# Comparison of Efficient Routing Table Data Structures

Dominik Schöffmann
Betreuer: Sebastian Gallenmüller
Betreuer: Paul Emmerich
Seminar: Future Internet WS2016/17
Lehrstuhl Netzarchitekturen und Netzdienste
Fakultät für Informatik, Technische Universität München
Email: schoeffm@in.tum.de

## ABSTRACT
The topic of this paper is to compare multiple IP address longest prefix matching lookup schemes, especially algorithms implemented in software. Goals which are key to constructing an efficient lookup algorithm are explained. Different data structures and their respective approaches to size compression get illustrated. These approaches include a hardware based algorithm called "DIR-24-8" for better reference, and multiple software based algorithms. Different trie structures are reviewed in detail. Tests about lookup acceleration were conducted by the author and are presented and discussed in this paper.

## Keywords
routing, lookup, IP, trie, dxr, poptrie

## 1. INTRODUCTION
"The datagrams are routed from one internet module to another through individual networks based on the interpretation of an internet address." [1] This quote from the IPv4 protocol specification dating back to 1981 still is the core of our modern Internet as we know, and use, it today.

Nowadays, routers within Internet Exchange Points (IXPs) are announcing routes to specific subnets, to each other, thus compiling comprehensive databases of next hops to those subnets. Once the packets enter an autonomous system, other routing protocols take over and guide the datagram to its destination. Oftentimes, there is talk about these algorithms used to exchange routes, like BGP and OSPF, whereas the methods of actually using this information is regularly hidden in a black box fashion. Either specialized hardware from well-known vendors such as Cisco, Juniper or Huawei is found inside the datacenters, or just plain general purpose x86 hardware using an open source Operating System (OS). The almost definitely most used OSs are Linux and FreeBSD. Both are highly reliable, well tested and deployed in a multitude of places around the world.

This paper aims to give an overview of popular and fast data structures which are either implemented in the aforementioned OSs, or are available in specialized libraries. These libraries are particularly interesting in combination with modern high performance packet frameworks such as netmap [2] or Intel DPDK [3]. As a matter of fact, Intel DPDK does include an implementation of a modified version of the DIR-24-8 algorithm which is explained later [4]. The pfSense router distribution is currently planning to deploy netmap in combination with a path compressed trie, which also is explained later [5].

## 2. RELATED WORK
Various authors published research regarding routing lookup data structures. Some dating back into the last millennium others were just recently presented. Since this paper gives details about the various algorithms, previous research is not presented in depth at this point.

Gupta et al. [6] developed a hardware optimized routing algorithm called DIR-24-8. This algorithm is not in use today in its original form, but a basic understanding is useful nevertheless.

Nilsson and Tikkanen [7] published a research paper about dynamic tries and compression techniques which might be utilized. Their work deeply influenced the lookup scheme of the Linux kernel [8].

The trie which is used for lookups within the FreeBSD kernel is described in a book written by McKusick and Neville-Neil [9].

More recently, specialized library implementations were developed and presented. The DXR algorithm is described by Zec el al. [10], which borrows some aspects from the DIR-24-8 algorithm, but performs better in software.

Poptrie is the fastest software based routing lookup algorithm presented in this paper. It is the only data structure utilizing specific instructions available in modern CPUs. It was published by Asai and Ohara [11] in 2015.

## 3. OPTIMIZATION GOALS
In order to build a fast routing lookup data structure, multiple factors are important. All of them are direct consequences of the hardware architecture, most notably the CPU, in use.

If the data structure is too big to fit inside the CPU cache, main memory accesses are needed. Such accesses introduce a high latency, during which no work inside the processing thread can be performed. Therefore the core might idle and thus waste resources.

Another important element is the number of memory accesses which need to be performed within one lookup. Even


doi: 10.2313/NET-2017-05-1_03

if the data structure is small enough to fit into the L3 cache, getting the data from there might still be expensive. If data needs to be fetched from the L3 cache, it is better to have as few of such accesses as possible. Modern cache lines store 64 bytes each, therefore all accesses transfer 64 bytes in one step. It is more efficient to store more data in the same cache line and thus utilize the size of a cache line, than to distributing it across multiple cache lines.

