly added node. Then we consider next 3 bits and repeat the process. When the system is done considering 9 bits of the network prefix (011110100), the value prefix left to consider is 2. Since the system has only 2 prefix left and we are considering 3 to insert, the lookup will give the wrong prefix value. While lookup, the system should return 11 as the longest prefix length while the system will return 12. To handle this, we consider only 2 bits and add one '0' to make it 3-bit stride. The decimal value for 010 will be 2 where the last 0 from the right is a padded bit. Add child to the second reference. The current node which is the parent of the new node that was just added, will adjust its bits to mark that it will have broken prefix children. The newly added node will store the broken prefix length which is 2. This can be tracked while traversing. The 3-way trie for this example is shown in Figure 3.

**Algorithm 1:** Insertion in m-way trie

```
current = root
while not the end of line and prefix > 0 do
  if prefix >= m then
    Decimal = decimal value of next m bits
    prefix = m
    bits_considered = m
  else
    Decimal = decimal value of remaining n bits
    prefix = n
    bits_considered = n
  end

  if current->child[Decimal] not null then
    prev = current
    current = current->child[Decimal]
    if bits_considered is not m then
      prev->hasBrokenPrefix = true
      current->brokenPrefix = n
    end
  else
    new_child = create_new_node()
    current->child[Decimal] = new_child
    prev = current
    current = current->child[Decimal]
    if bits_considered is not m then
      prev->hasBrokenPrefix = true
      current->brokenPrefix = n
    end
  end
  if prefix is 0 then
    current->isLeaf = true
  end
end
```

**Lookup:** While running lookup the system will consider m-bits of the lookup IP address in each stride and traverse the trie. We do not stop if we reach a leaf node, we just store the value of current prefix in a variable and continue the traversal as we can find a prefix larger than the current prefix. Stop only if a null node is encountered while traversing. Two solutions for the longest common prefix lookup are proposed in this paper: top down approach for lookup (iterative) and bottom up approach (recursive).

In the top down approach, i.e. iterative approach, the system starts from the root node and keep traversing down. Every time we encounter a node marked as the leaf node, store the value of current longest common prefix as the value of bits of input IP considered until now. For the nodes marked as having broken prefixes, we traverse the lookup IPv6 bit by bit and store the longest common prefix values as the largest value of broken prefix. We keep traversing till we encounter a null node to achieve the largest possible prefix length. The largest value of longest common prefix stored in earlier steps will become the output.

In the bottom up approach, i.e. recursive approach, the system keeps traversing down by considering m bits at a time until it reaches the null node. Then the system start to backtrack and the first leaf node it finds while backtracking will give the longest common prefix length. The leaf discussed in the either step can be a true leaf or a broken leaf node. The benefit of the bottom up approach over top down is that the system does not have to traverse bit by bit for every
broken prefix node while traversing. Rather the system might just has to traverse bit by bit for just one node. This increases the throughput of the lookup as the number of computations required for bit by bit lookup is reduced. Bottom up lookup algorithm is described in Algorithm 2.

NOTE: Code implementation available at [https://github.com/yj29/multiway_trie](https://github.com/yj29/multiway_trie)

**Algorithm 2: Bottom up lookup in m-way trie**

```plaintext
current = root
start_index=0
lookup_IP = binary representation of lookup IPv6

if current is null then
    return 0
end

temp = 0
Decimal = decimal value of next m bits starting from start_index
prefix = lookup_recursive(current− > child[Decimal],
start_index + m)
if prefix > 0 then
    return prefix
end

if current− > isLeaf is true then
    temp = start_index
end

if temp is not 0 then
    return temp;
end
else
    if current− > hasBrokenPrefix is true then
        return start_index + longest broken prefix
    end
end
return 0
```

**IV. RESULTS**

To compute the results, the m-way trie implementation explained in the above section is integrated with Cisco’s Vector Packet Processing (VPP). The results are computed on two datasets, DT [8] and RV [9]. DT data has 17 different prefixes while RV data has 34 different prefixes. Test results are obtained by testing against 500,032 lookup IPs for both DT and RV datasets respectively.

From Figure 4, it is evident that the performance of m-way trie is optimal with value of m as 2. The number of packets processed for DT and RV is almost same for 2-way trie. The result shows a decline of performance with an increase in the value of m. Though the height and number of nodes per trie decrease with the increase of m, but at the same time, the computation also increases which results in a decline of the number of packet processing rate.

Additionally, data was captured for the top down lookup approach. Figure 7 and Figure 8 shows that the bottom up approach is working faster than the top down approach. This is because in top down approach the number of computations increases, as for every broken prefix node, we have to do a bitwise check to obtain the longest common prefix.

The key thing to notice that as we increase the value of m, we consider more bits of the IPv6 while trie construction hence, more the value of m, less will be the number of nodes in the trie. Figure 5 also shows the same results. In the figure, it is clear that with the increase in value from 2 to 6 the total number of nodes are decreasing.

The number of nodes decreases but the size of the overall trie will increase as the number of references that every node has to carry for its children also increases. For m with value 2, the size of a single node is 40 bytes. Size of each node is 72 bytes, 136 bytes, 264 bytes and 520 bytes for m 3, 4, 5 and 6 respectively. For m = 6, the size of trie data structure reaches more than 1GB. Figure 6 illustrates that the entire trie structure cannot fit into cache memory as its size is beyond the cache memory. Hence the performance m-way trie is slower than other filter approaches like Cuckoo and Bloom as these filters footprints are small enough to fit in the cache memory.
V. CONCLUSION

The extensive testing of the datasets provided an in depth analysis of the m-way trie data structure and the algorithms proposed in this paper. Since the datasets provided distinctive input and output labels, testing produced accurate results. The bottom up approach works better than top down approach as it requires less computation while lookup process. 2-way trie gives the best solutions in both top down and bottom up approach. This paper provided a premise for pop-trie data structure that further enhances the performance as the data structure compresses the size of nodes. This enables cache memory to more efficiently handles the pop trie data structure.

ACKNOWLEDGMENT

I would like to thank - for his guidance and giving me the opportunity to work on this on-going research project. This project helped to enhance my knowledge about Computer Networks. The work on this Capstone project was carried out at -.

REFERENCES