Furthermore, specifics about the targeted CPU architecture can also be of use, in order to build efficient data structures. This includes single instructions as can be observed with the poptrie algorithm. [11]

# 4. HARDWARE LOOKUP

The vast amount of traffic flowing through datacenters and IXPs are handled by hardware based routers. Those routers contain application-specific integrated circuits (ASICs) which are built to allow fast routing lookups. When reviewing routing algorithms, it is important to keep the differences of the underlying hardware in mind, in order to choose which method fits the given platform best.

## 4.1 DIR-24-8

The DIR-24-8 is a hardware based routing lookup algorithm, which was first introduced by Gupta et al. in 1998 [6].

The basic design uses three distinct tables, which are called "TBL24", "TBLlong" and "Next Hop". Routes with a prefix length of equal or less than 24 bits are completely stored in the TBL24, prefixes longer than that need to use the TBLlong. In the next hop table the IP addresses of all the next hops, which might be used by any route are stored. An IP address which is to be routed using this scheme is split into two parts, the first 24 bit and the last 8 bit. The upper part is used to index the TBL24 which contains $2^{24}$ entries, whereby each entry can have two different kinds of content. Depending on the routes, there might, or might not, be a route having a prefix length of more than 24 bit, sharing the first 24 bit with the IP address for which the lookup is performed. In this case the TBLlong is used. [6]

If no such route exists, the entry's most significant bit is set to 0. The 15 least significant bits are then used to index the next hop table. In this case, the lookup is finished. [6]

Whenever there is a route with a prefix length, longer than 24 bit, the table TBLlong is needed. This is indicated by a 1 in the most significant position of the corresponding TBL24 entry, which then also contains an offset into the TBLlong. Each of the 256 possible options of the last 8 bit of the IP address are represented by their corresponding next hop inside of the TBLlong. As soon as the next hop index is extracted from the TBLlong and used to index the next hop table, the lookup is finished. [6]

The DIR-24-8 algorithm is suitable for hardware implementations for multiple reasons. Oftentimes the next hop is found using a /24 entry of the routing table. All of these entries are stored in the TBL24, which furthermore has a fixed size. Therefore this table can be realized in hardware with a high-throughput and low-latency link to the CPU. The other two tables usually are not very large and thus fit into a reasonably sized RAM.

There exist multiple extensions to this data structure in order to save memory or memory accesses, and thus speed up the lookup. [6]

# 5. SOFTWARE LOOKUP

Although a significant part of the Internet's backbone infrastructure is powered by hardware routers, software lookup algorithms play an important role. As software routers and general purpose CPUs gain maturity and speed, they may be viable options in the future.

## 5.1 Naive

The naive lookup algorithm is quite simple. All of the routes are contained inside of a list, which is sorted by the prefix length. Longer prefixes are at the beginning of the list, shorter ones at the end.

Every time an IP address is to be routed, this list is traversed until the first match. In the worst case, this is the default gateway. A match is computed as follows:

$$(IP\&(\sim (2^{32-length}) - 1)) \oplus prefix \tag{1}$$

whereby *length* denotes the prefix length, & a binary AND operation, $\sim$ the one's complement, and $\oplus$ a binary XOR operation. If the result is zero, the route matches and the next hop is found. In case it is not zero, the next entry needs to be checked until a match is found.

## 5.2 Trie

Most modern lookup algorithms use some kind of tree as their basic data structure. Commonly, this tree is in fact a radix tree, which is also called a trie. The invariant of a trie is, that all the child nodes of a node share a common prefix, which is the key of the node.

Tries can be used is a number of different fashions, meaning that there is not one single valid basic lookup scheme using tries. The algorithm presented here is based upon the author's intention of simplifying the basic algorithm and the work of Nilsson and Tikkanen [7].

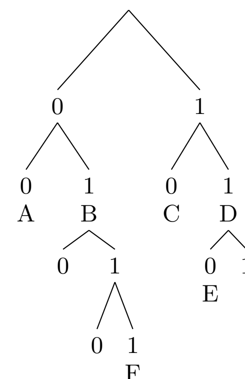



**Figure 1: Basic trie**

Figure 1 shows a basic trie, demonstrating the prefix sharing. The corresponding routing table if given in Table 1. The


doi: 10.2313/NET-2017-05-1_03

| Node | Route | Next Hop |
|---|---|---|
| A | 0.0.0.0/2 | 1.2.3.4 |
| B | 64.0.0.0/2 | 2.3.4.5 |
| C | 128.0.0.0/2 | 3.4.5.6 |
| D | 192.0.0.0/2 | 4.5.6.7 |
| E | 192.0.0.0/3 | 5.6.7.8 |
| F | 112.0.0.0/4 | 6.7.8.9 |

**Table 1: Basic trie, Routing table**

terms "left" and "right" are used to indicate a 0-bit and a 1-bit respectively. Inside of the trie, a route matches at exactly the position corresponding with the prefix.

When looking up an IP address, the n-th most significant bit of the address is used to index the n-th level of the trie. The trie is traversed using this fashion, until a leaf is found. Once this is achieved, the route at the leaf might match the IP address in question, or might not match it. If it does match, the search is over. If it does not match, the trie needs to be backtracked upwards again.

For example, if the IP address which is to be looked up is 128.0.1.24, the algorithm would take the first branch to the right, and the next branch to the left, reaching the leaf labeled C. Indeed this route matched the IP address, and the lookup was successful at this step.

Another example is the address 96.4.5.6, which would branch left, then two times right, and left again. The leaf which is found does not contain any route. Since the algorithm did not reach a leaf which corresponds with a route matching the IP address, upwards traversal of the trie is needed. As soon as the node B is rediscovered, the algorithm found the longest matching prefix and finishes.

It should again be noted, that this is not the only possible method of making use of a basic trie, but was chosen for simplicity. For example one simple extension would be to save the last found route, in order to avoid the upwards traversal.

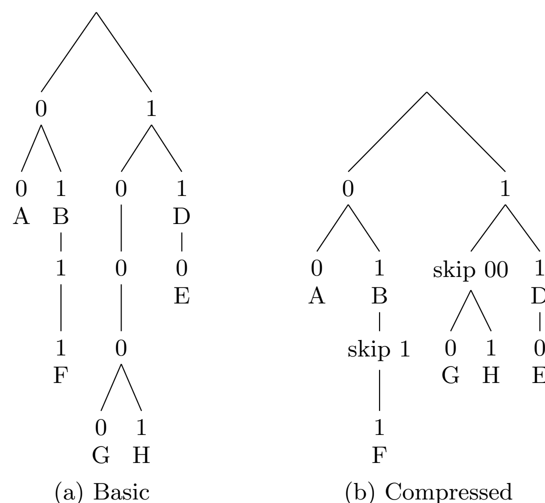
## 5.3 Path compression

When developing a routing lookup algorithm, the size of the data structure, and the number of memory accesses, are of big importance. One method to decrease the number of needed memory accesses is called "path compression". Again, different implementations exist, which utilize path compression in their own way. In the first part, the notation of the previous section is preserved, and the compression is influenced by Nilsson and Tikkanen [7]. The second part describes the approach of FreeBSD.

Path compression allows the data structure to skip intermediate nodes which only have one child each.

Figure 2 shows two tries, whereby Figure 2a is a none compressed trie and Figure 2b is the path compressed version, representing the same data. The routing table corresponding to Figure 2 is given in Table 2. Empty leafs are omitted for brevity.

So far the change is straight forward, by just adding an intermediate node, which represents a path of nodes which would only lead to this subtrie.




(a) Basic (b) Compressed

**Figure 2: Path compressed trie**

| Node | Route | Next Hop |
|---|---|---|
| A | 0.0.0.0/2 | 1.2.3.4 |
| B | 64.0.0.0/2 | 2.3.4.5 |
| D | 192.0.0.0/2 | 4.5.6.7 |
| E | 192.0.0.0/3 | 5.6.7.8 |
| F | 112.0.0.0/4 | 6.7.8.9 |
| G | 128.0.0.0/5 | 7.8.9.0 |
| H | 136.0.0.0/5 | 8.9.0.1 |

**Table 2: Path compressed trie, Routing table**

The FreeBSD Operating System actually uses such a path compressed trie as its IP lookup scheme, although in a different flavor. In the examples used here, routes could be internal nodes, whereas in the FreeBSD implementation, routes are always leafs. [9][12] The same routing table, as above, is also represented in Figure 3, using the FreeBSD implementation. It should be noted, that the numbers of the internal nodes depict bit positions, on which the trie branches. Furthermore, in this representation, two different routes sharing the same prefix, but having a different prefix length cannot be handled solely within the trie. As a solution, the nodes have a list of routes, which are checked in the order of the prefix length. Longer prefixes get checked first. Internal nodes can reference leafs, which is useful for the backtracking, and depicted in Figure 3 as the dashed lines.

Again, an example lookup may help to understand this data structure. The IP address to be looked up is 96.45.56.67. For simplicity, let's split the first octet up into its binary representation: $96_{10} = 01100000_2$ This means, that we need to take a left branch followed by two right branches, and left branches from there on.

Following these directions reveals the leaf F which does not match the IP address in question. The algorithm backtracks one level up, where is encounters a reference to the leaf B. B in return holds a route which matches the IP address. This terminates the algorithm.

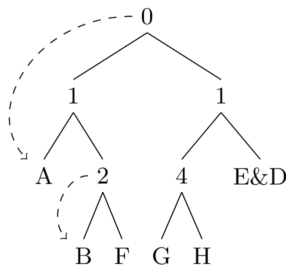



**Figure 3: FreeBSD compressed trie, Adapted from the FreeBSD book [9]**

Another interesting lookup yields the IP address 168.1.2.3. Again the binary representation of the first octet is helpful: $168_{10} = 10101000_2$, which means that the bits 0, 2, and 4 are set. In this case, the zeroth bit gets tested and evaluates to 1, so the search continues in the right subtrie. The second most significant bit (position 1) is 0, which means, that the left branch is taken. A 1 at position 4 means, that the route H is tested for a match, which is negative. Due to this, the backtracking process begins. The nodes labeled “4” and “1”, which get traversed in this order, do not reference any leafs. The process continues to the root node, which references a route which does not match. Since the root is already reached, there is no further node to visit. The lookup was unsuccessful, and the packet cannot be routed, because the routing table does not contain a route to this host.

It should be noted, that this rarely reflects the reality, since almost always a default gateway exists, which can be used as a last resort.

## 5.4 Level compression

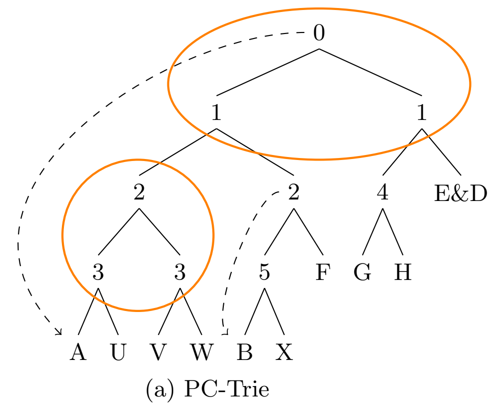
Although path compression is a pretty good start, for reducing the memory consumption and accesses, level compression can further help to achieve these goals. As the name already suggests, multiple nodes on the same level get merged, in order to have more information at the same memory location.

Saving nodes means, that less pointers to nodes need to be held, which is good for the memory consumption, and it also means, that less nodes need to be accessed to find the desired information, which is helpful to counter expensive memory accesses. The increased size of a single node is only a minor concern, because it usually still easily fits inside one cache line.

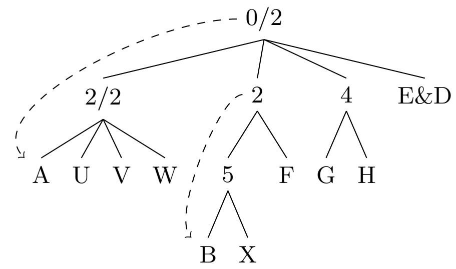
A node can be doubled when all of its children themselves have the same amount of children, and all children split at the same bit position for path compression.

Figure 4a shows an extended version of the FreeBSD trie used above. A level compressed version of this trie is given in Figure 4b. As can be seen, the level compression merges multiple nodes into a single one. The notation “x/y” of the internal nodes means, that this node starts comparing bits at position x for a length of y bits.

The internal structure of the nodes and therefore parts of




(a) PC-Trie




(b) LPC-Trie

**Figure 4: Level compressed FreeBSD trie, Adapted from the FreeBSD book [9], Nilsson and Tikkanen [7]**

the algorithm are dependent on the actual implementation to be used. The algorithm presented for path compressed tries still holds for this kind of level compressed trie, with only minor changes.

Although the FreeBSD trie does not use level compression, the Linux kernel does. [8] The Linux kernel furthermore adapts an optimization presented by Nilsson and Tikkanen [7]. Using the strict criteria presented in this paper imposes great cost on the maintenance of the trie. Abstaining from the demand that no child node is empty, but introducing thresholds as to when to double or halve a node allows for good compression at reduced cost.

## 5.5 Poptrie

Poptrie is a high performance IP lookup data structure based upon LPC tries, developed by Asai and Ohara [11]. It incorporates both path and level compression, as well as support for a new CPU instruction commonly found in commodity hardware, namely “popcount”. This instruction counts the bits set to 1 in an integer. The data structure is highly compressed in order to fit into lower level caches, compared to other schemes, which may have to make use of main memory or the L3 cache.

Poptrie maintains two arrays, one for the internal nodes, and one for the leafs. An internal node is a struct, which contains an offset into both of these arrays, and a field of bits. As usual the IP address is used as the key into the data structure which is divided into multiple parts, one for each level. The part of the IP address to be processed at the moment is used to index the bit field. If the corresponding


doi: 10.2313/NET-2017-05-1_03

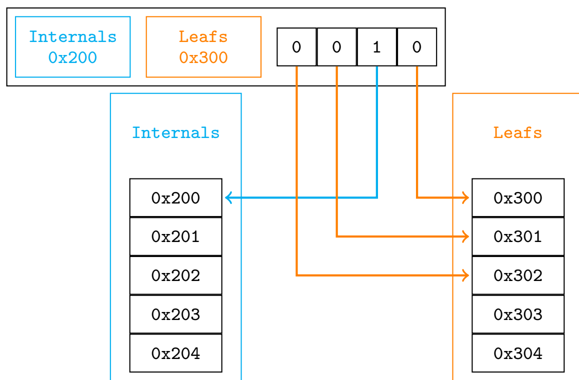



**Figure 5: Poptrie illustration(adapted from Asai and Ohara [11])**

bit is a 1, the next step is to use another node, if it is a 0, a leaf is reached. This mapping is illustration in Figure 5.

In case a leaf is not yet reached, but the trie is further traversed, the bit field is masked, to force the current index and everything of higher significance to 0. The remaining 1s in the field are counted. At this point the popcount instruction can be used to speed this process up. The resulting number is added to the offset into the array which is saved in the current node, and thus the next node is found.

As soon as a leaf will be accessed in the next step, the n-th least significant 0s are counted, whereby n is the current part of the IP address. Analogous to the internal nodes, this number is added to the offset and then yields the position of the searched leaf. The leaf contains the next hop to be used. This also means, that routes are projected similarly to DIR-24-8.

Multiple extensions are developed, which further reduce the size of the data structure and save traversal steps. The leaf vector extension reduces the number of leafs with the same content, by introducing a new vector inside the internal node. Another extension is direct pointing, which works by using the first x bits of the IP address to index an array of internal nodes and next hops. These internal nodes are anyways traversed for nearly all address lookups. This reduces the number of steps and memory lookups and thus increases the performance.

## 5.6 DXR

The DXR algorithm was developed by Zec et al. [10] in 2012. It has similarities to DIR-24-8 and the direct pointing extension of poptrie, but does not build upon a trie itself. Route projection, as with DIR-24-8 and poptrie is also used, meaning the discrete routes are expanded into ranges within the continues IP address space.

Three arrays are used inside of the DXR algorithm, which are called the "lookup table", the "range table" and the "next hop table". The lookup table is indexed using the first n bits of the IP address, thus having $2^n$ entries. Commonly used values for n are 16 and 18. An entry can have two different kinds of content. Either a route with a prefix longer than n exists, which thus cannot be handled inside the lookup table,

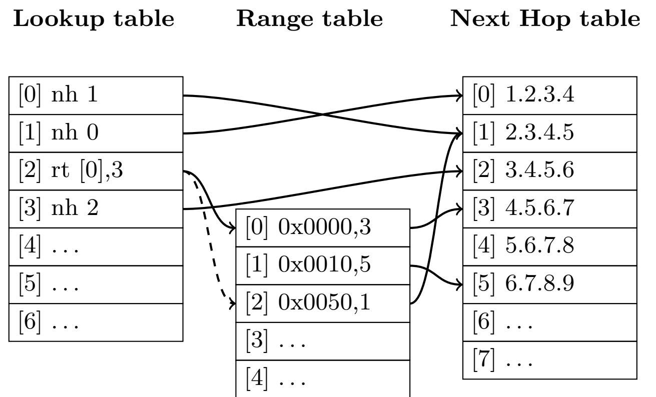



**Figure 6: DXR data structures, redrawn from Zec et al. [10]**

or such a route is not present. If there is no such route for a given n-bit prefix, the corresponding entry in the lookup table directly references an entry in the next hop table. This can be seen in Figure 6 for the entries 0x00, 0x01, 0x03.

In the likely case, that a route with a prefix length of more than n bits needs to be handled, the range table is used. Therefore the lookup table entry contains an offset into the range table, and the number of entries corresponding to this lookup table entry. An entry in the range contains the first IP address of the range, and the next hop responsible for the IP addresses between this first IP address (including) and IP address of the next entry (excluding). In order to find the correct range entry, a binary search algorithm can be used. An example for a range table is also included in Figure 6.

According to Zec et al. [10], DXR is, depending on the exact setting, up to 3.5 times as fast as a DIR-24-8 software implementation.

# 6. PRACTICAL EXPERIMENT

In the scope of this paper, the naive lookup scheme and a basic trie were developed and tested. The used dataset was obtained from the IXP in Amsterdam. Both algorithms were developed in an address-by-address and a batched fashion. Furthermore cache prefetching was evaluated. All tests were preformed on an Intel i5-5200U CPU, and GCC 5.4.0 was used. This CPU has a cache of 3 MB.

For the naive approach, neither optimization, batching nor prefetching, did yield any performance speedup. Using plain C arrays instead of C++ STL containers, and enabling compiler optimization, were very effective. These methods reduced the time needed to successfully route 100000 addresses from 10 seconds to 0.2 seconds.

For a basic trie the performance gain is still small, but noticeable. The time to route 10000000 addresses in dependence of the batch size is presented in Figure 7.

A batch size of one indicates the not-batched version of the algorithm. It can be observed, that the batch sizes 2, 16, and 32 have the best speed. In the end, the best enhancement


doi: 10.2313/NET-2017-05-1_03

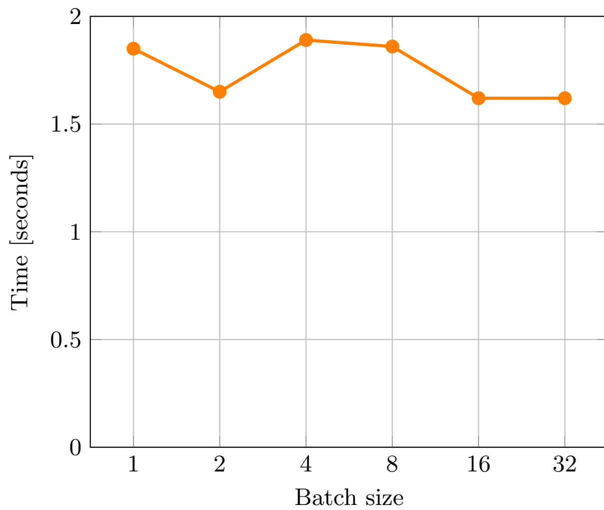



**Figure 7: Performance of the basic trie**

still is to enable the compiler optimization, which produced a speedup of factor 3.1.

# 7. CONCLUSION

Various algorithms for routing lookups and their respective data structures are described in this paper.

Prominently tries are used for real-world software routing, while the fastest known algorithm is also based on a trie. Multiple compression techniques exist such as path compression and level compression, which can help to reduce the size of the trie. Caches can be used more efficiently this way, since the memory accesses are cheaper, if the data is already in a low-level cache.

Apart from tries, the DIR-24-8 hardware based algorithm, and the DXR algorithm are presented. Both utilize multiple tables in order to split the IP address into two parts and use the content of the tables to access a next hop table.

An experiment based upon a naive lookup scheme and a basic trie implementation was performed. The result showed, that batching was of no significance for the naive algorithm, and of marginal use for the trie.

# 8. REFERENCES


[1] J. Postel. (1981, September) Internet protocol. [Online]. Available: https://tools.ietf.org/html/rfc791

[2] L. Rizzo, "netmap: a novel framework for fast packet I/O," in *USENIX Annual Technical Conference*, 2012.

[3] DPDK Developers. (2016, September). [Online]. Available: http://dpdk.org/

[4] Intel, *DPDK Programmer's Guide*, version: 16.04.

[5] J. Thompson, "pfSense around the world, better IPsec, tryforward and netmap-fwd," October 2015. [Online]. Available: https://blog.pfsense.org/?p=1866

[6] P. Gupta, S. Lin, and N. McKeown, "Routing lookups in hardware at memory access speeds," in *INFOCOM'98. Seventeenth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE*, vol. 3. IEEE, 1998, pp. 1240–1247.

[7] S. Nilsson and M. Tikkanen, "An experimental study of compression methods for dynamic tries," *Algorithmica*, vol. 33, no. 1, pp. 19–33, 2002.

[8] Linux Kernel Developers. (2016, October) Version: 4.7.2. [Online]. Available: https://cdn.kernel.org/pub/linux/kernel/v4.x/linux-4.7.2.tar.xz

[9] M. K. McKusick and G. V. Neville-Neil, *The Design and Implementation of the FreeBSD Operating System.* Pearson Education, 2004.

[10] M. Zec, L. Rizzo, and M. Mikuc, "Dxr: towards a billion routing lookups per second in software," *ACM SIGCOMM Computer Communication Review*, vol. 42, no. 5, pp. 29–36, 2012.

[11] H. Asai and Y. Ohara, "Poptrie: A compressed trie with population count for fast and scalable software ip routing table lookup," in *ACM SIGCOMM Computer Communication Review*, vol. 45, no. 4. ACM, 2015, pp. 57–70.

[12] FreeBSD Developers. FreeBSD src tree. Version: 9a3c 17b7 80af 5d03 15d0 d362 a209 8484 a37b a0ef. [Online]. Available: https://github.com/freebsd/freebsd